Verification with Memory Models as Input

Vom Fachbereich Informatik der
Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Florian Furbach

Datum der wissenschaftlichen Aussprache: 10.11.2020

Dekan:	Prof. Dr. Jens Schmitt
Gutachter:	Prof. Dr. Roland Meyer
Gutachter:	Prof. Dr. Klaus Schneider
Gutachter:	Dr. Georg Zetzsche

D 386

Abstract

To improve efficiency of memory accesses, modern multiprocessor architectures implement a whole range of different weak memory models. The behavior of performance-critical code depends on the underlying hardware. There is a rising demand for verification tools that take the underlying memory model into account. This work examines a variety of prevalent problems in the field of program verification of increasing complexities: testing, reachability, portability and memory model synthesis.

We give efficient tools to solve these problems. What sets the presented methods apart is that they are not limited to some few given architectures. They are universal: The memory model is given as part of the input. We make use of the `cat` language to succinctly describe axiomatic memory models. `cat` has been used to define the semantics of assembly for x86/TSO, ARMv7, ARMv8, and POWER but also the semantics of programming languages such as C/C++, including the Linux kernel concurrency primitives.

This work shows that even the simple testing problem is NP-hard for most memory models. It does so using a general reduction technique that applies to a range of models. It examines the more difficult program verification under a memory model and introduces Dartagnan, a bounded model checker (BMC) that encodes the problem as an SMT-query and makes use of advanced encoding techniques. The program portability problem is shown to be even harder. Despite this, it is solved efficiently by the tool Porthos which uses a guided search to produce fast results for most practical instances. A memory model is synthesized by Aramis for a given set of reachability results. Concurrent program verification is generally undecidable even for sequential consistency. As an alternative to BMC, we propose a new CEGAR method for Petri net invariant synthesis. We again use SMT-queries as a back-end.

Acknowledgements

First and foremost, I would like to thank my scientific adviser Roland Meyer. I am deeply grateful for his interest in my work, the insights I gained from him, and for everything I have learned from him. I am immensely grateful to have had such a supportive adviser who was always available for advice.

Moreover, I am honored and grateful that Klaus Schneider and Georg Zetzsche agreed to take the time to review this thesis.

Furthermore, I am thankful to my coauthors during my time as a doctoral student. I was privileged to collaborate with Maximilian Senftleben, Klaus Schneider and Roland Meyer on my first publication. I am grateful for their time and for everything they taught me about research and writing publications. In particular, I thank Maximilian Senftleben for the long hours we spend developing proofs together.

In 2016, I had the opportunity to spend two weeks in Helsinki to work with Keijo Heljanko and Hernan Ponce de Leon at Aalto University together with Roland Meyer. I am thankful for this great visit. It started a collaboration of exciting research in weak memory verification which was later joined by Natalia Gavrilenko and lasted years. I am grateful for the many stimulating discussions we had, the challenging questions they raised, and the many things I learned from them. Moreover, I am thankful for the heavy lifting that Hernan Ponce de Leon and Natalja Gavrilenko did in the tool development.

I thank Peter Chini and Roland Meyer for all the work we did together studying Petri net invariants. I am grateful to Chrisitan Eder for lending us his expertise and to Marvin Triebel for all his advice and the question he raised during his visit.

Moreover, I am glad to have been part of the Concurrency Theory Group, first in Kaiserslautern with Georgel Calin, Egor Derevenetc, Georg Zetzsche, Reiner Hüchting, Emanuele D'Osualdo, Klaus Madlener, and Roland Meyer and then later as the Institute of Theoretical Computer Science at TU Braunschweig with Peter Chini, Sebastian Wolff, Sebastian Muskalla, Elisabeth Neumann, Prakash Saivasan, Mike Becker, Sebastian Schweizer, and Jürgen Koslowski. It is an amazing group and it was a pleasure to work in it. I fondly remember many interesting debates and fun times together.

I am grateful to everybody who read the draft of the thesis and gave feedback, especially to my father for his thorough review. I thank my family and friends for all their support and encouragement.

Contents

Chapter 1

Introduction

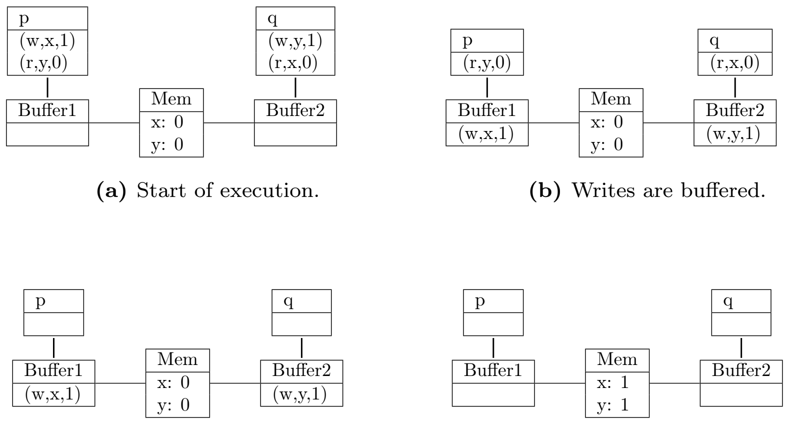
For multiprocessors, communication over shared memory is a performance bottleneck. To improve processor utilization, modern architectures implement various optimizations like a distributed shared memory or write buffers, often organized per memory cell or per processor. However, with these optimizations, different threads may observe the writes of other threads at different points in time. This results in different local views of the threads on the shared memory. *Weak memory models* [8, 95, 115, 157]have been developed as an interface to the programmer that abstracts from architectural details. They specify, without reference to the processor architecture, the possible concurrent executions.

In general, weak memory models allow for behavior of the memory operations that are impossible on a sequentially consistent (SC) memory [111]. Under SC, operations are immediately seen by all threads or, phrased differently, the sequences of memory operations are interleaved. SC matches the developer's intuition about program behavior. As a result of this discrepancy, algorithms that have been developed with SC in mind can have undesirable effects when run on a system with a weak memory. In particular, mutual exclusion algorithms and other programs with data races behave incorrectly if the program order is relaxed only slightly.

```
Procedure p
x = 1;
while y=1 do
    x = 0;
    Sleep(time);
    x = 1;
end
Critical Section;
x = 0;
```

```
Procedure q
y = 1;
while x=1 do
    y = 0;
    Sleep(time);
    y = 1;
end
Critical Section;
y = 0;
```

Figure 1.1: Dekker's mutex algorithm.



(a) Start of execution. **(b)** Writes are buffered.

(c) Initial values are read. **(d)** Writes take effect.

Figure 1.2: Behavior of Dekker under PSO and TSO.

Example To illustrate the problems caused by consistency with weak memory, we examine total store ordering (TSO). It is implemented by x86 and well studied. Under the memory model, each thread has a store buffer of pending writes. A thread can see its own writes before they become visible to other threads (by reading them from its buffer), but once a write hits the memory it becomes visible to all other threads simultaneously: TSO is a multi-copy-atomic memory model [58].

Fig. 1.1 shows a simplified version of Dekker's mutual exclusion algorithm (it does not use a token variable). The algorithm consists of procedures *p* and *q* that are executed by two threads. Both threads claim a resource by setting their variable to 1. If the resource is claimed by the partner, a thread releases the resource and waits for some time until it claims the resource again. If the resource is not claimed by the partner, the thread enters its critical section and releases the resource afterwards.

Intuitively, the protocol guarantees mutual exclusion. It should prevent the threads *p* and *q* from entering their critical sections at the same time. However, if the algorithm is executed on an architecture where the threads buffer their writes, like TSO or partial store ordering (PSO) , it may behave incorrectly (Figure 1.2). Both *p* and *q* issue their writes and put them into their buffers (Fig. 1.2 b). No write was passed to the shared memory yet. Since each thread can only access its own write buffer, both will still see the initial value of the shared memory for the variable of the other thread (Fig. 1.2 c). This means both read 0 and enter the critical section (Fig. 1.2 d). To detect bugs like this, considerable effort has been made to develop verification methods for concurrent programs that take into account weak memory models.

There is a rising demand for verification tools that consider the underlying memory model. Semantics and verification under weak memory models have been the subject of study at least since 2007. Initially, hardware architectures have been addressed [13, 24, 45, 76, 123, 148, 149] First, the behavior of x86 and TSO has been clarified [45, 149], then the POWER architecture has been addressed [148, 123], now ARM is being tackled [76]. The study of weak memory also looks beyond hardware, in particular C11 and C++11 received considerable attention [31, 30, 110]. Recently, an axiomatic memory model for the Linux kernel has been introduced [22]. Research in semantics goes hand in hand with the development of verification methods. They come in two flavors: program logics [165, 163] and algorithmic approaches [46, 28, 63, 19, 24, 41, 64, 1, 5]. These methods and tools are usually designed for a specific memory model.

Since there are many memory models and their number keeps growing, universal verification tools are required. Such a tool would need to take a description of the memory model as input. This means we require a flexible specification language in which all memory models considered so far can be expressed succinctly. A key insight is that, for verification purposes, the semantics is best formulated in an axiomatic style. The memory model is given in terms of constraints that restrict a set of candidate executions. The serial view based memory models developed by Steinke and Nutt [157] presents a uniform way to define many memory models by specifying the possible local views on the memory. Another considerable achievement in this line of research is the flexible specification language `cat` [13, 17, 24], developed by Alglave et al. In `cat`, basically all memory models of interest can be expressed. The `cat` language has been used to define the semantics of assembly for x86/TSO, ARMv7, ARMv8, and POWER but also the semantics of programming languages such as C/C++, including the LINUX kernel concurrency primitives. It describes a memory model as a set of (acyclicity) constraints over relations between memory events. `cat` is made for rapid prototyping. New models are easy to write so that the developer is able to quickly, yet precisely, assess the behavior of the program of interest on the corresponding hardware.

While `cat` is successful as a modeling language, the tool support is lagging behind. Memory model-aware verification tools are still being developed for specific memory models. NIDHUGG [1, 5] implements stateless model checking for TSO, POWER, and a version of ARM. CBMC [19] is a bounded model checker for TSO. The RCMC tool [102] targets the C11 programming language. Other verification problems (e.g. fence insertion to restore sequential consistency) are tackled by MEMORAX [2, 3, 4], OFFENCE [21], FENDER [108], and DFENCE [118]. These tools support TSO and similar models, such as PSO or RMO, but cannot handle POWER or ARM.

What is missing are verification tools that are *modular* in the following sense: Besides the program, they should take a memory model as an input and then perform the analysis relative to that model. The HERD7 tool [24] accompanying `cat` satisfies this requirement. Unfortunately, it is designed for litmus tests and limited to small programs.

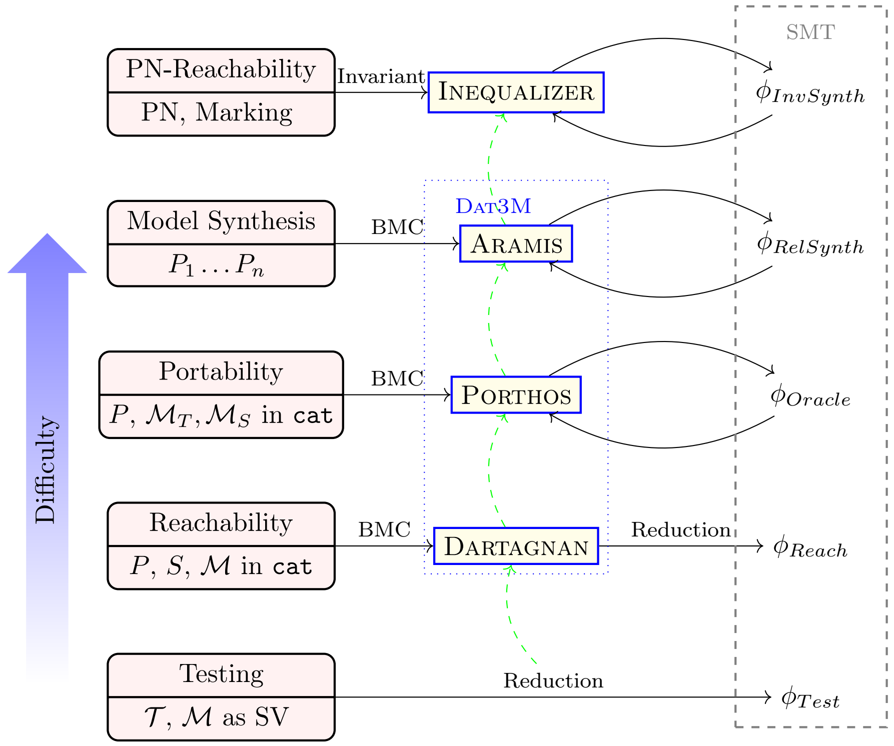


Figure 1.3: An overview of the results.

As shown in Fig. 1.3, this work examines a variety of common problems in the field of concurrent program verification of increasing complexities: testing, program verification, program portability, and memory model synthesis. Starting off by studying the easy testing problem, we keep building on our solution for a current problem in order to solve the following harder problem. In the figure, this is represented by the green dashed arrows.

We give efficient methods to solve these problems. We present our tool suite DAT3M consisting of DARTAGNAN, PORTHOS, and ARAMIS. What sets the suite apart is that it is not developed towards certain architectures. It is universal: The memory model is given **as part of the input**.

First, the thesis examines a prominent approach to program analysis: testing. The testing problem takes a test $\mathcal{T}$ that consists of input sequences of reads and writes on the memory, one for each thread in the concurrent program. The task is to check whether these sequences can form an execution of the entire program that respects the constraints of a given memory model $\mathcal{M}$.

We determine the complexity of the testing problem for most of the known memory models. Moreover, we study the impact on the complexity of parameters like the number of concurrent threads, the length of their executions, and the number of shared variables. In order to show properties for multiple memory models, we use the more structured serial views (SV) here to describe memory models instead of `cat`. For most cases, we determine the lower bound of the complexity by proving that the testing problem is **NP**-hard. What differentiates this from previous related results is a uniform approach that avoids to consider each memory model individually but allows the use of a single reduction on a whole range of memory models. For the weaker memory models, we show that the problem is polynomial. We present polynomial-time testing algorithms that are inspired by determinization methods for automata. We prove that the upper bound of the complexity is in **NP** by describing a uniform reduction of the testing problem to SAT.

Next, the thesis examines the more difficult program reachability problem under a memory model. Solving this problem is essential for program verification against safety properties. Here, we want to know whether a program P can reach a state that satisfies an assertion S under a memory model $\mathcal{M}$. We approach this by adapting our reduction of testing and expanding on it. We introduce DARTAGNAN, a bounded model checker (BMC) for concurrent programs under weak memory models that encodes tasks as SMT-queries. The tool encodes not only the program but also its semantics as defined by the memory model. What makes DARTAGNAN scale are its advanced encoding techniques, most notably the relation analysis and its handling of recursively defined relations. The relation analysis is a novel static analysis technique that significantly reduces the size of the encoding. Based on the constraints imposed by the memory model, it computes approximations of the relation domains, deciding which parts of the relation encoding can be safely omitted. Some `cat` memory models (like Power) contain recursive definitions. We show that we can avoid computing the least fixed point semantics for recursion and thus simplify the encoding.

The program portability problem asks whether a program P can be ported from one memory model $\mathcal{M}_S$ to another $\mathcal{M}_T$ without changing its behavior. We distinguish between two portability notions. A program is trace portable if it has the same set of executions under both models. It is state portable if it has the same set of reachable states. We present PORTHOS, a bounded portability checker. It solves bounded trace portability by expanding on the SMT encoding technique of DARTAGNAN. For this, we cannot avoid computing the least fixed point semantics for recursion. We do so with a compact encoding of a Kleene fixpoint iteration using integer variables. We also add encodings of inconsistency with a memory model. This encoding is a reduction and proves that bounded trace portability is in co-**NP**.

The state portability problem is shown to be Π_2^P-complete and thus even harder than trace portability. Despite this, it is solved efficiently by PORTHOS with a guided search that uses the trace-portability method as a heuristic. This produces fast results for most practical instances.

The next studied problem is memory model synthesis. Formulating an appropriate model for an architecture is a difficult task. It requires a deep understanding of both the architecture itself and axiomatic models in general. We present the tool ARAMIS which synthesizes a memory model using SMT-queries with the encodings of DARTAGNAN and PORTHOS as back-ends. The tools input consists of the behavior of the given architecture described using litmus tests $P_1 \ldots P_n$ and their possible outcomes: a set of litmus tests that succeed under the architecture and a set of tests that do not succeed. Aramis generates a list of constraints over relations and attempts to combine them to a `cat` memory model that behaves correctly for all litmus tests. It supports two methods of generating this list: Enumerating all possible constraints and a guided search. The guided search consists of a counter-example guided inductive synthesis (CEGIS) loop. Here, we use a template relation where we we only specify the required behavior and let the SMT-solver choose an instantiation of the template consistent with the requirements.

Concurrent program verification is generally undecidable. With the BMC Dartagnan, we under-approximate the program behavior. In order to more accurately determine the behavior of a concurrent program, we complement this approach with an over-approximation. We attempt to disprove Petri net reachability through the generation of invariants. We use inductive invariants of Petri nets in the form of linear inequalities in order to examine over-approximations for sequential consistency. Such inequalities function as proofs for the non-reachability or non-coverability of a marking. Applying tools from Discrete Mathematics we prove structural theorems about the space of inductive inequalities and give a close over-approximation. From these insights, we derive linear constraint systems for synthesizing invariant candidates. We also show that deciding whether a candidate is inductive is actually **NP**-complete and give an optimal algorithm that solves the problem. In order to synthesize inductive invariants, we again employ a CEGIS approach that encodes tasks as SMT-queries. It utilizes both, the algorithm and the linear constraints.

We present experimental analyses of the presented tools and compare them with other weak memory verification tools. We find our approach very competitive despite the modularity it offers.

Outline: This work is structured as follows. Chapter 2 contains our formalisms for programs semantics, tests, and memory models. In Chapter 3 we study the testing problem. We cover the reachability problem in Chapter 4, the portability problem in Chapter 5, and the memory model synthesis problem in Chapter 6. We study Petri net invariants in Chapter 7. Finally, we present a conclusion.

Chapter 2

Programs and Memory Models

We introduce our language for concurrent programs and the core of the language `cat`. To define the semantics of a program relative to a memory model in an axiomatic style, we proceed in two steps. We first associate with the program (and independent of the memory model) a set of possible executions that are candidates for the semantics. The memory model consists of a set of constraints that rule out some of those executions. The executions consistent with the constraints form the semantics of the program under the memory model. Our presentation follows [17, 169].

2.1 Programs

Our tool suite DAT3M has the ambition of being widely applicable, from assembly over operating system code written in C/C++ to lock-free data structures. The tool accepts programs in PPC, x86, AArch64 assembly, and a subset of C11, all limited to the subsets supported by HERD7's .litmus format. It also reads our own .pts format which has C11-like syntax with support for C11-atomics. This format contains the key operations we are interested in. We give our language and semantics for shared memory concurrent programs. The syntax of the key parts relevant for this work is given in Fig. 2.1. Programs consist of a finite number of threads from our C11-like language. The operations explicitly read from the shared memory into local registers, write from registers into memory, and support local computations on the registers. The shared memory consists of a heap with spaces allocated before the execution.

We refer to global variables as memory locations and to local variables as registers. We support pointers, meaning a register may hold the address of a location. Addresses and values are integers, and we allow for pointer arithmetic. Different synchronization mechanisms are available, including variants of read-modify-write, and various fences.

$$\begin{aligned}
\langle \mathit{prog} \rangle &::= \texttt{program}\ \langle \mathit{thrd} \rangle^{*} \\
\langle \mathit{thrd} \rangle &::= \texttt{thread}\ \langle \mathit{tid} \rangle\ \langle \mathit{inst} \rangle^{+} \\
\langle \mathit{inst} \rangle &::= \langle \mathit{reg} \rangle \leftarrow \langle \mathit{exp} \rangle \mid \langle \mathit{inst} \rangle ; \langle \mathit{inst} \rangle \\
&\quad \mid \langle \mathit{reg} \rangle = \texttt{load}(\langle \mathit{loc} \rangle, \langle \mathit{atom} \rangle) \\
&\quad \mid \langle \mathit{loc} \rangle = \texttt{store}(\langle \mathit{reg} \rangle, \langle \mathit{atom} \rangle) \\
&\quad \mid \texttt{while}\ \langle \mathit{pred} \rangle\ \langle \mathit{inst} \rangle \\
&\quad \mid \texttt{fence}\ \langle \mathit{fence} \rangle \\
&\quad \mid \texttt{if}\ \langle \mathit{pred} \rangle\ \texttt{then}\ \langle \mathit{inst} \rangle\ \texttt{else}\ \langle \mathit{inst} \rangle \\
\langle \mathit{atom} \rangle &::= \texttt{sc} \mid \texttt{rel} \mid \texttt{acq} \mid \texttt{con} \mid \texttt{rx}
\end{aligned}$$

Figure 2.1: Programming language.

We model the heap with a memory location for each variable and a set of locations for each memory allocation of an array. Every location x has an address $ad(x)$, which is an integer variable and its value is chosen at runtime. In an array, the locations are required to have consecutive addresses. The same arithmetic operations are allowed for memory addresses as for regular integer values.

Store and load operations perform writes to and reads from the heap. A load contains the address *loc* it reads from and the register *reg* in which it stores the value. A store has two expressions as parameters, the value *reg* it writes and the address *loc* it writes to. Loads and stores have a memory-order tag *atom* as a parameter, which defines additional ordering guarantees for these events (needed for e.g., modeling C11 low level atomics), used by the memory model. The `sc` tag guarantees a sequentially consistent semantics for the access; `rel`/`acq` and `rel`/`con` implement the message-passing idiom; the `rx` (relaxed) tag maps directly to hardware accesses giving minimal guarantees on how those accesses are performed. Weaker guarantees yield higher performance but they usually allow for additional program behavior that is hard to predict.

Local events represent thread-local assignments. Conditionals in the program are encoded as if-instructions. Fences instructions have a type and an optional subtype as different architectures provide specific fence instructions with their corresponding semantics. There are various fence instructions (e.g. sync, lwsync, and isync on Power and mfence on x86) that enforce ordering and visibility constraints among instructions. We refrain from explicitly defining the fence types or the expressions and predicates used in assignments and conditionals.

In addition, our tools support different variants of atomic read-modify-write instructions, such as RMW (atomic read-modify-writes) [105], including conditional RMWs such as compare-exchange. A RMW reads the value in location and then updates it (the new value may depend on the old value), all in one atomic event. We also support the RCU (read-copy-update) [22, 126] synchronization operations needed by the Linux kernel. An RCU is a locking

C11	x86	POWER	ARMv7
Load `rx`	`MOV`	lwz	ldr
Load `con`	`MOV`	lwz; lwsync	ldr; dmb ish
Load `acq`	`MOV`	lwz; lwsync	ldr; dmb ish
Load `sc`	`MOV`	sync; lwz; lwsync	ldr; dmb ish
Store `rx`	`MOV`	stw	str
Store `rel`	`MOV`	lwsync; stw	dmb ish; str
Store `sc`	`MOV; mfence`	sync; stw	dmb ish; str; dmb ish

Table 2.1: Compiler mappings for x86, POWER and ARMv7.

mechanism that supports concurrency between a single updater and multiple readers in a scalable manner. We give a detailed description of these instructions in Section A.1 of the appendix.

A program is compiled and then executed at the assembly level. The program is converted to hardware specific assembly code according to a given compiler mapping. The compiler mapping replaces load and store operations with their corresponding assembly memory accesses and adds fences to enforce the ordering guarantees provided by the memory model tag. Each compiler uses its own mapping. We use the mappings given in Table 2.1, which are the ones used by the LLVM 4.0 compiler [128]. Other mappings, like the one from [48], can be easily added. Our only requirement is that the mapping of each atomic operation contains a single memory access.

It is worth noting that we assume the compiler does not perform any optimization; the program to be verified has already been optimized. Compiler optimizations under weak memory models are an active topic of research [110, 124, 164, 169], but they are out of the scope of this work.

2.1.1 Assertions

We use the assertion language of Herd7 [20]. Assertions are reachability conditions. They define conditions on the variables that should hold in the final state. The syntax is as follows:

$$
\begin{aligned}
\langle \mathit{assert} \rangle &::= \langle \mathit{quant} \rangle \langle \mathit{form} \rangle \mid \ \mathit{filter}\ \langle \mathit{form} \rangle \langle \mathit{quant} \rangle \langle \mathit{form} \rangle\\
\langle \mathit{quant} \rangle &::= \mathit{exists} \mid \mathit{not\text{-}exists} \mid \mathit{forall}\\
\langle \mathit{form} \rangle &::= \langle \mathit{intval} \rangle \langle \mathit{op} \rangle \langle \mathit{intval} \rangle \mid \neg \langle \mathit{form} \rangle \mid \langle \mathit{form} \rangle \wedge \langle \mathit{form} \rangle\\
\langle \mathit{op} \rangle &::= \ = \mid \neq \mid \geq \mid \leq \mid > \mid <\\
\langle \mathit{intval} \rangle &::= \langle \mathit{location} \rangle \mid \langle \mathit{register} \rangle \mid \langle \mathit{int} \rangle.
\end{aligned}
$$

An assertion contains a Boolean formula defining inequalities over the values of registers and locations. The assertion has a quantifier over the reachable states that should satisfy the formula: there is one, there is none, or all reachable states should satisfy it. The *filter* expression restricts the set of states that should be checked. It is syntactic sugar and can be expressed in the main formula. We included it to retain compatibility with HERD7.

Assertion can also be used freely throughout the code as assert instructions, rather than being limited to the end of the execution. Semantically, these assertions do not stop the execution but record the failure and continue. Each instructions `assert(exp)` loads the values of `exp` into fresh registers and adds the corresponding condition to the assertions on the final state.

2.2 Executions.

The semantics of a program is given in terms of *executions*, partial orders between memory events where the events represent occurrences of the instructions and the ordering edges represent dependencies. An execution $X = (\mathbb{E}, rf, co)$ consists of memory events executed by the program of interest as well as two relations between those executed events [13, 169]. The set $\mathbb{E}$ states which events have been executed in each thread. This forms the control flow of the program. The reads-from relation *rf* specifies from which store each load gets its value. The coherence order *co* is the order in which stores to a variable take effect in the memory. This information is sufficient to describe an execution unambiguously. Further induced relations can be obtained from the source code of the program.

The axioms in Fig. 2.2 describe necessary properties for an execution. They ensure that an execution represents a valid path through the control flow and is consistent with the data-flow semantics. Note, that only executed events may be related. In this section, any set of events we define is restricted to executed events. We denote the memory events as $\mathbb{M}$, reads as $\mathbb{R}$, and writes as $\mathbb{W}$: $\mathbb{M} := \mathbb{R} \cup \mathbb{W}$. By $\mathbb{R}_l$ and $\mathbb{W}_l$ we refer respectively to the reads and writes that access location l. The events of thread t form the set $\mathbb{E}_t$. Axiom ① contains the *path* predicate which we do not make explicit. It requires the executed events $\mathbb{E}$ to form a valid path in the threads' control flow. By Axioms ② and ③, the *reads-from relation rf* gives for each read a unique write to the same location from which the read obtains its value. Here, $r_1; r_2 := \{(x, y) \mid \exists z : (x, z) \in r_1 \text{ and } (z, y) \in r_2\}$ is the composition of the relations r_1 and r_2. We write $r^{-1} := \{(y, x) \mid (x, y) \in r\}$ for the inverse of relation r. Events accessing the same location in the memory are related by *loc*. Predicate $total(r, A)$ holds if r is a total order on the event set A. Finally, $id(A)$ is the identity relation on the set A. By Axioms ④ and ⑤, the *coherence relation co* relates different writes to the same location (*loc*), and it forms a total order for each location. We will assume the existence of an initial write event for each location which assigns value 0. This event is first in the coherence order.

Fig. 2.3 presents further relations that describe program and execution semantics. These relations are induced by the execution and the program and they are required to satisfy the given axioms. Relations *int* and *loc* are equivalences relating events belonging to the same thread ⑥ and accessing the

$\mathbb{E}$	executed events	rf, co	$\subseteq \mathbb{E} \times \mathbb{E}$	reads-from, coherence order

① $path$ ② $rf \subseteq (\mathbb{W} \times \mathbb{R}) \cap loc$ ③ $rf; rf^{-1} = id(\mathbb{W})$
④ $co \subseteq ((\mathbb{W} \times \mathbb{W}) \cap loc) \setminus id(\mathbb{E})$ ⑤ $total(co, \mathbb{W}_l)$

Figure 2.2: Executions; adapted from [169].

$loc,\ int, ext$	$\subseteq$	$\mathbb{E} \times \mathbb{E}$	same location, thread internal/external
$po, addr, data, ctrl$	$\subseteq$	$\mathbb{E} \times \mathbb{E}$	program order, address/data/control dependency
$fence(f)$	$\subseteq$	$\mathbb{E} \times \mathbb{E}$	fences of type f

⑥ $equiv(int, \mathbb{E})$ ⑦ $equiv(loc, \mathbb{M})$ ⑧ $po \subseteq int$ ⑨ $total(po, \mathbb{E}_t)$
⑩ $addr \subseteq (\mathbb{R} \times \mathbb{M}) \cap po$ ⑪ $data \subseteq (\mathbb{R} \times \mathbb{W}) \cap po$ ⑫ $ctrl \subseteq (\mathbb{R} \times \mathbb{E}) \cap po$
⑬ $fence(f) \subseteq po$ ⑭ $ext \subseteq \mathbb{E} \times \mathbb{E} \setminus int$

Figure 2.3: Induced relations.

same location ⑦. Relations $po, addr, data$ and $ctrl$ represent program order and address/data/control dependencies. Axiom ⑧ states that the *program order po* is an intra-thread relation which ⑨ forms a total order when projected to the executed events in the same thread (predicate $total(r, A)$ holds if r is a total order on the set A).

A read r is related to a memory event e by the *address dependency* $data(r, e)$ if the address accessed by e depends on the value read by r. Similary, a read r is related to a write w by the *data dependency* $data(r, w)$ if the value written by w depends on the value read by r. The *control dependency* relates a read to a branching event where the value of the branching condition depends on the read. *Address dependencies* are either read-to-read or read-to-write ⑩, *data dependencies* are read-to-write ⑪, and *control dependencies* originate from reads ⑫. Fence relations are architecture specific and relate only events in program order ⑬.

In our setting, fence instructions generate relations between events. For each type of fence we introduce a relation between events separated by the fence in the program order. For each type of fence instruction f, we define the corresponding set of fence events $\mathbb{F}_f$. Each instruction is associated with its corresponding relation: $fence(f) := po; id(\mathbb{F}_f); po$. The exception is the *isync* fence instruction used by the Power architecture (see Fig. 2.10): $fence(cd\text{-}isync) := ctrl; id(\mathbb{F}_{isync}); po$.

A *state* is a function that assigns a value to each location and register. A state $state(X)$ *computed* by a given execution X is defined as follows. The value of a location is given by the last write on that location according to the co relation. The value of a register depends on the last executed event (according to the control flow) writing to the register.

$$\begin{tikzcd}
Iy0 \arrow[rr, bend left, "co"] \arrow[dr, bend right, "rf"'] & Wx1 \arrow[d, "po"'] & Wy1 \arrow[d, "po"] & Ix0 \arrow[ll, bend right, "co"'] \arrow[dl, bend left, "rf"] \\
 & Ry0 & Rx0 &
\end{tikzcd}$$

Figure 2.4: An execution of Dekker's Mutex algorithm

Example We revisit our running example with the execution presented in an operational manner in Fig. 1.2. We give a graph illustrating the same execution in Fig. 2.4. At the beginning of the execution, both x and y have the value 0 which is expressed by the initial writes to the memory $Iy0$ and $Ix0$. Procedure p executes write event $Wx1$ on the memory followed by read event $Ry0$. Procedure q executes $Wy1$ followed by $Rx0$. Obviously the two writes of the program occur after the initial writes in the coherence order. The reads access the initial values, so the initial writes are related with the reads on the corresponding variables in rf. Note that the program order is not part of the execution, it is induced by the relation from the set of executed events and the control flow of the program. We usually add it to the execution graphs (or traces) for readability.

2.3 Memory Models

Informally, a memory model defines when store operations executed by one thread become visible to other threads. This means a memory model determines the semantics of a program on a hardware architecture. The semantics is defined in terms of executions. Memory models define a consistency predicate on executions. The semantics of a program on that memory model is then given by the executions of the program that satisfy the predicate [13, 19, 123]. We introduce memory models and consistency.

We use the language `cat` [13, 17, 24] to define memory models, the core of which is shown in Fig. 2.5. There are functional programming features in `cat` that we do not support since they are not needed to define the hardware architectures of interest. Together with its accompanying tool Herd7 [20], `cat` has been used not only to formalize the semantics of assembly for x86/TSO, POWER, ARMv7, and ARMv8, but also programming languages like C/C++, transactional memory extensions, and recently the Linux kernel concurrency primitives [13, 22, 24, 30, 54, 110, 141].

In `cat`, memory models define relations over the events in executions. If two events e_1 and e_2 are related by r, then $\mathsf{r}(e_1, e_2)$ holds. The base relations $\langle b \rangle$ contain rf and co which define the execution (see Section 2.2) and other relations induced by an execution together with the program semantics. They are common to all memory models and include the address (addr), data and control (ctrl) dependencies, thread internal (int) and external (ext) communication, the read-modify-write relation (rmw), the relation between events on the same location

$$
\begin{aligned}
\langle MCM\rangle &::= \langle assert\rangle \mid \langle rel\rangle \mid \langle MCM\rangle \wedge \langle MCM\rangle \\
\langle assert\rangle &::= acyclic(\langle r\rangle) \mid irreflexive(\langle r\rangle) \mid empty(\langle r\rangle) \\
\langle r\rangle &::= \langle b\rangle \mid \langle name\rangle \mid \langle r\rangle \cup \langle r\rangle \mid \langle r\rangle \cap \langle r\rangle \mid \langle r\rangle \setminus \langle r\rangle \\
&\quad \mid \langle r\rangle^{-1} \mid \langle r\rangle^{+} \mid \langle r\rangle^{*} \mid \langle r\rangle ; \langle r\rangle \mid \langle set\rangle \times \langle set\rangle \\
\langle b\rangle &::= \mathsf{id} \mid \mathsf{int} \mid \mathsf{ext} \mid \mathsf{po} \mid \mathsf{fence}(\langle f\rangle) \\
&\quad \mid \mathsf{rmw} \mid \mathsf{ctrl} \mid \mathsf{data} \mid \mathsf{addr} \mid \mathsf{loc} \mid \mathsf{rf} \mid \mathsf{co}. \\
\langle f\rangle &::= sync \mid lwsync \mid isync \mid fence \\
\langle set\rangle &::= \mathbb{E} \mid \mathbb{M} \mid \mathbb{F} \mid \mathbb{W} \mid \mathbb{R} \mid \mathbb{A}
\end{aligned}
$$

Figure 2.5: Core of CAT [17].

(loc), and the different types of fence relations (fencerel($\langle f\rangle$)). We distinguish between the following three groups of base relations.

Control flow: id, int, ext, po, fencerel($\langle f\rangle$), ctrl, and rmw.

Thread internal assignment: data and addr.

Memory: loc, rf, and co.

For control flow relations, $\mathsf{r}(e_1, e_2)$ is fully determined by two parameters: positions of the events in the control flow and whether the branch containing the events is taken in a particular execution. Thread internal assignments additionally require that no branch between e_1 and e_2 with events that can overwrite the value of register r written by e_1 is taken. Memory relations depend on the order of memory accesses in a particular execution. The relation loc is execution-dependent since we allow for pointer arithmetic and thus the address on a memory event can only be resolved at runtime. Events are instances of instructions that may access the shared memory such as reads $\mathbb{R}$, writes $\mathbb{W}$ (including initial writes), memory events $\mathbb{M}$ (the union of both), fences $\mathbb{F}$, and events $\mathbb{A}$ that are part of an atomic construction such as RMW. Events of each thread are executed sequentially according to the program order.

So-called derived relations are defined using operations on relations such as transitive closure, union, intersection, and composition. These derived relations are built from the base relations and named relations which we will address next. The operators are the standard set-theoretic ones together with inverse ($^{-1}$), transitive closure ($^{+}$), transitive and reflexive closure (*), as well as relational composition (;).

The `cat` language also supports recursive definitions of relations. We assume a set $\langle name\rangle$ of relation names (different from the predefined relations) and require that each name used in the memory model has a corresponding defining equation $\langle name\rangle := \langle r\rangle$. The relation $\langle r\rangle$ may again contain relation names, *Consistent*$_{SC}$

$\textcircled{15}$ $acyclic(po \cup rf \cup fr \cup co)$

Figure 2.6: SC using the from-read relation $fr := rf^{-1}; co$.

making the system of defining equations recursive. Importantly, `cat` allows to define derived relations as least solutions to a system of equations. The semantics of such recursive definitions is well defined only if they behave monotonously [17]. This means that if one relation becomes larger, every other relation can either remain unchanged or become larger, never smaller. Mostly, the `cat` operators are already monotonous, the only non-monotonous construct is the right hand side of the difference operator "$\smallsetminus$". We forbid recursive definitions in the right side of it to enforce monotonicity in a syntactic manner.

To define the notion of consistency for executions, a memory model requires a number of constraints over its relations to hold. There are acyclicity, irreflexivity, and emptiness constraints over derived relations.

An execution is consistent with a model if it satisfies all constraints: Given a program P and a memory model $\mathcal{M}$, an execution X of P is *consistent with* $\mathcal{M}$ if the derived relations of $\mathcal{M}$ constructed from the base relations of X satisfy the constraints of $\mathcal{M}$. We denote the set of consistent executions by $cons_{\mathcal{M}}(P)$.

We present the `cat` definition of sequential consistency (SC) in Fig. 2.6. Here, we define the named from-read relation $fr := rf^{-1}; co$ which relates a read event to the writes that occur after the one it reads from in the coherence order. The memory model consists of only one acyclicity constraint on the union of rf, fr, po, and co.

Intuitively, an execution is sequentially consistent if there is an interleaving of the memory events of the threads. This interleaving respects the relations rf and co and thus a write is followed by the events that read from it. Since the interleaving also respects fr, there is no write on the same variable between a write and a read related by rf. It is easy to see that an interleaving exists if and only if the acyclicity constraint is satisfied.

Example We again revisit our running example (given in Fig. 1.2 and Fig. 2.4). Fig. 2.7 shows, that the execution is not sequentially consistent since it contains a cycle (given by the red edges) that violates the acyclicity constraint.

The `cat` formalization of TSO, the memory model describing the x86 architecture, is given in Fig. 2.8. We define the named relation $rfe := rf \smallsetminus int$, which restricts relation rf to external communication. We do the same with co. We denote the external restriction of any relation with the suffix "e" and the internal restriction with "i". In TSO, a read may access the memory even before a write on a different location that is earlier in po but has been delayed by the write

$$\begin{tikzcd}
Iy0 \arrow[rr, bend left, "co"] \arrow[dr, bend right, "rf"'] & Wx1 \arrow[d, "po"'] & Wy1 \arrow[d, "po"] & Ix0 \arrow[ll, bend right, "co"'] \arrow[dl, bend left, "rf"] \\
 & Ry0 \arrow[ur, "fr"'] & Rx0 \arrow[ul, "fr"'] &
\end{tikzcd}$$

Figure 2.7: An execution of Dekker, inconsistent with SC.

Consistent$_{TSO}$

rfe := rf ∖ int
coe := co ∖ int
ghb-tso := po-tso ∪ com-tso ∪ implied
com := co ∪ fr ∪ rf
com-tso := co∪fr∪rfe
po-tso := (po ∖ $\mathbb{W} \times \mathbb{R}$) ∪ fence(mfence)
implied := po ∩ ($\mathbb{W} \times \mathbb{R}$) ∩ (($\mathbb{M} \times \mathbb{A}$) ∪ ($\mathbb{A} \times \mathbb{M}$))
po-loc := po ∩ loc

⑯ *acyclic*(po-loc ∪ com) ⑰ *acyclic*(ghb-tso) ⑱ *empty*(rmw ∩ (fre; coe))

Figure 2.8: TSO.

buffer of the thread (see Fig. 1.2). In this sense, the program order is relaxed. Events on the same variable still arrive in the memory in program order, this is ensured by Constraint ⑯. Events can only arrive out of order if the first event is a write and the second a read, otherwise they must behave sequentially consistent. This is represented by Constraint ⑰, where we exclude write-read pairs ($\mathbb{W} \times \mathbb{R}$) from the program order. Relation implied ensures that we do not exclude write-read pairs that are part of an atomic construct. Constraint ⑱ enforces atomicity of read-modify-write events by invalidating executions with an external write appearing in between the atomic pair of events and writing to the same location.

Example Fig. 2.9 shows, that the execution is consistent with TSO. Here, the cycle that violated the SC constraint is no longer an issue. Both constraints of TSO are similar to SC, however in Constraint ⑯, *po* is restricted to *loc*. The pairs $(Wx1, Ry0), (Wy1, Rx0) \in po$ (denoted by grey edges) operate on different variables and thus $(Wx1, Ry0), (Wy1, Rx0) \notin po \cap loc$. Furthermore $(Wx1, Ry0), (Wy1, Rx0) \notin po \setminus (\mathbb{W} \times \mathbb{R})$ and there is no cycle in the execution that does not contain those two pairs. The execution is consistent with TSO.

We present the Power memory model in Fig. 2.10. It is not multi-copy-atomic, meaning once a write hits the memory it does not necessarily become visible to all other threads simultaneously. It is a rather complex model with four constraints and many named relations. Note that the power model makes use of recursively defined relations. For instance, $ii := ii_0 \cup ci \cup (ic; ci) \cup (ii; ii)$ contains a direct recursion since ii itself is used in the definition . It also contains more elaborate recursions, it depends on relation ci and ic, which again use ii.

$$\begin{tikzcd}
Iy0 \arrow[rr, bend left, "co"] \arrow[dr, bend right, "rf"'] & Wx1 \arrow[d, "po"'] & Wy1 \arrow[d, "po"] & Ix0 \arrow[ll, bend right, "co"'] \arrow[dl, bend left, "rf"] \\
 & Ry0 \arrow[ur, "fr"'] & Rx0 \arrow[ul, "fr"'] &
\end{tikzcd}$$

Figure 2.9: An execution of Dekker consistent with TSO.

Consistent$_{Power}$

$\textcircled{16}$ $acyclic((po \cap loc) \cup rf \cup fr \cup co)$ $\textcircled{19}$ $acyclic(hb)$

$\textcircled{20}$ $irreflexive(fre; prop; hb^*)$ $\textcircled{21}$ $acyclic(co \cup prop)$

Preserved Program Order

$dp := addr \cup data \quad rdw := (po \cap loc) \cap (fre; rfe) \quad detour := (po \cap loc) \cap (coe; rfe)$

$ii_0 := dp \cup rdw \cup rfi \quad ci_0 := fence(cd\text{-}isync) \cup detour$

$ic_0 := \varnothing \quad cc_0 := dp \cup (po \cap loc) \cup ctrl \cup (addr; po)$

$ii := ii_0 \cup ci \cup (ic; ci) \cup (ii; ii) \quad ci := ci_0 \cup (ci; ii) \cup (cc; ci)$

$ic := ic_0 \cup ii \cup cc \cup (ic; cc) \cup (ii; ic) \quad cc := cc_0 \cup ci \cup (ci; ic) \cup (cc; cc)$

$ppo := ((\mathbb{R} \times \mathbb{R}) \cap ii) \cup ((\mathbb{R} \times \mathbb{W}) \cap ic)$

Fences

$fence := \mathsf{fence}(sync) \cup (\mathsf{fence}(lwsync) \setminus (\mathbb{W} \times \mathbb{R}))$

Thin Air

$hb := ppo \cup \mathsf{fence}(fence) \cup rfe$

Propagation

$prop\text{-}base := (fence \cup (rfe; fence)); hb^*$

$prop := ((\mathbb{W} \times \mathbb{W}) \cap prop\text{-}base) \cup (com^*; prop\text{-}base^*; \mathsf{fence}(sync); hb^*)$

Figure 2.10: Power [24].

2.4 Tests

The CAT language provides a powerful and versatile way to define memory models. However, in order to show common properties of memory models, a more restrictive and thus more uniform description is useful. This is why we use serial views in our study of the testing problem. In the following, we describe tests and their executions and present a definition of memory models based on serial views according to [157].

A test consists of a finite set of threads that operate on a shared memory. Each thread is given as a sequence of read/write events that, intuitively, correspond to the local view of the thread on the shared memory. Here, both the memory location and the value of a read/write events are fully determined. A test for the example in Figure 1.1 is

$$(w, x, 1).(r, y, \bot) \parallel (w, y, 1).(r, x, \bot).$$

Formally, an event e is an element in $\mathcal{EV} := (\mathcal{C} \times \mathcal{L} \times \mathcal{D}) \times (\mathcal{ID} \times \mathbb{N})$. The event executes a write or read event from $\mathcal{C} := \{w, r\}$ on a location from the set $\mathcal{L}$,

$$\begin{tikzcd}
(w,y,\bot,\varepsilon,0) \arrow[dr, bend right, "rf"'] & (w,x,1,p,0) \arrow[d, "po"'] & (w,y,1,q,0) \arrow[d, "po"] & (w,x,\bot,\varepsilon,1) \arrow[dl, bend left, "rf"] \\
& (r,y,\bot,p,1) & (r,x,\bot,q,1) &
\end{tikzcd}$$

Figure 2.11: The execution and program order of test $\mathcal{T}_{Dekker}$.

assuming a fixed value from some data domain $\mathcal{D}$ that contains the initial value $\bot$. Each event carries a thread identifier from the set $\mathcal{ID}$ which has a distinguished element ε for the initialization thread. Moreover, an event has an issue index, a natural number that determines the order in which events are issued by a thread. Given an event $e = (c, x, v, p, i) \in \mathcal{EV}$, we use $cmd(e) = c$, $loc(e) = x$, $val(e) = v$, $thrd(e) = p$, and $idx(e) = i$ to access the command, the variable, the value, the thread identifier, and the issue index. Given a set of events $\mathcal{T} \subseteq \mathcal{EV}$, we denote a subset that share certain properties using wildcard $*$. For example, the set of events writing variable x, $\{e \in \mathcal{T} \mid cmd(e) = w \text{ and } loc(e) = x\}$, is denoted by $(w, x, *, *, *)_{\mathcal{T}}$. By slight abuse of notation, we use w to denote a write event and similarly r for a read event. To establish the initial value 0 for all variables, we introduce write events $(w, x, \bot, \varepsilon, i)$ that belong to the initialization thread ε.

Definition 1. *A* test *is a finite set of events $\mathcal{T} \subseteq \mathcal{EV}$ with $|(*, *, *, p, i)_{\mathcal{T}}| \leq 1$ for all $p \in \mathcal{ID}$ and $i \in \mathbb{N}$. Moreover, if $|(*, x, *, *, *)_{\mathcal{T}}| > 0$ then $|(w, x, \bot, \varepsilon, *)_{\mathcal{T}}| = 1$.*

We formally define $\mathcal{T}_{Dekker}$, the test for the program in Fig. 1.1 given above:

$$\begin{gathered}(w, x, 1, p, 0).(r, y, \bot, p, 1) \parallel (w, y, 1, q, 0).(r, x, \bot, q, 1) \\ \parallel (w, y, \bot, \varepsilon, 0).(w, x, \bot, \varepsilon, 1).\end{gathered}$$

The test consists of two threads with identifiers p and q, and the initial thread ε. It checks for a violation of the mutex property and therefore ignores the while loop. More precisely, the test represents the path where both threads p and q write 1 to their variable but read the initial value $\bot$ from the variable of the partner, and hence both enter the critical section.

For notational convenience, and like in the example above, we present tests in terms of their threads, and threads as sequences of events. In this case, we may omit both the thread identifier and the issue index. We further assume that the initial thread ε only writes $\bot$ to each used variable. If we fix an ordering on the variables, this fully defines the initial thread and we can omit it as well. With these conventions, we arrive at the previous description of $\mathcal{T}_{Dekker} : (w, x, 1).(r, y, \bot) \parallel (w, y, 1).(r, x, \bot)$.

The semantics of tests is defined in terms of executions. An *execution of a test* consists of the reads-from relation rf.

Note that this differs from the semantics of programs in two key points. While the execution of a program describes the control flow, the control flow of a test is already known. An execution of a test does not give the coherence relation co, it is left implicit.

2.5 Serial Views

Memory models restrict the set of executions to so-called consistent ones. In the Steinke-Nutt framework [157], the executions of a test consistent with a memory model are defined in terms of serial views. Roughly, a serial view of an execution is the total order in which the events become visible to a thread. This total order has to be compatible with the execution: a read receives its value from the most recent write to the variable, where most recent refers to the serial view. A thread may, however, not see all the events of other threads. To model this, the definition of serial views takes a subset of events as a parameter. Given an execution $rf \subseteq \mathcal{T} \times \mathcal{T}$, we call a subset $\mathcal{E} \subseteq \mathcal{T}$ *source-closed* if for all $r \in \mathcal{E}$ and $w \in \mathcal{T}$ with $w \overset{rf}{\rightarrow} r$, we have $w \in \mathcal{E}$.

Definition 2 (Serial View)**.** *Consider an execution $rf \subseteq \mathcal{T} \times \mathcal{T}$, a source-closed set $\mathcal{E} \subseteq \mathcal{T}$, and a strict partial order $< \subseteq \mathcal{E} \times \mathcal{E}$. A strict total order $<_{sv} \subseteq \mathcal{E} \times \mathcal{E}$ satisfies $<_{sv} \in SerialView(rf, \mathcal{E}, <)$ if the following holds:*

(i) It refines $<$, which means $< \subseteq <_{sv}$.

(ii) For all pairs $w \overset{rf}{\rightarrow} r$ with $w, r \in \mathcal{E}$ we have $w <_{sv} r$. Moreover, there is no $w' \in \mathcal{E}$ so that $w <_{sv} w' <_{sv} r$ and $loc(w) = loc(w')$.

We say $<_{sv}$ is a serial view of $\mathcal{E}$ in rf that respects $<$.

Recall that *strict* partial and total orders are asymmetric and transitive. To give an example of a memory model definition in the framework of Steinke and Nutt, we formalize sequential consistency (SC). It ensures that every thread observes all events in the order in which they were issued. Using Definition 2, an execution is valid under SC if there is a serial view of all events that respects the program order. This means a strict total order $<_{sv}$ exists such that $<_{sv} \in SerialView(rf, \mathcal{T}, po)$. We write $\exists <_{sv} \in SerialView(rf, \mathcal{T}, po)$. The program order $po \subseteq \mathcal{T} \times \mathcal{T}$ relates events within the same thread according to their issue index, and after all initial writes.

Definition 3 (SC)**.** *An execution $rf \subseteq \mathcal{T} \times \mathcal{T}$ is* consistent with SC *if*

$$\exists <_{sv} \in SerialView(rf, \mathcal{T}, po) .$$

The example test $\mathcal{T}_{Dekker}$ admits only one execution, depicted in Fig. 2.11, where the reads observe the writes of the initial thread ε. In any serial view for SC, at least one initial write has to be overwritten before the corresponding read can be processed. The formalism captures this as follows. Assume there was a strict total order $<_{sv}$ that satisfies the requirements of Definitions 2 and 3. Since the serial view respects the program order po, the maximal event in $<_{sv}$ is one of the reads, say $(r, y, \bot, p, 1)$. The write $(w, y, 1, q, 0)$ is larger than $(w, y, \bot, \varepsilon, 0)$ in $<_{sv}$ because the latter is an initial event:

$$(w, y, \bot, \varepsilon, 0) <_{sv} (w, y, 1, q, 0) <_{sv} (r, y, \bot, p, 1).$$

Since $(w, y, \bot, \varepsilon, 0)\mathit{rf}(r, y, \bot, p, 1)$, we obtain a contradiction to Definition 2(ii). The argumentation is similar if the read on x is maximal, and indeed there is no strict total order that forms a serial view for SC. This behavior changes if we examine the test under a weaker model.

Hutto and Ahamad [97] developed the SLOW model to solve the exclusion and dictionary problems with minimal consistency maintenance. The model requires that the threads observe all writes to the same variable in program order. Furthermore, local writes must be visible immediately. In the Steinke-Nutt framework, SLOW is formalized with a different serial view for every thread and every variable. The view contains all write events on this variable and all events of this thread on the variable, and respects the program order. We have [157, Theorem 3.7]:

Definition 4. *An execution* $\mathit{rf} \subseteq \mathcal{T} \times \mathcal{T}$ *is* consistent with SLOW *if*

$$\forall p \in \mathcal{ID}\ \forall x \in \mathcal{L}\ \exists <_{sv} \in\ \mathit{SerialView}\,(\mathit{rf}, (*, x, *, p, *)_{\mathcal{T}} \cup (w, x, *, *, *)_{\mathcal{T}}, \mathit{po})\,.$$

Since writes on different variables can be observed out of order, the execution in Figure 2.11 is valid under SLOW. We prove this by constructing the required serial views. Only the following two are of interest since they contain a read action. We give them as sequences:

$$\begin{aligned} <_{sv} \text{ of } q, x: &\quad (w, x, \bot, \varepsilon, 1).(r, x, \bot, q, 1).(w, x, 1, p, 0) \\ <_{sv} \text{ of } p, y: &\quad (w, y, \bot, \varepsilon, 0).(r, y, \bot, p, 1).(w, y, 1, q, 0). \end{aligned}$$

In the following chapter, we will utilize the presented formalisms for tests and serial views in a study of the testing problem.

Chapter 3

Testing

A core problem behind many verification methods is the so-called *testing problem under weak memory models*. The testing problem, as it has first been studied by Gibbons and Korach [89], is defined as follows.

> **Problem:** Given a test $\mathcal{T} \subseteq \mathcal{EV}$, is there an execution $rf \subseteq \mathcal{T} \times \mathcal{T}$ that is consistent with memory model $\mathcal{M}$?

We say a test *succeeds under* $\mathcal{M}$ if there is an execution consistent with $\mathcal{M}$, otherwise it *fails under* $\mathcal{M}$. In the example, $\mathcal{T}_{Dekker}$ fails under SC but succeeds under TSO. We use the term *testing algorithms* to refer to algorithms that solve the testing problem. Note that our notion of testing checks the consistency of a concurrent execution wrt. a weak memory model. It does not exercise a program on a set of inputs. We consider the testing problem $\textsc{Test}(\mathcal{M})$ for every memory model $\mathcal{M}$ shown in Figure 3.1. We also consider restricted variants of the testing problem that admit more efficient algorithms.

The testing problem has various applications in program analysis. Testing algorithms are used as subroutines in over-approximate (may) program analyses. Assume the over-approximation results in a counterexample to a correctness statement. To check whether the counterexample corresponds to an actual execution, we extract for each thread the sequence of events on the memory and understand the counterexample as a test. If the test succeeds, the counterexample is genuine. Otherwise, the failing test suggests a refinement of the may analysis. This leads to a CEGAR-like verification loop [56]. As an under-approximation, testing is used during debugging. We check reachability of an undesirable state for each thread and then solve the testing problem on the collected sequences of events. Further applications are synchronization inference algorithms [3, 21, 41, 118, 18]. Their task is to determine the placement of synchronization primitives within a program. A final application is the estimation of best and worst case execution times. Here, testing algorithms can rule out infeasible paths to improve the analysis.

The memory models are classified in [10, 129, 157]. A memory model $\mathcal{M}_w$ is called *weaker than* another memory model $\mathcal{M}_s$, denoted by $\mathcal{M}_s \leq \mathcal{M}_w$ and

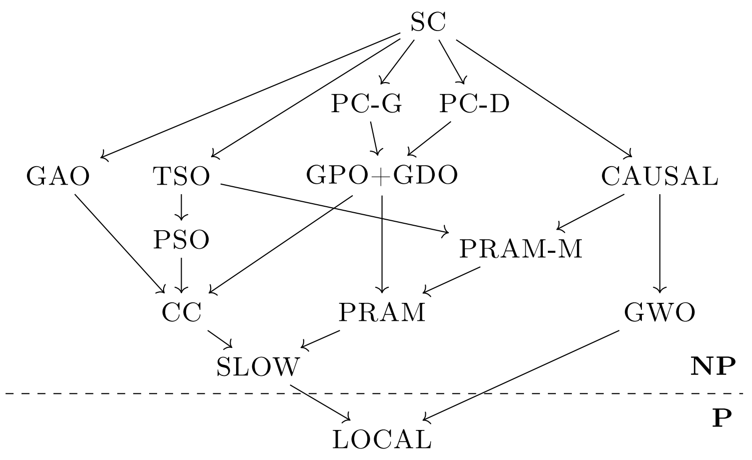


Figure 3.1: Hierarchy of weak memory models [157].

indicated by a path in Figure 3.1, if every execution allowed under $\mathcal{M}_s$ is also consistent with $\mathcal{M}_w$. Memory models are usually defined by axioms [119], in an operational way, or via local views. Steinke and Nutt [157] have shown that many weak memory models can be obtained as a combination of four basic models called GAO, GWO, GDO, and GPO. To be precise, this applies to SC, Pipelined RAM (PRAM) [117], CAUSAL consistency [97], cache consistency (CC) [90], two variants of processor consistency (PC-G, PC-D) [90, 9], SLOW consistency [97], and LOCAL consistency [93]. As a consequence of this characterization via basic models, these memory models form the hierarchy depicted in Figure 3.1. The hierarchy shows that SC is the strongest and LOCAL consistency is the weakest model.

3.1 Overview

We present *algorithms and complexity results for the testing problem under the memory models in the Steinke-Nutt hierarchy.* As shown in Figure 3.1, the general problem is **NP**-complete for all models except for LOCAL. Hence, reductions to SAT lead to optimal testing algorithms. For LOCAL consistency, we provide a polynomial-time testing algorithm. We also conduct a fixed-parameter analysis that explains what makes the testing problem hard.

To derive these results, we develop *a new proof technique and two algorithmic concepts.* For new memory models that are likely to come up, our general concepts will make it easy to devise optimal testing algorithms.

We show that the general testing problem is **NP**-hard for all memory models except for LOCAL. Rather than constructing separate reductions for each memory model, we extend the concept of reductions. We propose *range reductions that cover a range of memory models $\mathcal{M}$ with $\mathcal{M}_S \leq \mathcal{M} \leq \mathcal{M}_W$.* The concept of range reductions demonstrates that hierarchies of architectures are not only useful from a semantic point of view (showing the relationship between models), but also from an algorithmic point of view. Once established, they allow us to propagate hardness results between memory models. We present four range reductions. The first covers the range from SC to SLOW and the second from SC to GWO. Since SLOW and GWO are the weakest models above LOCAL in the Steinke-Nutt hierarchy of Figure 3.1, those two reductions are sufficient to derive all hardness results. The third range reduction from SC to CC and the fourth from SC to PSO provide additional results for fixed parameter testing problems.

We show that the general testing problem under LOCAL and restricted testing problems under SLOW, CC, and PRAM can be solved in polynomial time. It was surprising to us that LOCAL admits a polynomial-time testing algorithm, since it is the intuitive belief that weak memory models make the algorithmic analysis harder. Technically, LOCAL has its own testing algorithm. The algorithms for SLOW, CC, and PRAM again rely on a common idea: *determinization.* Threads are given as sequences of events. We first develop non-deterministic algorithms that read these sequences in polynomial time. Then we show how to determinize the algorithms, using ideas from the power-set construction for finite automata. We also fix an error from the algorithm for CCin our earlier publication [79].

We show that the testing problem is in **NP** for all models in the Steinke-Nutt hierarchy of Figure 3.1. In the framework of Steinke and Nutt, the testing problem under a memory model is defined as the ability to serialize certain partial orders. We give a universal reduction of the testing problem to SAT. It computes a propositional formula that is satisfiable if and only if the partial order admits a serialization. Since the testing problem is generally **NP**-hard, it follows that these SAT-based testing algorithms are optimal for almost all models. While the reduction to SAT is intuitive, we would like to emphasize that it heavily relies on the view-based formulation of memory models [157]. Phrased differently, one or our key contributions is to observe that the Steinke-Nutt framework is well-suited for SAT.

In many applications, we are rarely faced with the general testing problem. First, current architectures still have a small number of cores, which limits the number of concurrent threads. Second, protocols often consist of short threads which means the number of the read/write events in each thread is limited. Finally, the number of synchronization locations is usually small. For these reasons, we consider variants of the testing problem where one of the three parameters is bounded for all inputs: $\text{TEST}_T(\mathcal{M})$ assumes a fixed number of threads in input tests, $\text{TEST}_S(\mathcal{M})$ fixes the size of threads (i.e. the number of events in a thread), and $\text{TEST}_L(\mathcal{M})$ studies the problem for a fixed number of locations. The analysis of these restricted testing problems allows us to identify the sources of hardness for testing.

Besides the practical applications of the testing problem sketched above, our study was motivated by our curiosity on how a memory model influences the complexity of system analysis. Initially, we speculated about the results and discussed two scenarios. On the one hand, one may argue that program analysis becomes harder when using weaker memory models because the number of states increases with the use of intermediary buffers and caches. This effect is actually observed in the analysis of reachability, where the complexity jumps from **PSPACE** for SC [104] to non-primitive recursive for TSO, PSO, and an approximation of POWER [28, 29]. On the other hand, an analysis may become easier because the program's executions are less constrained, a view that is suggested by Alglave in [11]. Our main finding is that, in case of testing, we can confirm Alglave in a strictly formal way. We show that for stronger memory models the testing problem is **NP**-hard (even under restrictions), while for weaker models it tends towards **P**.

Outline: The chapter is organized as follows. After a discussion of related work, we present reductions of **NP**-hard testing problems to SAT in Section 3.3. Then we develop polynomial-time testing algorithms in Section 3.4. We show that all considered testing problems are in **NP** in Section 3.5. We summarize the results and conclude the chapter in Section 3.6.

The results in this chapter have largely appeared in [78] and more detailed in [79]. Algorithm 2 fixes an error in the publications and Section 3.4.4 is new. In addition, examples to illustrate the range reductions are added.

3.2 Related Work

Despite its many applications, there are not that many works on the algorithmics and complexity of the testing problem. Gibbons and Korach [89] studied the testing problem under SC as well as linearizability. They show that in both cases the problem is **NP**-complete, whereas fixed-parameter variants can be solved in polynomial time. Cantin, Lipasti, and Smith [50] extended these results. They state **NP**-completeness of the testing problem under SPARC's memory models (TSO, PSO, RMO) [168], processor consistency PC, release consistency, and a model of the PowerPC architecture. Conflict serializability was studied in [74], with and without synchronization. Bouajjani et al. [173] give a fast heuristic for testing under SC and TSO by gradually checking weaker criteria of the memory model. For causal consistency, the testing problem is solved in polynomial time if the execution is data-independent — meaning its behavior does not depend on the values that are written or read [42, 172].

For transactional databases, the testing problem has been studied under different memory models as well [35]. The complexity ranges from polynomial to **NP**-completeness. In distributed systems with multilevel consistency, an execution can entail memory accesses with different levels of consistency. Here, the testing problem is shown to be **NP**-complete [42] but polynomial for data-independent instances [43].

Common to all mentioned approaches is that they are tailored towards few specific memory models. The testing problem, however, is important for virtually all memory models — existing ones and those of future architectures.

We would like to give a remark on [50]. These authors argue as follows: since synchronization primitives can be added to a program so as to enforce SC behavior despite a weak execution environment, the testing problem for weak memory models must be at least as hard as for SC (where it is **NP**-hard due to [89]). This argument does not apply if programs come free from synchronization primitives. This is the case in the presented study, and one purpose of testing is determining where to insert synchronization primitives in order to enforce certain behavior.

Our contributions are comprehensive complexity results for the testing problem. For weak memory models, results about decidability and complexity of verification problems are rare. Reachability has been considered by Atig et al. and shown to be decidable but non-primitive recursive for TSO, PSO [28] and for an approximation of POWER [29]. Robustness requires the absence of causality cycles, and has been shown to be decidable in polynomial space for TSO [40], partitioned global address spaces [49], and for POWER [69]. The testing problem has a lower complexity as it handles single sequences of events rather than sets. Our multi-parameter complexity analysis is related to [70]. Esparza and Ganty study pattern-based verification under SC. We target more complex memory models but consider the weaker testing problem.

Testing can also be understood as an under-approximation of reachability, similar to runtime verification and bounded model checking (BMC). Related work about reachability is presented in Section 4.3.

Finally, we discuss our choice to base this study of the testing problem on the Steinke-Nutt hierarchy rather than the recent framework of Alglave [13]. The view-based formulation is close to formal languages so that we were able to extract polynomial-time algorithms from it. Furthermore, the uniform description makes it easier to identify common properties of different models for the range reductions.

3.3 NP-hard Testing Problems

We derive basic hardness results that guide our search for testing algorithms in the next sections. Interestingly, the main findings in this section are not the hardness proofs themselves, but a new theory of reductions. So-called range reductions allow us to derive several hardness results with only one encoding. Besides economical considerations (we obtain 38 hardness results for different models with only 4 reductions), range reductions show that several models share a common difficulty, and hence give an idea of what makes algorithmic analysis under weak memory models hard. As with reductions, the challenge is of course to find an encoding of an **NP**-hard problem that meets the requirements of a range reduction.

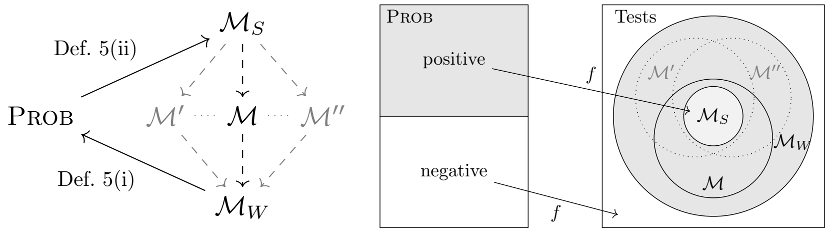


Figure 3.2: Illustration of an $\mathcal{M}_S \leq \mathcal{M}_W$-range reduction of Prob to the testing problem. The left-hand side shows the implications in Def. 5 as solid directed edges. The dashed edges represent the weaker-than relation among memory models. The Venn diagram on the right-hand side illustrates the validity of tests obtained from a range reduction. Positve instances of Prob are mapped to tests that succeed under $\mathcal{M}_S$. Negative instances of Prob are mapped to tests that fail under $\mathcal{M}_W$.

3.3.1 Range Reductions

When we refer to a decision problem Prob, we mean a set of elements together with a predicate $\psi : \text{Prob} \to \{0,1\}$. In the case of testing, $\text{Test}(\mathcal{M})$ contains all tests $\mathcal{T}$ and the predicate asks for an execution of $\mathcal{T}$ that is consistent with $\mathcal{M}$. A reduction $f : \text{Prob} \to \text{Test}(\mathcal{M})$ is a function that maps instances of Prob to tests so that

$$\psi(x) \text{ holds} \quad \text{iff} \quad \text{test } f(x) = \mathcal{T} \text{ succeeds under } \mathcal{M}. \tag{3.1}$$

Our goal is to conclude such an equivalence not only for a single memory model, but for a range of models $\mathcal{M}$ that are weaker than a given model $\mathcal{M}_S$ and stronger than another model $\mathcal{M}_W$. To this end, we reformulate Equivalence (3.1) in such a way that it comprises all models $\mathcal{M}_S \leq \mathcal{M} \leq \mathcal{M}_W$.

Definition 5. *A function f from instances of* Prob *to tests is an $\mathcal{M}_S \leq \mathcal{M}_W$-*range reduction of Prob to the testing problem *if the following holds:*

(i) If test $f(x) = \mathcal{T}$ succeeds under $\mathcal{M}_W$, then predicate $\psi(x)$ holds.

(ii) If predicate $\psi(x)$ holds, then test $f(x) = \mathcal{T}$ succeeds under $\mathcal{M}_S$.

Fig. 3.2 illustrates the definition. If function f is polynomial-time computable and Prob is **NP**-hard, we derive **NP**-hardness of the testing problem.

Lemma 1. *Let* Prob *be* **NP***-hard and let f be a polynomial-time computable $\mathcal{M}_S \leq \mathcal{M}_W$-range reduction of* Prob *to the testing problem. Then* $\text{Test}(\mathcal{M})$ *is* **NP***-hard for all memory models $\mathcal{M}_S \leq \mathcal{M} \leq \mathcal{M}_W$.*

Proof. We show Equivalence (3.1) for any $\mathcal{M}$ with $\mathcal{M}_S \leq \mathcal{M} \leq \mathcal{M}_W$. Assume $\psi(x)$ holds. With Definition 5(ii), test $f(x) = \mathcal{T}$ succeeds under $\mathcal{M}_S$. Since $\mathcal{M}_S \leq \mathcal{M}$, the test remains successful under $\mathcal{M}$. For the reverse direction, assume $\mathcal{T}$ succeeds under $\mathcal{M}$. Then the test remains successful under $\mathcal{M}_W$. With Definition 5(i), $\psi(x)$ holds. Since f is polynomial-time computable, **NP**-hardness follows. $\square$

In the remainder of the section, we show that almost all testing problems are **NP**-hard. Using Lemma 1, we achieve this with only two range reductions. We give reductions of SAT to the testing problem that range from SC to SLOW and from SC to GWO. This covers the full Steinke-Nutt hierarchy except for LOCAL. In Section 3.4, we show that TEST(LOCAL) is indeed in **P**. We give two more reductions that prove hardness results for fixed parameter tests.

When developing range reductions, the challenge is to guarantee the implication in Definition 5(i): $\psi(x)$ follows from a successful test under the weak memory model. The difficulty with weak executions is to ensure a consistent view of events over multiple threads. To derive the implication, the following approach turned out useful. We first construct a reduction to the strong memory model. For the range reductions presented here, this strong model is SC. Then, we modify the reduction so that it remains valid under the weak model. To achieve this, we study the relaxations of the weak memory model (relative to the strong model) and design a test that is insensitive to these relaxations.

3.3.2 Fixed Variable Testing from SC to SLOW

We present an SC $\leq$ SLOW-range reduction of SAT to the testing problem. Formally, we regard a SAT formula φ as a set of clauses where each clause cl is again a set of literals. Given a set of variables $\mathcal{V}$, a literal lit is either a variable $x \in \mathcal{V}$ or a negated variable $\neg x$. Consider a SAT instance $\varphi = \{cl_1 \dots cl_n\}$ with $cl_i = \{lit_{i,1} \dots lit_{i,J_i}\}$. We translate the formula to a test $f(\varphi) = \mathcal{T}$ that satisfies the following. If φ is satisfiable, then $\mathcal{T}$ succeeds under SC. Moreover, if the test succeeds under SLOW, then the formula is satisfiable. The challenge is to satisfy this second requirement: conclude satisfiability of the SAT instance despite the weak executions under SLOW. The trick is to observe that SLOW preserves the order of events on the same variable, and then introduce only one variable ξ used by the threads in the reduction. The data domain $\mathcal{D}$ of ξ is defined as follows. For each variable $x \in \mathcal{V}$ in the SAT instance φ there is a corresponding value $x \in \mathcal{D}$, and for each clause $cl \in \varphi$ of the SAT instance there is a value $cl \in \mathcal{D}$. The test consists of a thread t, and for each variable $x \in \mathcal{V}$ in the propositional formula φ, there are two further threads p_x and n_x:

$$\mathcal{T} \;:=\; \prod_{x \in \mathcal{V}} (p_x \parallel n_x) \parallel t.$$

To explain the construction, we first extract the clauses $cl \in \varphi$ that contain a propositional variable x positively or negatively:

$$\begin{aligned} POS(x) &:= \{cl \in \varphi \mid x \in cl\} \\ NEG(x) &:= \{cl \in \varphi \mid \neg x \in cl\}. \end{aligned}$$

Test $\mathcal{T}$ defines two threads for each variable $x \in \mathcal{V}$. The first thread p_x writes to ξ, one by one, the clauses cl that contain x positively. Afterwards, the thread writes x to ξ to indicate that the variable has been handled. The second thread n_x is similar, but writes the clauses that contain x negatively. The intuition is as follows. If thread p_x is executed, variable x is set to true, and consequently all clauses that contain x positively will be satisfied. Likewise, if thread n_x is executed, then variable x is false, and all clauses that have negative occurrences of x are satisfied:

$$\begin{aligned} p_x &:= [\bullet_{cl\in POS(x)}(w,\xi,cl)].(w,\xi,x) \\ n_x &:= [\bullet_{cl\in NEG(x)}(w,\xi,cl)].(w,\xi,x). \end{aligned}$$

We use $\bullet_{cl\in POS(x)}$ to denote an iterated concatenation of the following events, assuming a total ordering on the clauses.

Finally, the $\mathcal{T}$ contains a test thread t. For its definition, we again use the iterated concatenation and assume the same order of clauses as above:

$$t := [\bullet_{x\in\varphi}(r,\xi,x)].[\bullet_{cl\in\varphi}(r,\xi,cl)].$$

Thread t first reads all values x from ξ, to make sure that for each variable x in the SAT instance either p_x or n_x has terminated. Assume that only one of the threads has terminated when thread t proceeds with reading clause values from its variable ξ. Say n_x has terminated. The remaining thread p_x effectively assigns true to variable x. Thread t continues to read all clauses in its second part. Here, It ensures that all clauses are satisfied by the variable assignment formed by the remaining threads.

Theorem 1. *Both* $\text{Test}(\mathcal{M})$ *and* $\text{Test}_L(\mathcal{M})$ *are* **NP**-*hard for all memory models* $\text{SC} \leq \mathcal{M} \leq \text{SLOW}$.

Proof. We show that the above function f is an $\text{SC} \leq \text{SLOW}$-range reduction of SAT to the testing problem that is polynomial-time computable. We claim that test $\mathcal{T}$ is indeed successful under SC if φ is satisfiable. To see this, note that a satisfying assignment for φ acts as a mapping from clauses to variables. Each clause cl has a variable x that satisfies this clause when set to true or false. Assume x has to be false to satisfy cl. We execute p_x completely and read (r,ξ,x) in thread t. The full thread n_x remains, and we write cl to ξ where needed to satisfy (r,ξ,cl).

To see that a SLOW execution of $\mathcal{T}$ gives a satisfying assignment for φ, note that the test only has one variable. Therefore, thread t observes all writes in program order. This means, whenever it processes an (r,ξ,x) it has either processed p_x or n_x before. With this, the execution defines a mapping from clauses to variables. If (r,ξ,cl) in t receives its value from p_x, then x can be set to true to satisfy cl. Since a consistent execution means we satisfy all clauses, we obtain a satisfying assignment for φ. As the reduction only uses one variable, even $\text{Test}_L(\mathcal{M})$ is **NP**-hard for all memory models $\text{SC} \leq \mathcal{M} \leq \text{SLOW}$. ☐

The theorem shows that testing is **NP**-hard if the events of one thread to the same variable cannot be reordered — which is the case in most memory models. The reduction relies, however, on an unbounded number of threads. As we will show, $\text{Test}(\text{SLOW})$ becomes polynomial if we fix this parameter.

Example We illustrate this reduction on the formula

$$\varphi = \underbrace{(x \vee y)}_{cl_1} \wedge \underbrace{\neg x}_{cl_2}.$$

The test $\mathcal{T} = f(\varphi) = p_x \parallel n_x \parallel p_y \parallel n_y \parallel t$ is as follows:

$$\begin{aligned} p_x &:= (w, \xi, cl_1).(w, \xi, x) \\ n_x &:= (w, \xi, cl_2).(w, \xi, x) \\ p_y &:= (w, \xi, cl_1).(w, \xi, y) \\ n_y &:= (w, \xi, y) \\ t &:= (r, \xi, x).(r, \xi, y).(r, \xi, cl_1).(r, \xi, cl_2). \end{aligned}$$

We use the satisfying assignment $x = 0,\ y = 1$ to construct the following *serial view*:

$$\begin{aligned} <_{sv} := {} & (w, \xi, cl_1).(w, \xi, x).(r, \xi, x).(w, \xi, y).(r, \xi, y). \\ & (w, \xi, cl_1).(r, \xi, cl_1).(w, \xi, cl_2).(r, \xi, cl_2).(w, \xi, x).(w, \xi, y). \end{aligned}$$

The execution rf is implicit, every read accesses the preceding write. The serial view begins with threads p_x and n_y followed by the corresponding reads of values x and y by t. Then, the actual assignment is processed. It is ensured that the assignment satisfies both clauses by first writing the values cl_1 and cl_2 using n_x and p_y and then reading the values using t.

This serial view clearly shows that the test succeeds under SC. Since it only contains one location ξ and only one thread contains reads, it is also the only non-trivial serial view of SLOW. It follows that the test has the same semantics under SLOW.

3.3.3 Fixed Size and Variable Testing from SC to GWO

*Global write order (*GWO*)* is one of the four basic memory models defined by Steinke and Nutt (cf. Section 1 and [157]). We define the reduction of the program order po to only one thread p and to the write events respectively:

$$po_p \ := \ po \cap (*, *, *, p, *)_{\mathcal{T}} \times (*, *, *, p, *)_{\mathcal{T}}.$$

$$po_w \ := \ po \cap (w, *, *, *, *)_{\mathcal{T}} \times (w, *, *, *, *)_{\mathcal{T}}.$$

The GWO model requires the following consistency: if a thread observes an order between two writes (because it reads from the first write and later performs the second), then all threads agree on this order. Formally, we introduce the *write-read-write order* $\mathsf{wo} := rf \ ; \ po_w$. It holds

$$w_1 \xrightarrow{\mathsf{wo}} w_2 \text{ if } \exists r \in \mathcal{T} : \ w_1 \xrightarrow{rf} r \ \wedge \ r \xrightarrow{po} w_2.$$

Definition 6. *An execution $rf \ \subseteq \mathcal{T} \times \mathcal{T}$ is* consistent with GWO *if*

$$\forall p \in \mathcal{ID} \ \exists <_{sv} \in \ SerialView\big(rf, (*, *, *, p, *)_{\mathcal{T}} \cup (w, *, *, *, *)_{\mathcal{T}}, po_p \cup \mathsf{wo}\big).$$

We construct an SC ≤ GWO-range reduction of SAT to the testing problem. The idea is to add reads that enforce a global order wo among critical write events. Given a formula φ, we construct a test $f(\varphi) = \mathcal{T}$ that satisfies the following. If φ is satisfiable then $\mathcal{T}$ succeeds under SC, and if the test succeeds under GWO, then the formula is satisfiable. There is a variable a for the assignment, a variable c for checking clauses, and auxiliary variables h and b. The values of the variables a and b can be pairs. This allows us to only use a fixed number of variables. For each literal lit, let $v(lit)$ determine its variable. Moreover, we set $b(lit) := 1$ for a positive literal and $b(lit) := 0$ for a negative literal. The encoding is

$$\mathcal{T} := \prod_{x \in \mathcal{V}} (n_x \parallel p_x \parallel q_x \parallel r_x) \parallel \prod_{\substack{cl \in \varphi \\ lit \in cl}} l_{lit} \parallel \prod_{cl \in \varphi} t_{cl}.$$

Test $\mathcal{T}$ defines four threads per variable $x \in \mathcal{V}$. The first two threads n_x and p_x write to a, respectively the value $(x, 0)$ or $(x, 1)$, read that value again, and then write $(x, 1)$ to b. The third thread q_x reads value $(x, 1)$ from b and writes value $(x, 2)$ to a:

$$\begin{aligned}
n_x &:= (w, a, \begin{pmatrix} x \\ 0 \end{pmatrix}).(r, a, \begin{pmatrix} x \\ 0 \end{pmatrix}).(w, b, \begin{pmatrix} x \\ 1 \end{pmatrix}) \\
p_x &:= (w, a, \begin{pmatrix} x \\ 1 \end{pmatrix}).(r, a, \begin{pmatrix} x \\ 1 \end{pmatrix}).(w, b, \begin{pmatrix} x \\ 1 \end{pmatrix}) \\
q_x &:= (r, b, \begin{pmatrix} x \\ 1 \end{pmatrix}).(w, a, \begin{pmatrix} x \\ 2 \end{pmatrix}).
\end{aligned}$$

The idea is that at least one of the writes of $(x, 1)$ or $(x, 0)$ to a by p_x or n_x is overwritten with value $(x, 2)$ by thread q_x. Say it is $(x, 0)$. Under the write-read-write order, all threads which read $(x, 2)$ from a can only read the remaining write of $(x, 1)$ by p_x afterwards. This corresponds to the value 1 being assigned to a.

There is a thread l_{lit} for each literal lit. It checks whether the literal is satisfied by the assignment. Let $cl(lit)$ denote the clause cl such that $lit \in cl$. Note this is well defined since each literal is unique and can only occur in one clause. The thread waits for the corresponding variable to be overwritten with 2 and afterwards reads the satisfying value. To be precise, if it is a negative literal of variable x then it reads $(x, 0)$ from a, otherwise it reads $(x, 1)$ from a. After the reads, the thread writes the literal's clause k to variable c:

$$l_{lit} := (r, a, \begin{pmatrix} v(lit) \\ 2 \end{pmatrix}).(r, a, \begin{pmatrix} v(lit) \\ b(lit) \end{pmatrix}).(w, c, cl(lit)).$$

For each clause $cl_k \in \varphi$ there is a test thread t_{cl_k}. It ensures that all clauses up to the current one were satisfied by at least one literal. The test thread reads the value $k - 1$ from variable h of the previous test thread, tries to read the clause cl_k from variable c, and then writes value k to auxiliary variable h:

$$t_{cl_k} := (r, h, k - 1).(r, c, cl_k).(w, h, k).$$

Note that t_{cl_1} reads the initial write $(w, h, 0)$. Thread r_x ensures that after all test threads t_i have finished the remaining literal threads l_{lit} may finish as well. It first checks whether the last auxiliary value n was written to h, and then writes $(x, 0)$ and $(x, 1)$ to a:

$$r_x \;:=\; (r, h, n).(w, a, \begin{pmatrix} x \\ 0 \end{pmatrix}).(w, a, \begin{pmatrix} x \\ 1 \end{pmatrix}).$$

Theorem 2. *The testing problem* $\textsc{Test}(\mathcal{M})$ *as well as* $\textsc{Test}_S(\mathcal{M})$ *and* $\textsc{Test}_L(\mathcal{M})$ *are* **NP***-hard for all memory models* $\mathrm{SC} \leq \mathcal{M} \leq \mathrm{GWO}$.

Proof. Let f be the reduction presented above. We show that function f is an $\mathrm{SC} \leq \mathrm{GWO}$-range reduction of SAT to the testing problem that is polynomial-time computable. We claim that test $\mathcal{T}$ is successful under SC if φ is satisfiable. Consider a satisfying assignment Φ for φ. For a variable $x \in \mathcal{V}$, we execute p_x if $\Phi(x) = 0$ and n_x if $\Phi(x) = 1$. Then, we execute q_x followed by the first read of all literal threads l_{lit} with the same variable: $v(lit) = x$. Afterwards, we execute n_x or p_x, whichever remains. Finally, all literal threads l_{lit} that correspond to satisfied literals with this variable may read the correct value and then halt before the write. We repeat this for all variables. Afterwards, for each clause cl the last event of the literal threads that correspond to satisfied literals of this clause may execute, followed by threads t_{cl}. This is repeated for all clauses. To see that this will work, note that we assume Φ to be satisfying. This means for each clause cl there is a literal thread that wrote cl to c. To conclude, we only have to guarantee that the remaining literal threads terminate. For each variable x we execute r_x up to its first write, followed by all remaining literal threads which correspond to negative literals of x. Then r_x executes its last write such that the remaining literal threads which correspond to positive literals of x can execute.

We claim that a GWO execution of $\mathcal{T}$ induces a satisfying assignment Φ for φ. Recall that the write-read-write order in GWO ensures the following: if one thread issued a write after reading another write, then these two writes are ordered for all threads. In an execution, thread t_k reads value cl_k from the clause variable c. This clause variable is written by some thread l_{lit} corresponding to a satisfied literal. These literal threads determine the assignment: $\Phi(v(lit)) := b(lit)$. To see that Φ is well-defined, consider two literal threads with the same variable x that are both read by test threads. Both literal threads receive their value from the same p_i or n_i, as the other thread n_i or p_i occurs before $(w, a, (x, 2))$ in wo.

As the reduction requires a thread size of at most three, even $\textsc{Test}_S(\mathcal{M})$ is **NP**-hard for all memory models $\mathrm{SC} \leq \mathcal{M} \leq \mathrm{GWO}$. As the reduction requires only four variables, even $\textsc{Test}_L(\mathcal{M})$ is **NP**-hard for all memory models $\mathrm{SC} \leq \mathcal{M} \leq \mathrm{GWO}$. The reduction shows **NP**-hardness of the corresponding restricted versions of the testing problem. $\square$

Example We illustrate this reduction on the formula $\varphi = \underbrace{(x \vee \neg x)}_{cl_1}$.

The test is as follows:

$$
\begin{aligned}
\mathcal{T} &\coloneqq p_x \parallel n_x \parallel q_x \parallel r_x \parallel l_x \parallel l_{\neg x} \parallel t_{cl_1} \\
n_x &\coloneqq (w, a, \binom{x}{0}).(r, a, \binom{x}{0}).(w, b, \binom{x}{1}) \\
p_x &\coloneqq (w, a, \binom{x}{1}).(r, a, \binom{x}{1}).(w, b, \binom{x}{1}) \\
q_x &\coloneqq (r, b, \binom{x}{1}).(w, a, \binom{x}{2}) \\
r_x &\coloneqq (r, h, 1).(w, a, \binom{x}{0}).(w, a, \binom{x}{1}) \\
l_x &\coloneqq (r, a, \binom{x}{2}).(r, a, \binom{x}{1}).(w, c, cl_1) \\
l_{\neg x} &\coloneqq (r, a, \binom{x}{2}).(r, a, \binom{x}{0}).(w, c, cl_1) \\
t_{cl_k} &\coloneqq (r, h, 0).(r, c, cl_1).(w, h, 1).
\end{aligned}
$$

We use the satisfying assignment $x = 0$ to construct the following *serial view*:

$$
\begin{aligned}
<_{sv} \coloneqq\ & \underbrace{(w, h, 0)}_{\epsilon} \underbrace{(w, a, \binom{x}{1}).(r, a, \binom{x}{1}).(w, b, \binom{x}{1})}_{p_x} . \underbrace{(r, b, \binom{x}{1}).(w, a, \binom{x}{2})}_{q_x} . \\
& \underbrace{(r, a, \binom{x}{2})}_{l_{\neg x}} . \underbrace{(r, a, \binom{x}{2})}_{l_x} . \underbrace{(w, a, \binom{x}{0}).(r, a, \binom{x}{0})}_{n_x} . \underbrace{(r, a, \binom{x}{0})}_{l_{\neg x}} . \underbrace{(w, b, \binom{x}{1})}_{n_x} . \\
& \underbrace{(r, h, 0)}_{t_{cl_1}} . \underbrace{(w, c, cl_1)}_{l_{\neg x}} . \underbrace{(r, c, cl_1).(w, h, 1)}_{t_{cl_k}} . \\
& \underbrace{(r, h, 1).(w, a, \binom{x}{0}).(w, a, \binom{x}{1})}_{r_x} . \underbrace{(r, a, \binom{x}{1}).(w, c, cl_1)}_{l_x} .
\end{aligned}
$$

The execution *rf* is again implicit, every read accesses the preceeding write. The serial view begins with p_x. Thread q_x signals that one assignment to x has been processed by setting a to $(x, 2)$. Then, the actual assignment in n_x is processed. Thread $l_{\neg x}$ signals that cl_1 is satisfied by setting c to cl_1 and t_{cl_1} reads this. Afterwards, r_x ensures that l_x can finish. This serial view clearly shows that the test succeeds under SC. Any execution consistent with GWO also establishes a wo order between the key writes which is analogue to a SC serial view. The test has the same semantics under GWO.

3.3.4 Fixed Size and Location Testing from SC to CC

We now introduce the cache consistency model as defined by [157]. Here only the events on the same variable occur in program order:

Definition 7. *An execution $rf \subseteq \mathcal{T} \times \mathcal{T}$ is* consistent with CC *if*

$$\forall x \in \mathcal{L}\ \exists <_{sv} \in \mathit{SerialView}\left(rf, (*, x, *, *, *)_{\mathcal{T}}, po\right).$$

We describe an SC $\leq$ CC-range reduction of SAT to the testing problem that fixes both, the size of threads and the number of locations. Consider a SAT instance φ. We translate it to a test $f(\varphi) = \mathcal{T}$ that satisfies the following. If formula φ is satisfiable, then $\mathcal{T}$ succeeds under SC. Moreover, if the test succeeds under CC, then the formula is satisfiable. Again, our reduction only uses one variable ξ. The data domain $\mathcal{D}$ of ξ is defined as follows. For each variable $x \in \mathcal{V}$, there are two values $(x, 0), (x, 1) \in \mathcal{D}$, and for each clause $cl \in \varphi$ there is a value $cl \in \mathcal{D}$. Furthermore, the numbers $0 \ldots |\varphi|$ are in the data domain. For a positive literal $lit = x$, let $n(lit) := (x, 0)$ and $p(lit) := (x, 1)$. For a negative literal $lit = \neg x$, define $n(lit) := (x, 1)$ and $p(lit) := (x, 0)$ vice versa. The test is:

$$\mathcal{T} \quad := \quad \prod_{x \in \mathcal{V}} (p_x \parallel n_x \parallel r_x) \parallel \prod_{\substack{cl \in \varphi \\ lit \in cl}} l_{lit} \parallel \prod_{cl \in \varphi} t_{cl}.$$

The test defines three threads for each variable x in φ. The first two threads p_x and n_x write to ξ respectively the corresponding value $(x, 0)$ or $(x, 1)$. They model an assignment of φ such that if $(x, 1)$ is observed after $(x, 0)$ then x is set to true and the other way around:

$$p_x := (w, \xi, \begin{pmatrix} x \\ 1 \end{pmatrix}) \qquad n_x := (w, \xi, \begin{pmatrix} x \\ 0 \end{pmatrix}).$$

There is a thread l_{lit} in $\mathcal{T}$ for each literal lit which tests whether it is satisfied by the guessed assignment. If the literal is a positive instance of x then it tries to read $(x, 0)$ and $(x, 1)$ afterwards, if the literal is a negative instance then it tries to read first $(x, 1)$ and then $(x, 0)$. After both reads, it writes the value $cl(lit)$ signaling that the literal's clause is satisfied:

$$l_{lit} := (r, \xi, n(lit)).(r, \xi, p(lit)).(w, \xi, cl(lit)).$$

For each clause cl_i there is a test thread which ensures that all clauses up to the current one were satisfied by at least one literal. Therefore it reads the value corresponding to the previous test thread, tries to read the clause value cl_i and then writes its value:

$$t_{cl_i} := (r, \xi, i - 1).(r, \xi, cl_i).(w, \xi, i).$$

Note that t_1 reads the initial value 0. The threads r_x check for each variable x in φ, whether the last test thread $t_{|\mathcal{V}|}$ was successful by reading $|\mathcal{V}|$ and then write $(x,0)$, $(x,1)$ and again $(x,0)$. This allows all threads corresponding to unsatisfied literals to finish by reading their expected order of values:

$$r_x \coloneqq (r,\xi,|\mathcal{V}|).(w,\xi,\binom{x}{0}).(w,\xi,\binom{x}{1}).(w,\xi,\binom{x}{0}).$$

Theorem 3. *The problems* $\text{T}\small{\text{EST}}(\mathcal{M})$*,* $\text{T}\small{\text{EST}}_L(\mathcal{M})$*, and* $\text{T}\small{\text{EST}}_S(\mathcal{M})$ *are* **NP***-hard for all memory models* $\mathrm{SC} \leq \mathcal{M} \leq \mathrm{CC}$*.*

Proof. Let f be the reduction presented above. We show that the function f is an $\mathrm{SC} \leq \mathrm{CC}$-range reduction of SAT to testing that is polynomial-time computable. We claim that the test $\mathcal{T}$ indeed succeeds under SC if φ is satisfiable.

Given a satisfying assignment Φ, assume variable x is true in this assignment. Thread n_x is executed first, followed by the first reads of all threads l_{lit} that correspond to a positive literal of variable x. Afterwards p_x is executed, again followed by the second reads of the former l_{lit}. This is done successively for each variable. Then, for each clause cl_i, a test thread t_{cl_i} reads the prior value $i-1$. Then the threads l_{lit} that already executed their two reads execute their write of cl_i. This way t_{cl_i} can read the value cl_i and continue writing its value i, thus enabling the next test thread. As Φ is a satisfying assignment, there must exist at least one thread for each clause cl that can write the value cl. This means each test thread t_{cl} can read its value cl. To finish the test, we must also allow the threads corresponding to unsatisfied literals to complete. All threads r_x execute their first read. For each variable x, we execute the write of $(x,0)$ from thread r_x. The write enables the first reads of the remaining threads that correspond to positive literals of x. Then r_x writes $(x,1)$. The write enables the first reads of the remaining threads that correspond to negative literals of x. Moreover, the write allows the second reads of the positive literals of x to execute. Finally, r_x writes $(x,0)$ again. Now the remaining threads corresponding to negative literals of x can execute their second read. Finally, all literal threads l_{lit} that have not yet executed their write can do so.

To see that a CC execution of $\mathcal{T}$ gives a satisfying assignment of φ, note that the test only has one variable and thus consistency under CC and SC coincide. All events are ordered in a total order, especially the order of p_x and n_x is determined for each variable x. In an execution, for each $cl \in \varphi$, the test thread t_{cl} reads cl from one of the writes of a literal thread l_{lit}. As explained before, this corresponds to the literal being satisfied by the modeled assignment. Assuming lit is a positive literal of x then the assignment of x is true, if it is a negative literal of x then the assignment of x is false. This assignment is by construction a satisfying assignment of φ.

The reduction only uses one location and is fixed in the size of threads, This ensures that even $\text{T}\small{\text{EST}}_L(\mathcal{M})$ and $\text{T}\small{\text{EST}}_S(\mathcal{M})$ are **NP**-hard for all memory models $\mathrm{SC} \leq \mathcal{M} \leq \mathrm{CC}$. □

The theorem shows that testing is still **NP**-hard if events to the same location cannot be reordered and the size of threads is fixed. The range reduction relies, however, still on an unbounded number of threads. We will show that the problem is polynomial for a fixed number of threads.

Example We illustrate this reduction on the formula $\varphi = \underbrace{(x \vee \neg x)}_{cl_1}$. The test is as follows:

$$\begin{aligned}
\mathcal{T} &:= p_x \parallel n_x \parallel r_x \parallel l_x \parallel l_{\neg x} \parallel t_{cl_1} \\
p_x &:= (w, \xi, \binom{x}{1}) \\
n_x &:= (w, \xi, \binom{x}{0}) \\
r_x &:= (r, \xi, 1).(w, \xi, \binom{x}{0}).(w, \xi, \binom{x}{1}).(w, \xi, \binom{x}{0}). \\
l_x &:= (r, \xi, \binom{x}{0}).(r, \xi, \binom{x}{1}).(w, \xi, cl_1). \\
l_{\neg x} &:= (r, \xi, \binom{x}{1}).(r, \xi, \binom{x}{0}).(w, \xi, cl_1). \\
t_{cl_1} &:= (r, \xi, 0).(r, \xi, cl_1).(w, \xi, 1).
\end{aligned}$$

We use the satisfying assignment $x = 0$ to construct the following *serial view*:

$$\begin{aligned}
<_{sv} := {} & \underbrace{(w, \xi, 0)}_{\epsilon} \underbrace{(r, \xi, 0).}_{t_{cl_1}} . \underbrace{(w, \xi, \binom{x}{1}).}_{p_x} . \underbrace{(r, \xi, \binom{x}{1})}_{l_{\neg x}} . \underbrace{(w, \xi, \binom{x}{0})}_{n_x} . \underbrace{(r, \xi, \binom{x}{0})}_{l_{\neg x}} . \\
& \underbrace{(w, \xi, cl_1)}_{l_{\neg x}} . \underbrace{(r, \xi, cl_1).(w, \xi, 1)}_{t_{cl_1}} . \underbrace{(r, \xi, 1).(w, \xi, \binom{x}{0})}_{r_x} . \underbrace{(r, \xi, \binom{x}{0})}_{l_x} . \\
& \underbrace{(w, \xi, \binom{x}{1})}_{r_x} . \underbrace{(r, \xi, \binom{x}{1}).(w, \xi, cl_1)}_{l_x} . \underbrace{(w, \xi, \binom{x}{0})}_{r_x} .
\end{aligned}$$

The execution *rf* is again implicit, every read accesses the preceeding write. The serial view incorporates the assignment $x = 0$ in the order it processes p_x and n_x. First $(x, 1)$ is written and then $(x, 0)$. The thread $l_{\neg x}$ reads the values in the same order, it is satisfied by the assignment. Thread $l_{\neg x}$ signals that cl_1 is satisfied by setting ξ to cl_1 and the test thread of cl_1 reads this. Afterwards, r_x ensures that l_x can finish.

This serial view clearly shows that the test succeeds under SC. Since there is only a single location ξ, the serial view shows that the test succeeds under CC as well.

3.3.5 Fixed Thread Testing from SC to PSO

PSO consistency is best explained by describing a possible computer architecture as defined in [28]. Each thread has a set of FIFO buffers, one for each location. These buffers hold the writes to that location that have been executed by the thread but have not yet been commited to memory. If a thread reads from a location, it first snoops the buffer for that variable. In case the buffer is not empty, the read receives the value from the most recent buffered write. Otherwise, if the buffer is empty, the read obtains the value from memory. When a write leaves a buffer and is committed to memory, it becomes visible to all threads. We say that the write has been processed. Since reads obtain their values immediately (either from a buffer or from memory), we say they are processed immediately.

PSO is formally defined by SPARC [168] using axioms. The definition describes how events are observed from the viewpoint of the shared memory. This should be contrasted with the view-based approach focusing on the observations made by the threads. Unfortunately, no view-based definition for PSO has been introduced at this time. Therefore, the following reduction relies on the operational model sketched above [28].

We give an SC $\leq$ PSO-range reduction of 3SAT to testing. Consider a 3SAT instance on variable set X of the form $\varphi = \{cl_1 \dots cl_n\}$ with $cl_i = \{a_i, b_i, c_i\}$. We translate it to a test $f(\varphi) = \mathcal{T}$ that satisfies the following. If formula φ is satisfiable, then $\mathcal{T}$ succeeds under SC. Moreover, if the test succeeds under PSO, then the formula is satisfiable.

Each variable $x \in \mathcal{V}$ of the 3SAT instance φ is also a location. For each literal *lit* of the 3SAT instance, let $loc(lit)$ be the variable occurring in *lit* and let $b(lit)$ be 0 if *lit* is negated and 1 otherwise. For every clause cl_i in the formula and every thread p that we define in the test, we introduce a synchronization variable $z_{i,p}$. Moreover, every literal $l_i \in \{a_i, b_i, c_i\}$ has two synchronization variables $y_{l,0}$ and $y_{l,1}$.

We construct the test $\mathcal{T} := \mathcal{T}_p \parallel \mathcal{T}_q$ as follows:

$$\mathcal{T}_p := p_a \parallel p'_a \parallel p_b \parallel p'_b \parallel p_c \parallel p'_c$$
$$\mathcal{T}_q := q_a \parallel q'_a \parallel q_b \parallel q'_b \parallel q_c \parallel q'_c \ .$$

The test first guesses some assignment Φ for the variables in φ. This is implemented by the threads p_a and p'_a, where p_a writes the value 1 into every variable and p'_a writes 0. The threads p_b and p'_b read these values and thus ensure that the writes left the PSO buffers and reached the memory. The resulting value of a variable x is given by the write which entered the shared memory last, either from p_a or p'_a. Here, we assume that the initial value of a location is not 0 but some other value $\perp$ that is never read. So it is ensured that p'_b actually reads the values of the writes of p_a.

The test then processes each clause $cl_i = a_i \lor b_i \lor c_i$. We introduce *Sync* sequences to ensure that all threads of $\mathcal{T}_p$ handle the same clause. For each clause literal $l_i \in \{a_i, b_i, c_i\}$, the threads p_l and p'_l then perform a read on its variable. Such a read can only be processed if the corresponding literal is satisfied. Afterwards, the variable of the next literal in the clause is inverted and changed back by two *Flip* sequences. This enables the read of the next literal: if it is not already satisfied by Φ, it can be processed after its value is changed by a *Flip*:

$$\begin{array}{llll}
p_a := [\bullet_{x\in\mathcal{V}}(w,x,0)]. & [\bullet_{i\leq|\varphi|}\mathit{Sync}_i(p_a). & (r,\mathit{var}(a_i),b(a_i)). & \mathit{Flip}(b_i,0)] \\
p'_a := [\bullet_{x\in\mathcal{V}}(w,x,1)]. & [\bullet_{i\leq|\varphi|}\mathit{Sync}_i(p'_a). & (r,\mathit{var}(a_i),b(a_i)). & \mathit{Flip}(b_i,1)] \\
p_b := [\bullet_{x\in\mathcal{V}}(r,x,0)]. & [\bullet_{i\leq|\varphi|}\mathit{Sync}_i(p_b). & (r,\mathit{var}(b_i),b(b_i)). & \mathit{Flip}(c_i,0)] \\
p'_b := [\bullet_{x\in\mathcal{V}}(r,x,1)]. & [\bullet_{i\leq|\varphi|}\mathit{Sync}_i(p'_b). & (r,\mathit{var}(b_i),b(b_i)). & \mathit{Flip}(c_i,1)] \\
p_c := & [\bullet_{i\leq|\varphi|}\mathit{Sync}_i(p_c). & (r,\mathit{var}(c_i),b(c_i)). & \mathit{Flip}(a_i,0)] \\
p'_c := & [\bullet_{i\leq|\varphi|}\mathit{Sync}_i(p'_c). & (r,\mathit{var}(c_i),b(c_i)). & \mathit{Flip}(a_i,1)]\ .
\end{array}$$

Test $\mathcal{T}_q$ contains a counterpart for each thread in $\mathcal{T}_p$. The threads in $\mathcal{T}_q$ use *SyncF* sequences to synchronize the *Flip* sequences in $\mathcal{T}_p$.

$$\begin{aligned}
\forall l \in \{a,b,c\} : q_l &= [\bullet_{i\leq|\varphi|}\mathit{SyncF}(l_i,0)] \\
q'_l &= [\bullet_{i\leq|\varphi|}\mathit{SyncF}(l_i,1)]
\end{aligned}$$

The $\mathit{Sync}_i(*)$ sequences are constructed such that they can only be processed when all threads in $\mathcal{T}_p$ have reached their $\mathit{Sync}_i(*)$ sequence. A *Sync* signals that it has been reached by setting its variable $z_{i,*}$ to 1. Then, it attempts to read 1 from the variables of all other *Sync* elements. It follows, that once a *Sync* sequence is processed, the writes of all other *Sync* elements have also been processed. Since reads are processed immediately, all reads before the *Sync* components must also have been processed. For a given clause cl_i and thread p, we define

$$\mathit{Sync}_i(p) := (w,z_{i,p},1).[\bullet_{p'\in\mathcal{T}_P}(r,z_{i,p'},1)].$$

Note that the *Sync* sequences do not ensure that all buffers are empty, only that all preceding reads are processed. For a literal l and a value $v\in\{0,1\}$, we define the sequence $\mathit{Flip}(l,v)$ which inverts the value of $\mathit{loc}(l)$, provided the current value is v:

$$\mathit{Flip}(l,v) := (r,\mathit{loc}(l),v).(w,y_{l,v},1).(w,\mathit{loc}(l),1-v).(r,y_{l,v},0).$$

The purpose of the *SyncF* sequence is to make sure that the inverting write has left the buffer before $\mathit{Flip}(l,v)$ is finished. Note that *SyncF* has to be executed in parallel with the corresponding *Flip*. More precisely *SyncF* is started after *Flip* but is processed before it. This is the case since *SyncF* needs to observe a write setting its synchronization variable $y_{l,v}$ to 1, and *Flip* waits for a matching write of *SyncF* setting the value back:

$$\mathit{SyncF}(l,v) := (r,y_{l,v},1).(r,\mathit{var}(l),1-v).(w.y_{l,v},0).$$

Note, that we can assume w.l.o.g. that the literals in the same clause have different variables, so the *Sync* and *SyncF* sequences of different literals don't interfere with each other.

Theorem 4. *The testing problem* $\text{TEST}_T(\mathcal{M})$ *is* **NP***-hard for all memory models* $\text{SC} \le \mathcal{M} \le \text{PSO}$*.*

Proof. We show that the above function f is an SC $\le$ PSO-range reduction of 3SAT to testing with a fixed number of threads that is polynomial-time computable. First, we prove that for any satisfying assignment Φ there is an SC consistent execution of $\mathcal{T}$ resulting in Φ. Let Pre_i be the the test containing the prefixes of $\mathcal{T}$ ending with $Sync_i$ in $\mathcal{T}_p$ and with $SyncF(l_{i-1}, *)$ in $\mathcal{T}_q$. We use an induction over these prefixes.

Induction Basis $(i = 1)$ Let φ be a formula with a satisfying assignment Φ. There is an interleaving of $[\bullet_{x\in\mathcal{V}}(w, x, 0)]$, $[\bullet_{x\in\mathcal{V}}(w, x, 1)]$, and $[\bullet_{x\in\mathcal{V}}(r, x, 0)]$ and $[\bullet_{x\in\mathcal{V}}(r, x, 1)]$ ending with this assignment in the shared memory. When these sequences have been processed, we execute the *Sync* elements. Obviously, this prefix $Pre_1(F)$ can be processed if the first 0 clauses are satisfied.

Induction Step $(i \to i + 1)$ Assume Φ satisfies the first i clauses. By the induction hypothesis, there is is an execution of Pre_i ending in Φ. Since Φ satisfies at least one of the literals in clause cl_i, the reads belonging to this literal are enabled. Wlog., let $(r, var(a_i), b(a_i))$ be such a read. After it is processed, the variable of b_i is inverted by either $Flip(b_i, 0)$ or $Flip(b_i, 1)$. So if b_i is not satisfied by Φ, the threads p_b and p'_b can now execute their reads $(r, var(b_i), b(b_i))$. Now the next variable is inverted and the reads of c can execute. The second *Flip* inverts the variable again so any execution of Pre_{i+1} ends in the original assignment Φ that was given after Pre_i.

It remains to show that a PSO execution yields a satisfying assignment Φ of the formula. The execution synchronizes the threads at the *Sync* elements. Moreover, the variable assignment remains the same every time the *Sync* elements are processed. Let Φ be that assignment. After the first thread has finished its $Sync_i$ sequence, it performs a read of a literal in clause cl_i. This means every clause contains a satisfied literal and thus Φ satisfies the formula.

We have shown that $\mathcal{T}$ is an SC $\le$ PSO-range reduction of 3SAT to testing with a fixed number of threads. ☐

3.4 Testing Problems in P

We show that for very weak memory models (restricted) testing problems can be solved in polynomial time. To check whether a given test $\mathcal{T}$ succeeds under a memory model, the task is to find an execution $rf \subseteq \mathcal{T} \times \mathcal{T}$ that satisfies certain serial views. Interestingly, the algorithms we propose do not construct the execution but directly construct the serial views.

The intuition is as follows. According to Definition 2(ii), a serial view $<_{sv}$ has to respect a given execution rf. This means for every read r occurring in $<_{sv}$ the serial view implicitly gives the write w with $w \xrightarrow{rf} r$: it is the last write before the read that has the same variable. This suggests that serial views induce a unique execution, and so we only have to compute the serial views. We now develop concepts that make this argument work.

3.4.1 Read Partitioning and Constructive Serial Views

The catch in the argumentation is that serial views are defined for subsets of events $\mathcal{E} \subseteq \mathcal{T}$. This means a serial view only induces a partial execution on this subset. To define the perception of the shared memory for all reads in $\mathcal{T}$, a memory model typically asks for several serial views, say $<^1_{sv}$ for requirement $\mathit{SerialView}\,(rf, \mathcal{E}_1, <_1)$ to $<^k_{sv}$ for $\mathit{SerialView}\,(rf, \mathcal{E}_k, <_k)$. The problem is that the partial executions for $\mathcal{E}_1$ to $\mathcal{E}_k$ may be incompatible. Serial view $<^1_{sv}$ may give $w_1 \xrightarrow{rf} r$ while $<^2_{sv}$ yields $w_2 \xrightarrow{rf} r$ with $w_1 \neq w_2$.

Partial executions can, however, be composed to a full execution of test $\mathcal{T}$ if they do not conflict in the assignment of writes to reads. To ensure this, we call a memory model *read-partitioning* if for every read $r \in \mathcal{T}$ there is precisely one subset of events $\mathcal{E}_j \subseteq \mathcal{T}$ so that $r \in \mathcal{E}_j$. SLOW, PRAM, CC, and the LOCAL model examined below are all read-partitioning.

Serial views are defined relative to an execution. To construct a serial view without knowing the execution, we modify Definition 2, Property (ii):

Definition 8. *Consider a set $\mathcal{E} \subseteq \mathcal{T}$ and a strict partial order $< \subseteq \mathcal{E} \times \mathcal{E}$. A strict total order $<_{csv} \subseteq \mathcal{E} \times \mathcal{E}$ is a* constructive serial view of $\mathcal{E}$ that respects $<$ *if it satisfies the following:*

(i) It refines $<$, which means $< \subseteq <_{csv}$.

(ii') For all reads $r \in \mathcal{E}$ there is a write $w \in \mathcal{E}$ with $loc(w) = loc(r)$, $val(w) = val(r)$, and $w <_{csv} r$. Moreover, there is no $w' \in \mathcal{E}$ so that $w <_{csv} w' <_{csv} r$ and $loc(w) = loc(w')$.

A constructive serial view avoids referencing the execution. Instead it requires that every read r has a preceding write $w <_{csv} r$ with appropriate variable and value. This allows us to reconstruct an execution. In the following lemma, we still assume that memory model $\mathcal{M}$ is defined by the serial views $\mathit{SerialView}\,(rf, \mathcal{E}_1, <_1)$ to $\mathit{SerialView}\,(rf, \mathcal{E}_k, <_k)$.

Lemma 2. *Let $\mathcal{M}$ be read-partitioning and consider a test $\mathcal{T}$. Then $\mathcal{T}$ succeeds under $\mathcal{M}$ if and only if there are constructive serial views $<^i_{csv}$ for $1 \leq i \leq k$.*

For the direction from right to left, note that read partitioning ensures every read r is assigned a unique write predecessor $w \xrightarrow{rf} r$ by its constructive serial view. The union of these assignments is the execution of the full test. Moreover, the constructive serial views are serial views of this execution. The direction from left to right actually holds for every memory model.

3.4.2 Test(LOCAL)

LOCAL consistency is defined as the weakest constraint that every shared memory system should satisfy [93]. It requires that every thread observes all visible events (all writes and its own reads). Moreover, each thread sees its own events in program order but may see the writes of other threads in an arbitrary order. The Steinke-Nutt formulation is as follows [157, Theorem 3.8]:

Definition 9. *An execution* $rf \subseteq \mathcal{T} \times \mathcal{T}$ *is* consistent with LOCAL *if*

$$\forall p \in \mathcal{ID}\ \exists <_{sv} \in\ \mathit{SerialView}\left(rf, (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}, po_p\right).$$

The definition introduces a serial view for each thread p. The corresponding subset $\mathcal{E} \subseteq \mathcal{T}$ contains all events of p as well as all writes in the test. The serial view only has to respect the program order of p. This means the events of p in $\mathcal{E}$ can be understood as a sequence $\tilde{e}_p = e_1 \dots e_n$. The writes of the other threads are given as an unordered set.

Algorithm 1: Constructive Serial View for LOCAL

```
Input: Thread $p$ with $\tilde{e}_p = e_1 \dots e_n$ and set of writes $W$ of all threads
       $q \neq p$
Result: Constructive serial view $s$, initially empty, $s := \varepsilon$.
1  $last[x] := 0$ for all $x \in \mathcal{L}$;
2  for $i = 1 \to n$ do
3      if $e_i = (w,x,v)$ then
4          $last[x] := v$; $s := s.e_i$;
5      else if $e_i = (r,x,v)$ and $last[x] = v$ then
6          $s := s.e_i$;
7      else if $e_i = (r,x,v)$ and $last[x] \neq v$ then
8          if $\exists w \in W : loc(w) = x$ and $val(w) = v$ then
9              $W := W \setminus \{w\}$;
10             $last[x] := v$;
11             $s := s.w.e_i$;
12         else
13             return not LOCAL consistent
14         end
15     end
16 end
17 return $s$ with remaining writes of $W$ inserted at the end;
```

Algorithm 1 computes a constructive serial view of $(*,*,*,p,*)_{\mathcal{T}}$ that respects po_p. It copies $\tilde{e}_p$ to the constructive serial view s, inserting writes from other threads where necessary to satisfy reads. To check whether a write is needed to satisfy a read, we store the last value that has been written to a location x in $last[x]$. The algorithm ensures that every read has a matching preceding write

(Lines 5 and 8). Since writes are inserted only when necessary, the algorithm never fails to find a constructive serial view if there is one.

Algorithm 1 terminates in polynomial time and it has to be called once for every thread in order to determine whether a test succeeds under LOCAL.

Theorem 5. *The testing problem* TEST(LOCAL) *is in* **P**.

3.4.3 $\textsc{Test}_T$(CC)

Although general testing is **NP**-hard for CC(see Section 3.3.4), we will now show that the problem becomes polynomial when we fix the number of threads. To prove this, we give a testing algorithm that is exponential only in the number of threads.

By Definition 7, we need to find a constructive serial view $<_{csv}$ for every $x \in \mathcal{L}$. The corresponding subset of events $\mathcal{E}_x := (*, x, *, *, *)_{\mathcal{T}}$ contains all events on x in the test. The serial view has to respect the program order. This means the events in $\mathcal{E}_x$ are given as sequences $\tilde{e}_p = e_{p,1} \dots e_{p,n_p}$ for every thread p. The task is to find an interleaving of these sequences so that every read obtains the desired value. Since there is only one variable, this is the case exactly if the directly preceding write has the same value. We can assume w.l.o.g. that no read is preceded (in program order) by a write with the same value. Such a read can always follow the write in the constructive serial view and is thus trivial. We can also assume that no thread contains a sequence of multiple identical reads since they can always be put together.

We give non-deterministic Algorithm 2 that solves the problem. In each processing step, Algorithm 2 processes segments of events from some threads and outputs an interleaving (cf. Figure 3.3). More precisely, Algorithm 2 chooses non-deterministically a set of threads $\mathcal{E} = \{p_1 \dots p_n\}$ where the next reads $r_1 \dots r_n$ have the same value v. It processes them up to the reads. By our assumption, the last value that a thread writes to x is different from v. The algorithm non-deterministically chooses a thread $q \notin \mathcal{E}$ that contains a write $w = (w, x, v, q)$ which is not preceded by an unprocessed read. It processes the sequence of writes in q up to and including the first such write. Now the reads are enabled and the algorithm processes them.

The presented algorithm fixes an error in the method previously presented in [79]. The previous method only selected one thread p instead of a set of threads $p_1 \dots p_n$ in a transition. Because of this, it could not always read the same write with multiple reads. We examine the simple test

$$p_1 : (w, x, 0).(r.x.1) \qquad p_2 : (w, x, 0).(r.x.1) \qquad q : (w, x, 1).$$

The previous method can choose the read of either p_1 *or* p_2 with the write of q in a transition and return $(w, x, 0).(w, x, 1).(r.x.1)$. Then it has to give up. Algorithm 2 may choose the reads of p_1 *and* p_2 with the write of q and return $(w, x, 0).(w, x, 0).(w, x, 1).(r.x.1).(r.x.1)$. It then terminates successfully.

Algorithm 2: Constructive Serial View for CC

```
Repeat while possible:
  • If the input sequences contain no more reads,
    return an arbitrary interleaving of the remaining events.
    Then terminate.
  • Otherwise, non-deterministically choose threads $p_1 \dots p_n$ and $q$
    such that the following holds:
    Let $r_i$ be the next read of the remaining input of $p_i$ for $i \leq n$.
    The remaining input of $q$ contains a first write $w$ that has the same
    value $v$ as all reads $r_1 \dots r_n$ and the prefix up to $w$ contains no read.
    Let $\alpha_i.r_i.\beta_i$ be the remainder of $p_i$ and $\gamma.w.\delta$ the remainder of $q$ with
    $\alpha_i, \gamma \in (w, *, *, *, *)^*_{\mathcal{T}}$ and $\beta_i, \delta \in \mathcal{E}^*$.
    Process $\gamma.w$ as well as $\alpha_i.r_i$ for all $i \leq n$ and return $\alpha_1 \dots \alpha_n.\gamma.w.r_1 \dots r_n$.
```

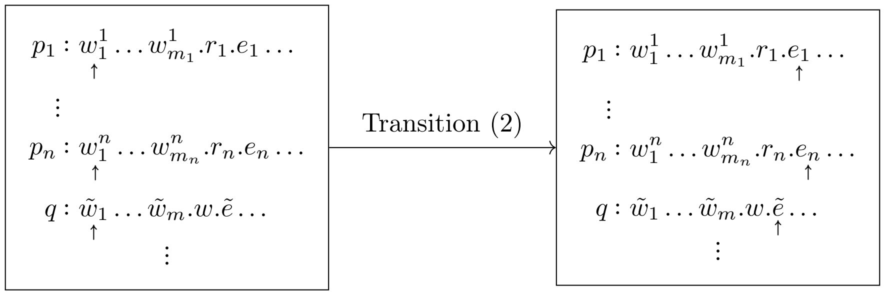

$p_1 : w_1^1 \dots w_{m_1}^1.r_1.e_1 \dots$
$p_n : w_1^n \dots w_{m_n}^n.r_n.e_n \dots$
$q : \tilde{w}_1 \dots \tilde{w}_m.w.\tilde{e} \dots$
Transition (2)
$p_1 : w_1^1 \dots w_{m_1}^1.r_1.e_1 \dots$
$p_n : w_1^n \dots w_{m_n}^n.r_n.e_n \dots$
$q : \tilde{w}_1 \dots \tilde{w}_m.w.\tilde{e} \dots$

Figure 3.3: The algorithm executes the second transition. It chooses reads $r_1 \dots r_n$ on threads $p_1 \dots p_n$ and write w with the same value on thread q. It processes the sequences $\alpha_i := w_1^i \dots w_{m_i}^i$ followed by r_i on p_i for $i \leq n$ and $\tilde{w}_1 \dots \tilde{w}_m.w$ on q. It returns $\alpha_1 \dots \alpha_n.\tilde{w}_1 \dots \tilde{w}_m.w.r_1 \dots r_n$.

Lemma 3. *For every test $\mathcal{T}$ it holds that Algorithm 2 accepts all inputs $\mathcal{E}_x$ restricted to variable x if and only if $\mathcal{T}$ is* CC *consistent.*

Proof. Since for each thread the order of events remains unchanged, the algorithm creates a total order respecting the program order. The algorithm ensures that for every read the preceding write has the correct value. This means the resulting order is a constructive serial view. It follows that the algorithm is correct: if Algorithm 2 accepts all inputs $\mathcal{E}_x$ of a test $\mathcal{T}$, then $\mathcal{T}$ is CC consistent.

It remains to prove completeness. Given a constructive serial view for CC, we can modify it to a sequence of events that is generated by the algorithm. We apply reorderings that respect the serial view property and the program order.

The idea is to move the reads to the front as far as possible without violating Property (ii') of Definition 8 or the program order. Moreover, writes of the same thread are packed together. When a read r receives its value from a write w in the constructive serial view, we select the earliest possible write w' in the same thread as w instead, and order the read after it.

To be precise, we proceed as follows. For any sequence of events α and thread p, let $\alpha|_p$ be the sub-sequence of α that only consists of events with the same thread. Given threads $p_1 \dots p_n$, we define $\alpha|_{\overline{p_1 \dots p_n}}$ as the subsequence of α that only contains events that do not belong to one of the listed threads.

1. Given a maximal sequence of the form $\alpha.w.r_1 \dots r_n$ in $<_{csv}$ such that $\alpha \in (w, x, *, *, *)^+_{\mathcal{T}}$ is a sequence of writes, $w \in (w, x, q, v, *)_{\mathcal{T}}$ a write and $r_i \in (r, x, *, v, *)_{\mathcal{T}}$ a read for $i \leq n$, we apply the following: The sequence $\alpha.w.r_1 \dots r_n$ is replaced by
$$\alpha|_{thrd(r_1)} \dots \alpha|_{thrd(r_n)}.\alpha|_q.w.r_1 \dots .r_n.\alpha|_{\overline{thrd(r_1) \dots thrd(r_n).q}}.$$
2. Given a sequence $e.\alpha.r$ in $<_{csv}$ such that $r = (r, x, q, b)$ is a read of a thread p, $e, \in (*, x, *, b)_{\mathcal{T}}$ is an event with the same value b and $\alpha \in (*, x, *, *)^+_{\mathcal{T}}$ is a sequence of events such that no event in α is from thread p. The sequence $e.\alpha.r$ is replaced by $e.r.\alpha$.

Both actions do not change the order between events of the same thread, they preserve respect to the program order. The first action does not change which write precedes a read. It preserves the constructive serial view property. Consider the second action. Recall that r and e have the same value. Event e is either a write with the same value as r or it is a read where the closest preceeding write of e (and thus also of r) has the same value according to Definition 8 (ii'). It follows that the second action preserves Definition 8 (ii') as well.

Since the actions always move reads forward, they can only be applied finitely many times. We apply them until no longer possible.

The resulting constructive serial view has the following properties: Like any constructive serial view, it can be partitioned into multiple sequences $\alpha.w.r_1 \dots r_n$ of writes followed by reads and finally a suffix of writes. Since we can no longer apply the first action, every such sequence of writes $\alpha.w$ followed by reads $r_1 \dots r_n$ has the following form:
$$\alpha|_{thrd(r_1)} \dots \alpha|_{thrd(r_n)}.\alpha|_q.w.r_1 \dots .r_n.$$

Since we can no longer apply the second action, we can specifically not apply it to a sequence of the form $\alpha|_{thrd(r_n)}.\alpha|_q.w.r_1 \ldots .r_i$ for some $i \leq n$. It follows that $\alpha|_q$ contains no write with the same value as w, it is the first write of the thread segment with the same value as the reads. Any such sequence can be constructed by the second action. The algorithm can do this repeatedly and then produce any suffix of writes. So every serial view with these properties is in the language given by the behavior of the algorithm. It follows that if there is a valid serial view, the algorithm can process the execution. □

Theorem 6. $\textsc{Test}_T$(CC) *is in* **P**.

Proof. Lemma 3 shows that the algorithm solves $\textsc{Test}_T$(CC). To show that the problem is polynomial, it remains to determinize the non-deterministic algorithm. We introduce, for every thread p, a pointer referencing the first event in $\tilde{e}_p$ that has not yet been processed. There are $|\mathcal{ID}|$ many pointers with at most $|\mathcal{E}|$ positions for each pointer. Hence, there are at most $|\mathcal{E}|^{|\mathcal{ID}|}$ many different pointer configurations.

To determinize the algorithm, we store sets of configurations. Like in the powerset construction for finite automata, the current set contains all configurations that the non-deterministic algorithm could have reached after having performed the processing steps so far. With sets of configurations, the algorithm no longer has to guess the threads. Instead, we compute all successor configurations of a given set of configurations. To determine the successor set takes time $2^{|\mathcal{ID}|} \times |\mathcal{E}|^{|\mathcal{ID}|+1}$. We check for every configuration in the current set whether it can reach another configuration by moving the pointers of some threads. Note that there are $2^{|\mathcal{ID}|}$many possible sets of threads and moving the pointers adds another factor of $|\mathcal{E}|$. In every step a read is processed. Since we have at most $|\mathcal{E}|$ many reads, the overall running time of the algorithm is $2^{|\mathcal{ID}|} \times |\mathcal{E}|^{|\mathcal{ID}|+2}$. With $|\mathcal{ID}|$ fixed, the resulting deterministic algorithm processes an input $\mathcal{E}_x$ in polynomial time. □

According to Definition 4, a test succeeds under SLOW if and only if there are constructive serial views for every thread p and variable x such that the following holds. The view contains all events of p on x and all writes to x of other threads. This testing problem amounts to a task that is similar to CC but with the additional restriction that only one thread contains reads. The presented algorithm Algorithm 2 can be used to solve the testing problem for SLOW [77].

Theorem 7. $\textsc{Test}_T$(SLOW) *is in* **P**.

3.4.4 $\textsc{Test}_S$(PRAM)

One of the first common memory models described was PRAM (Pipelined RAM), which was presented 1988 by Lipton and Sandberg [117]. They show that their shared memory system PRAM scales better than sequentially consistent systems as it is immune to high network latency. Additionally, synchronization costs remain low while performance increases significantly.

In a PRAM consistent execution, every thread observes the writes of an other thread in the order they were issued. But two different threads may see writes of several threads in a different order. PRAM consistency based on Steinke and Nutt [157, Theorem 3.2, Definition 4.1] (which itself uses the definition of Ahamad et al. [9]) is as follows:

Definition 10. *An execution $rf \subseteq \mathcal{T} \times \mathcal{T}$ is* consistent with PRAM *if*

$$\forall p \in \mathcal{ID}\ \exists <_{sv} \in\ \mathit{SerialView}\,(rf, (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}, po)\,.$$

We introduce Algorithm 3 that solves $\textsc{Test}_S$(PRAM) by performing the following for every thread p: Let the partial relation rf_p be such that it maps to every read in p a write with the same value and location. For every partial relation rf_p, check whether there is a serial view

$$\mathit{SerialView}\left(rf_p, (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}, po\right).$$

If there is such a *partial execution* rf_p for every thread p, then the execution $rf := \bigcup_{p\in\mathcal{ID}} rf_p$ is consistent with PRAM.

The algorithm tries out all possible partial executions (Line 1). It checks whether there is a corresponding serial view for a partial execution by using the following insight.

Let $<_{sv}$ be a serial view for p and partial execution rf_p. For any thread q that does not occur in the the partial execution, we can move the writes of q to the end of $<_{sv}$. This does not violate Definition 2 (ii) and as long we don't reorder the events of q it still respects po. This means when attempting to find a serial view for a given p and rf_p, it is sufficient to restrict ourselves to the events in the threads that occur in rf_p (Line 2). We can simply assume the events of the remaining threads form the suffix.

Algorithm 3: Pram(p)

```
1 forall partial executions $rf_p = w_1 \xrightarrow{rf_p} r_1 \ldots w_m \xrightarrow{rf_p} r_m$ do
2     $\mathcal{E} := (*,*,*,p,*)_{\mathcal{T}} \cup \bigcup_{i\le m} (w,*,*,thrd(w_i),*)_{\mathcal{T}}$
3     forall total orders $<$ of $\mathcal{E}$ that respect $po$ do
4         if $< \in \mathit{SerialView}\left(rf_p, \mathcal{E}, po\right)$ then
5             return $rf_p$
6     end
7 end
```

Theorem 8. $\textsc{Test}_S(\text{PRAM})$ *is in* **P**.

Proof. We claim that Algorithm 3 solves $\textsc{Test}_S(\text{PRAM})$ in polynomial time. The algorithm is obviously correct. It remains to prove that the run-time is polynomial for a maximal thread size n. A partial execution contains maximal n mappings since p can only contain n reads. The algorithm iterates over at most $|\mathcal{T}|^n$ possible partial executions. Since at most $n+1$ threads occur in rf_p and each thread less than $n+1$ events, the interleaving $<_{sv}$ has at most length $n+1^2$. This means for each partial execution, the algorithm iterates over at most $|\mathcal{T}|^{(n+1)\cdot 2}$ many sequences $<_{sv}$. Checking whether $<_{sv}$ is a serial view can of course be done in polynomial time. The algorithm performs this at most $|\mathcal{T}|^n \cdot |\mathcal{T}|^{(n+1)\cdot 2}$ many times. It is polynomial in $|\mathcal{T}|$. □

3.5 Reduction to Sat

We show that the testing problem is in **NP** for all memory models in the Steinke-Nutt hierarchy. To this end, we propose a polynomial-time reduction of the testing problem to SAT. The main contribution in this section is not so much the SAT encoding (which is quite intuitive), but rather the observation that the results in [157] work well with SAT. The Steinke-Nutt formulation of memory models is well-suited for SAT encodings for two reasons. First, the formulation is uniform: all memory models are defined via serial views, and memory models only differ in the serial views they require. Our SAT encoding inherits this uniformity: we handle all models with one reduction. More precisely, we propose two parameterized formulas that are instantiated and composed as required by a memory model. Second, the definition of whether a test succeeds is simple. It essentially requires to serialize partial orders, which is easily expressed in SAT. Finding a direct reduction of the testing problem to SAT, without using axiomatic descriptions of memory models, appears much harder.

3.5.1 Building Blocks of a Uniform Reduction

We define two propositional formulas in conjunctive normal form: $\text{EXE}(\mathcal{T})$ and $\text{SV}(\mathcal{T}, \mathcal{E}, <)$. The former takes as input a test $\mathcal{T}$ and encodes the existence of an execution. To this end, we introduce variables $ex_{w,r}$ for every pair of write and read events $w, r \in \mathcal{T}$ that use the same variable and access the same value, $loc(w) = loc(r)$ and $val(w) = val(r)$. Formula $\text{EXE}(\mathcal{T})$requires that every read has a write providing its value (left) and no read has two sources (right):

$$\text{EXE}(\mathcal{T}) := \bigwedge_{r \in \mathcal{T}} \left[\bigvee_{\substack{w \in \mathcal{T} \\ loc(w)=loc(r) \\ val(w)=val(r)}} ex_{w,r} \quad \wedge \bigwedge_{\substack{r, w_1, w_2 \in \mathcal{T}, w_1 \neq w_2 \\ loc(w_1)=loc(w_2)=loc(r) \\ val(w_1)=val(w_2)=val(r)}} (\neg ex_{w_1,r} \vee \neg ex_{w_2,r}) \right].$$

Lemma 4. $\text{EXE}(\mathcal{T})$ *is in CNF and cubic in the size of* $\mathcal{T}$. *Moreover,* $\text{EXE}(\mathcal{T})$ *is satisfiable if and only if there is an execution* $rf \subseteq \mathcal{T} \times \mathcal{T}$.

Satisfiability of the second formula $\mathrm{SV}(\mathcal{T},\mathcal{E},<)$ reflects the existence of a serial view of the events $\mathcal{E}$ in an execution. The formula takes as input a test $\mathcal{T}$, a subset of events $\mathcal{E} \subseteq \mathcal{T}$, and a strict partial order $< \subseteq \mathcal{E} \times \mathcal{E}$. Serial views are defined relative to an execution. To access the execution determined by $\mathrm{EXE}(\mathcal{T})$, formula $\mathrm{SV}(\mathcal{T},\mathcal{E},<)$ makes use of the variables $ex_{w,r}$ defined above.

Formally, a serial view is a strict total order $<_{sv} \subseteq \mathcal{E} \times \mathcal{E}$. We encode it with variables sv_{e_1,e_2}, one for each pair of events $e_1, e_2 \in \mathcal{E}$. Intuitively, variable sv_{e_1,e_2} is set to true iff $e_1 <_{sv} e_2$ holds. The following exclusive-or $\oplus$ ensures the serial view is total and asymmetric. The implication encodes transitivity:

$$\bigwedge_{\substack{e_1,e_2,e_3 \in \mathcal{E} \\ e_1 \neq e_3 \\ e_1 \neq e_2 \neq e_3}} \Big[(sv_{e_1,e_2} \oplus sv_{e_2,e_1}) \quad \wedge \quad (sv_{e_1,e_2} \wedge sv_{e_2,e_3} \rightarrow sv_{e_1,e_3}) \Big] \,. \tag{3.2}$$

Definition 2 requires that $<_{sv}$ refines $<$ to a total order:

$$\bigwedge_{\substack{e_1,e_2 \in \mathcal{E} \\ e_1 < e_2}} sv_{e_1,e_2} \,. \tag{3.3}$$

The next formula requires that for every pair $w \xrightarrow{rf} r$ we have $w <_{sv} r$ and that no write to the variable is placed in between:

$$\bigwedge_{\substack{w,r \in \mathcal{E} \\ loc(w)=loc(r) \\ val(w)=val(r)}} \left[(\neg ex_{w,r} \vee sv_{w,r}) \quad \wedge \quad \bigwedge_{\substack{w' \in \mathcal{E} \\ loc(w')=loc(r)}} (\neg ex_{w,r} \vee \neg sv_{w,w'} \vee \neg sv_{w',r}) \right] . \tag{3.4}$$

Formula $\mathrm{SV}(\mathcal{T},\mathcal{E},<)$ is the conjunction of the Formulas (3.2) to (3.4). To state the relationship between *SerialView* $(rf,\mathcal{E},<)$ in Definition 2 and $\mathrm{SV}(\mathcal{T},\mathcal{E},<)$, we restrict the satisfying assignments to the propositional variables. An assignment *respects rf* $\subseteq \mathcal{T} \times \mathcal{T}$ if $e_1 \xrightarrow{rf} e_2$ holds if and only if ex_{e_1,e_2} is set to true.

Lemma 5. $\mathrm{SV}(\mathcal{T},\mathcal{E},<)$ *is in CNF and cubic in its input. There is a strict total order* $<_{sv} \in$ *SerialView* $(rf,\mathcal{E},<)$ *if and only if* $\mathrm{SV}(\mathcal{T},\mathcal{E},<)$ *has a satisfying assignment that respects rf.*

3.5.2 A Uniform Reduction of Testing to SAT

We now show how to instantiate the above formulas to solve the testing problem for the memory models in the Steinke-Nutt hierarchy. We proceed by means of an example: we show how to reduce TEST(SLOW) to SAT. SLOW consistency serves as a representative example. The other models in the hierarchy only differ in the serial views they require.

Computing an execution is equivalent to determining a satisfying assignment for $\mathrm{EXE}(\mathcal{T})$. To make sure that the required serial views exist, we instantiate formula $\mathrm{SV}(\bullet,\bullet,\bullet)$ with appropriate parameters:

$$\mathrm{EXE}(\mathcal{T}) \wedge \bigwedge_{\substack{p \in \mathcal{ID} \\ x \in \mathcal{L}}} \mathrm{SV}(\mathcal{T}, (*,x,*,p,*)_{\mathcal{T}} \cup (w,x,*,*,*)_{\mathcal{T}}, po).$$

Test $\mathcal{T}$ succeeds under SLOW if and only if this formula is satisfiable. Note that the restriction on the admissible assignments in Lemma 5 is no longer needed: THe formula EXE($\mathcal{T}$) ensures that the assignment to the execution variables matches an execution.

One of the necessary parameters is the partial order $<$ that a serial view needs to respect. The encodings of the partial orders used in the Steinke-Nutt hierarchy are all rather straightforward. A detailed description on the encoding of relations is given in Chapter 4.

Theorem 9. TEST($\mathcal{M}$) *is in* **NP** *for all models* $\mathcal{M}$ *defined via serial views.*

All memory models in Figure 3.1 except TSO and PSO are defined via serial views. The testing problem for the latter has been shown to be in **NP** [50].

3.6 Conclusions

We determined the complexity of the testing problem for most known weak memory models. Figure 3.4 shows a summary of our results that cover all models in the Steinke-Nutt hierarchy of Figure 3.1. To derive these results, we developed three general concepts.

1. With range reductions, we proposed a proof technique for lower bounds that hold for a range of memory models. This way, we learned about the importance to construct tests that are insensitive to the relaxations of a memory model. Finding range reductions is challenging. However, they provide insights into the synchronization capabilities of a memory model and guide the search for testing algorithms. Furthermore, range reductions could be used to analyze other problems like reachability or robustness.

2. For very weak models, we developed polynomial testing algorithms, using determinization tricks from automata theory.

3. Finally, we presented a uniform reduction of the testing problem to SAT. It works for all memory models defined via serial views and proves membership in **NP**. Combined with the **NP** lower bounds, these SAT-based testing algorithms are optimal for most memory models.

We note that the three general concepts allowed us to fill the table in Figure 3.4 with only four reductions ($\mathbf{NPC}_{4-7}$) and three algorithms ($\mathbf{P}_{1-3}$). The algorithms in Section 3.4.2, 3.4.3, and 3.4.4 lead to the results $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$ respectively. The **NPC** results $\mathbf{NPC}_4$, $\mathbf{NPC}_5$, $\mathbf{NPC}_6$, and $\mathbf{NPC}_7$ stem from the reductions in Section 3.3.2, 3.3.3, 3.3.4, and 3.3.5, respectively.

We will also give a simple and efficient reduction of the testing problem to the reachability problem of acyclic programs in Section 4.4. The reduction is practical, the constructed reachability instance is roughly the same size as the input of the testing problem. We present a uniform approach to solve the reachability problem in Chapter 4 using a reduction to SMT with a succinct

Mem. Model	Complexity Class of $\textsc{Test}(\mathcal{M})$			
	$\textsc{Test}(\mathcal{M})$	$\textsc{Test}_T(\mathcal{M})$	$\textsc{Test}_S(\mathcal{M})$	$\textsc{Test}_L(\mathcal{M})$
SC	$\mathbf{NPC}_4$	$\mathbf{NPC}_7$	$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
TSO	$\mathbf{NPC}_4$	$\mathbf{NPC}_7$	$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
PSO	$\mathbf{NPC}_4$	$\mathbf{NPC}_7$	$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
PC-G	$\mathbf{NPC}_4$		$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
PC-D	$\mathbf{NPC}_4$		$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
GAO	$\mathbf{NPC}_4$		$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
GPO+GDO	$\mathbf{NPC}_4$		$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
Causal	$\mathbf{NPC}_4$		$\mathbf{NPC}_5$	$\mathbf{NPC}_4$
PRAM-M	$\mathbf{NPC}_4$			$\mathbf{NPC}_4$
GWO	$\mathbf{NPC}_5$		$\mathbf{NPC}_5$	
CC	$\mathbf{NPC}_4$	$\mathbf{P}_2$	$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
PRAM	$\mathbf{NPC}_4$		$\mathbf{P}_3$	$\mathbf{NPC}_4$
SLOW	$\mathbf{NPC}_4$	$\mathbf{P}_2$	$\mathbf{P}_3$	$\mathbf{NPC}_4$
LOCAL	$\mathbf{P}_1$	$\mathbf{P}_1$	$\mathbf{P}_1$	$\mathbf{P}_1$

Figure 3.4: Time complexity of the testing problem under the memory models in the Steinke-Nutt hierarchy of Figure 3.1. We use $\textsc{Test}_T(\mathcal{M})$, $\textsc{Test}_S(\mathcal{M})$, and $\textsc{Test}_L(\mathcal{M})$ for the restricted problems where the number of threads, their size, and the number of locations are fixed, respectively. **NPC** means **NP**-complete: the problem is **NP**-hard and in **NP**. Each index represents a different algorithm or reduction.

encoding. It builds on the key idea behind the encoding in Section 3.5: represent related event pairs as Boolean variables and encode certain ordering constraints on the relations. It is implemented in the DARTAGNAN tool.

The reduction to SAT gives a solution to the testing problem that is both uniform and optimal. However, it is optimal only in the complexity-theoretic sense that it shows membership in **NP**. In practice, the degree of the polynomial in the reduction matters. Therefore, we will introduce more compact encoding techniques in the BMC encoding of program reachability in Section 4.5. Besides the size of the encoding, also the solver technology is important. We will expand the theory of the formula to SAT with integer difference logic (IDL) as this provides efficient ways to encode properties of partial orders.

We have shown that even the relatively simple testing problem is almost always **NP**-hard. This is unfortunate as we expect other memory model aware problems to be even harder. In the following chapters however, we will show that we still manage to solve BMC in **NP**and give a heuristic for the even harder portability problem that is close to **NP** in most practical instances. We even synthesize memory models.

Chapter 4

Bounded Model Checking

After an extensive study of the testing problem in the previous chapter, we now examine the more general reachability problem. The reachability problem is of course only semi-decidable, so we examine a bounded variant. Here, every loop is unrolled a fixed number of times. The previous testing problem describes the reachability problem with additional data where the control-flow of the program and the data-flow are partially known. We know which values are read but we don't know which write event they originate from. It follows that we can expect the bounded reachability problem to be as least as hard as the testing problem. Indeed, it is *NP*-complete as well. We prove this by giving an efficient reduction to SMT, which we implement in the BMC verification tool DARTAGNAN. We also give a reduction of the testing problem to the reachability problem.

The reachability problem under weak memory is of great practical relevance. When developing concurrency libraries or operating system kernels, performance and scalability of the concurrency primitives is of paramount importance. This calls for advanced programming techniques, e.g., lock-free data structures, that rely on the concurrency primitives of the underlying hardware and the programming language runtime environment. The formal semantics of these primitives are often defined in terms of weak memory models which can be subtle. There is considerable interest in verification tools that take into account weak memory models [1, 5, 19, 102]. The success of the `cat` language indicates the need for universal verification tools that are not limited to specific memory models.

We present DARTAGNAN [140], a bounded model checker that takes memory models as inputs. To be precise, DARTAGNAN expects a concurrent program annotated by an assertion and a memory model in the `cat` language for which the verification should be conducted. It verifies the assertion on those executions of the program that are valid under the given memory model, and returns a counterexample execution if the verification fails. Being a bounded model checker, the verification results hold relative to an unrolling bound. Technically, DARTAGNAN determines an acyclic unwinding as common in BMC [55]. The encoding phase, however, is new. Not only the program, also its semantics as defined with the `cat` model, is translated into an SMT formula.

Having to take into account the semantics quickly leads to large encodings. To overcome this problem, Dartagnan implements multiple techniques to keep the encoding small. We use integer difference logic (IDL) to allow for compact representations of properties like acyclicity. We show that instead of the least fixpoint semantics of recursively defined relations in a memory model, it is sufficient to allow any fixpoint. This enables us to use a more straightforward encoding of recursive relations.

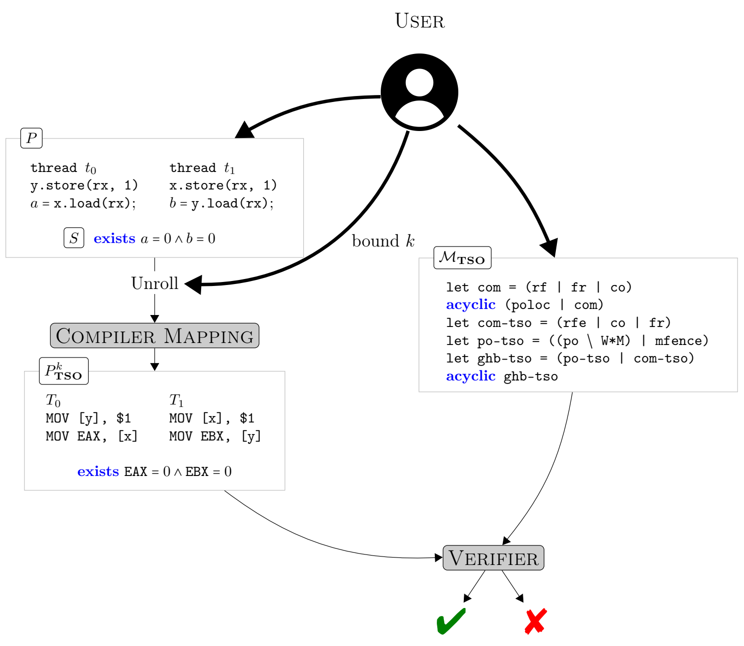
We use a novel *relation analysis*, a static analysis of the program semantics as defined by the memory model. More precisely, `cat` defines the program semantics in terms of relations over the events that may occur in an execution. Depending on constraints over these relations, an execution is considered valid or invalid. Relation analysis determines the pairs of events in a relation that may influence a constraint of the memory model. Any remaining events can be dropped from the SMT encoding.

The tool supports advanced programming constructs. Dartagnan's heap model supports pointers, arrays, and structures. Moreover, the set of synchronization primitives includes (conditional and unconditional) read-modify-write instructions as well as RCU primitives. With this, Dartagnan is on par with Herd7 in its verification capabilities. Most notably, Dartagnan accepts the intermediate verification language Boogie [116] as well as LLVM programs which are translated internally to Boogie using the Smack tool [143].

One motivation for this rich set of programming constructs is the recent proposal for a Linux kernel memory model [22]. It has already been used by the kernel developers to find bugs in and clarify details of the concurrency primitives. The model is expected to be refined and extended as the kernel development proceeds, and verification tools need to be able to quickly accommodate updates in the specification. So far, only the Herd7 tool [20] satisfied this requirement. Unfortunately, it is limited to fairly small programs (litmus tests). Dartagnan offers substantially better performance and thus scales better for larger programs.

We experimented with a series of benchmarks, including 4751 Linux litmus tests and 7 mutual exclusion algorithms executed on TSO, ARM, and Linux. Despite the flexibility of taking memory models as inputs, Dartagnan's performance is comparable to CBMC [19] and considerably better than that of Nidhugg [1, 5] which are both model-specific tools. Compared to the previous version of Dartagnan [138] and compared to Herd7 [20], we gain a speed up of more than two orders of magnitude, thanks to the relation analysis.

Outline: The remainder of the chapter is structured as follows. In Section 4.1 we describe the design of the tool. Section 4.2 presents its the input and parameters. In Section 4.3 is an overview of the related work and Section 4.4 contains a reduction of testing to reachability. Section 4.5 gives the encoding in an SMT formula. Our relation analysis technique is presented in Section 4.6 and Section 4.7 gives an example. A description of our alias analysis is in Section 4.8. This is followed by Section 4.9 which discusses the experimental results and finally a conclusion.



```
thread t0              thread t1
y.store(rx, 1)         x.store(rx, 1)
a = x.load(rx);        b = y.load(rx);

S exists a = 0 ∧ b = 0
```

```
let com = (rf | fr | co)
acyclic (poloc | com)
let com-tso = (rfe | co | fr)
let po-tso = ((po \ W*M) | mfence)
let ghb-tso = (po-tso | com-tso)
acyclic ghb-tso
```

```
T0                T1
MOV [y], $1       MOV [x], $1
MOV EAX, [x]      MOV EBX, [y]

exists EAX = 0 ∧ EBX = 0
```

Figure 4.1: DARTAGNAN from the user's perspective.

The results in this chapter have appeared in [137, 138, 84, 139]. In particular the encoding occurs with a different focus and level of detail in each of the publications. The description of DARTAGNAN and the experiments are based on [139, 84] The reduction of testing, the presentation of the alias analysis, the definition of the control and data flow encodings, as well as the formal correctness proof of the Knaster-Tarski encoding are new in this work.

4.1 Design and Implementation

Dartagnan is implemented in Java. It generates an SMT formula and uses Z3 as the backend solver. Dartagnan 's source code and benchmarks are available online at

https://github.com/hernanponcedeleon/Dat3M.

For programs, the tool allows the .litmus format used by HERD7 [20] (which allows for several assembly languages) , its own .pts format with C11-like syntax and support for C11-atomics and BOOGIE [116].

We present our tool from a user's perspective. We examine the bounded reachability problem it solves together with the required inputs and their formats. Fig. 4.1 illustrates DARTAGNAN s method for checking reachability.

Dartagnan expects a program P annotated with a reachability condition S, a memory model $\mathcal{M}$ of the target architecture, and an unrolling bound k for the bounded model checking. It recursively unwinds all loops in P up to the bound k. The unwound program and the reachability condition are then mapped to the assembly dialect of the target architecture (see Section 2.1). The resulting acyclic and annotated assembly program is handed over to the analysis. In Fig. 4.1, program P is a simplified mutex algorithm which is mapped to x86 (P^k_{TSO}) using the compiler mapping in Table 2.1. Dartagnan then verifies whether $\texttt{EAX} = 0 \wedge \texttt{EBX} = 0$ is reachable when running P^k_{TSO} under TSO. Reachability is used to check basic correctness properties like the validity of assertions or mutual exclusion. Other verification tasks (like the safety fragment of LTL, resource bounds, bounded response/fairness) can be reduced to reachability by instrumenting the program with auxiliary variables or observer threads [167]. In Fig. 4.1, we verify the mutex algorithm by checking whether both threads can read value 0 and thus enter their critical sections. Under TSO, this is possible (see Fig. 2.9).

Dartagnan takes the input program and computes an acyclic unrolling in a program with conditionals but without loops. It encodes this acyclic program together with the memory model into an SMT formula and passes it to the SMT solver Z3 [65]. The formula has the form

$$\phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}} \wedge \phi_S.$$

Here, formulas ϕ_{CF} and ϕ_{DF} encode the control flow and data flow of the program. The memory model dependent condition $\phi_{\mathcal{M}}$ ensures that the executions are *consistent* with the given model. Finally, ϕ_S is satisfied only if the final state reached by an execution satisfies the predicate S.

4.2 Input and Functionality

The tool accepts programs in PPC, x86, AArch64 assembly and the subset of C supported by the .litmus format used by Herd7. It also reads our own .pts format with C11-like syntax. Most notably, Dartagnan accepts the intermediate verification language Boogie [116]. LLVM programs are translated internally to Boogie using the Smack tool [143]. Fig. 4.2 shows the overall architecture of Dartagnan for Boogie programs. Since LLVM has support for many source languages such as C and C++ through clank [170], Dartagnan is applicable to a wide variety of inputs. The SMT solver is Z3 [65].

The key instructions of the input programs are given in Section 2.1. We support the assertion language of Herd7. The syntax is given in Section 2.1.1. Assertions define inequalities over the values of registers and locations. It comes with quantifiers over the reachable states: there is one state, there is none, or all reachable states should satisfy an assertion.

The compiled tool is run by executing the following command:

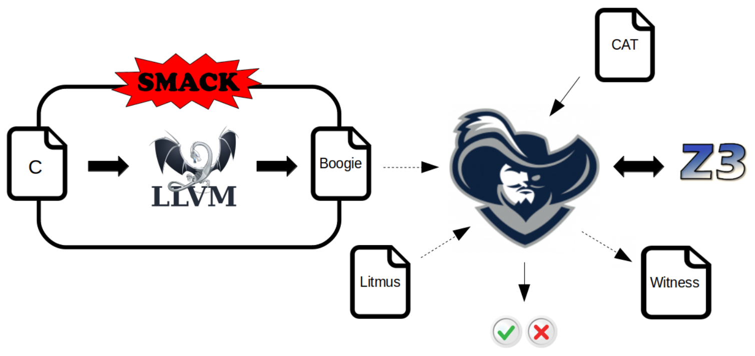


Figure 4.2: DARTAGNAN's architecture.

```
$ java -jar dartagnan/target/dartagnan-V-jar-with-dependencies.jar
  -cat <CAT file> -t <target> -i <program file> [options]
```

DARTAGNAN is invoked with two input files: the program, annotated with an assertion over the final states, and the memory model. adhering to the format given in Section 2.3: Moreover, there is a mandatory parameter target that fixes the definition of the control-flow relation ctrl as well as the compilation to assembly (see Table 2.1). While this could be specified in cat, it is a design decision of the language not to include the control-flow relation, and we offer the parameter for compatibility:

-input <file with the concurrent program annotated by an assertion>
-cat <file with the memory model>. -target {none|arm|arm8|power|tso}

When a violation is found, DARTAGNAN can return a witness execution. Additional parameters can be used to print these counterexample executions (see [140]). The graph always contains relations po, rf, and co which determine an execution. With rels, one can include further relations:

-draw <output file for the execution graph>
-rels <comma-separated list of relations shown in the execution graph>.

There are three optional parameters related to the verification process. Parameter unroll defines the unrolling bound. It is applied to all loops, including nested ones. The definition is a bit involved but standard. With bound two, the outer loop is first unwound once and the inner loop twice. Then outer loop is unwound again and the inner loop is unwound once.

The SMT encoding technique for recursive relations is defined by mode. It supports two encoding modes: Knaster-Tarski and IDL (Integer Difference Logic). They differ in the encoding of the Kleene iteration for fixed points in recursive and transitive relations of the cat model. In both modes, boolean variables are used to indicate presence or absence of a relation between pairs of events. The Knaster-Tarski encoding uses only one Boolean variable, but it encodes an arbitrary (not necessarily the least) fixed point. The IDL mode uses one Boolean variable for all Kleene iterations and an integer variable representing the step in which the pair was added to the relation. We show that the Knaster-Tarski encoding is sufficient for reachability analysis (see Section 4.5.7). We use it as the standard mode since it provides the most compact encoding.

Parameter alias defines whether the alias analysis should be a standard Anderson-style may points-to analysis, a control-flow sensitive variant, or if it should be disabled entirely (cf. Section 4.8).

-unroll <unrolling bound for while-loops> (default 1)
-mode {knastertarski | idl} (default knastertarski)
-alias {none | andersen | cfs} (default andersen).

In order to make sure not to miss a violation, DARTAGNAN also implements an iterative approach. Initially, the bounded model checking algorithm is called with an unrolling bound of one. If it finds a violation or can prove that all loops have been unrolled completely (this is done using unwinding assertions), the verification process terminates with a conclusive answer. If not, DARTAGNAN increases the bound by one and repeats the process. For program with an infinite state space, this does not terminate.

4.3 Related Work

While `cat` is successful as a modeling language, the tool support is lagging behind. As of today, none of the following tools (except HERD7) consider the description of the memory model as an input. They all implement (at best few) concrete models. NITPICK [37], SATCHECK [67], NEMOSFINDER [171], and MEMSAT [161] use SMT solvers. CBMC had been extended to support TSO and POWER [19] but POWER is no longer supported. CPPMEM [31] and HERD7 enumerate all executions, making them less scalable.

HERD7 [20] is the only tool aside from ours that is universal. It takes a CAT memory model as an inputs. Herd does not scale well to programs with a larger number of executions, including some of the larger LINUX kernel tests. The recent verification tool Cerberus-BMC [114] contains a precise reference semantics for C. It also allows for a `cat` definition of the memory model as input, however it is developed specifically towards providing the semantics of C. It is intended as an executable reference semantics for small test programs, not itself as a verification tool.

More efficient but technically involved and hard to generalize are Stateless Model Checkers. NIDHUGG [1, 5] is a stateless model checker supporting TSO, POWER, and a subset of ARMv7. It has recently been adapted to verify the implementation of RCU in the linux kernel [103]. It is excellent for programs with a small number of executions. For programs with a larger number of executions, our method outperforms NIDHUGG. RCMC [102] and CDSCHECKER [132] implement stateless model checking algorithms targeting C / C++11. The prototype tool VBMC [6] uses a reduction to reachability under SC to implement a BMC for programs running under the release-acquire semantics of C / C++11.

A modular proof technique has been introduced recently [14, 16]. It uses invariants to verify programs under a model given in `cat`. It generalizes the methods by Lamport and Owicki-Gries for sequential consistency and introduces a new style of program semantics.

Alglave et al. developed memory-model-aware BMC algorithms [19]. The approach is remarkable in that it applies to various models, and indeed inspired our SAT encodings. We under-approximate the behavior of a concurrent program with a bounded unrolling as well. Another under-approximation technique is bounded context switching (BCS). A parameterized analysis of BCS is given in [53] and Atig et al. extended this idea of BCS to TSO [27] A converse approach is used by Vechev et al., they developed over-approximate verification techniques that prove programs correct [107]. Verification under weak memory models is not limited to static analysis. Runtime verification techniques for TSO and PSO have been developed in [46, 47].

4.4 Reduction of Testing to Reachability

The testing problem is in **NP** for all models in the Steinke-Nutt hierarchy. We have shown this by giving a polynomial reduction of the testing problem to SAT (see Section 3.5). In order to take advantage of the efficient encoding techniques implemented in Dartagnan, we propose a reduction of testing to the reachability problem of an unrolled program under a CAT memory model.

The main idea is that a test corresponds to an unrolled program and a serial view property corresponds to an acyclicity constraint.

Given a test $\mathcal{T}$ and a model that defines properties $\mathit{SerialView}\,(rf, \mathcal{E}_1, <_1)$ ... $\mathit{SerialView}\,(rf, \mathcal{E}_k, <_k)$, the test $\mathcal{T}$ corresponds to a program containing stores and loads with the additional requirement that the value v that a read expects is already known. We encode this property in the assertion. We replace each write (w, x, v), with the corresponding instructions $r = v;\ x = \mathit{store}(r, \mathit{rel})$. For each load r, we add a fresh register q that stores the value which was read and add the constraint $q = v$ to the reachability assertion. It is easy to see that the executions of the program that reach the assertion correspond to the executions of the test.

Since the coherence order is only implied by serial views and different serial views may imply inconsistent orders, we use multiple coherence relations $co_<$ that are restricted to the different serial views. We introduce the corresponding restricted fr relation $fr_< := rf^{-1}; co_<$.

Lemma 6. *A execution rf of a test has a serial view $<_{sv}$ such that $<_{sv} \in \mathit{SerialView}\,(rf, \mathcal{E}, <)$ if and only if there is a total order $co_<$ between all writes in $\mathcal{E}$ such that $\mathit{acyclic}(< \cup co_< \cup rf \cup fr_<)$ holds.*

Proof. " $\Rightarrow$ " : Assume an execution rf of the test has a serial view $<_{sv}$ such that $<_{sv} \in \mathit{SerialView}\,(rf, \mathcal{E}, <)$. We define

$$co_< := <_{sv} \cap \bigcap_{x \in \mathcal{L}} ((w, x, *, *, *)_{\mathcal{T}} \times (w, x, *, *, *)_{\mathcal{T}}).$$

The relations $co_<$ and rf are refined by $<_{sv}$. It follows that if $w \overset{rf}{\rightarrow} r$ and $co_<(w, w')$,then w' occurs after w ($w <_{sv} w'$) and according to Definition 2 (*ii*), w' also occurs after r. It follows that $fr_<$ is also refined by $<_{sv}$ and thus it refines the union $< \cup co_< \cup rf \cup fr_<$. Since $<_{sv}$ is acyclic it follows that every relation it refines is acyclic: $\mathit{acyclic}(< \cup co_< \cup rf \cup fr_<)$.

" $\Leftarrow$ " : Assume there is a total order $co_<$ between all writes on the same location in $\mathcal{E}$ such that $\mathit{acyclic}(< \cup co_< \cup rf \cup fr_<)$ holds. We can easily refine any acyclic relation to a total order $<_{sv}$. We repeatedly remove an element without predecessor from the relation and add it as the last element in the total order. Assume Definition 2 (*ii*) doesn't hold for $<_{sv}$. Then there is a relation $w \overset{rf}{\rightarrow} r$ and a write w' on the same location that occurs after w and thus $co_<(w, w')$ holds. Furthermore r does not occur before w' and thus $r \overset{fr_<}{\rightarrow} w'$ does not hold. From $r \overset{rf^{-1}}{\rightarrow} w$ and $co_<(w, w')$ follows $r \overset{fr_<}{\rightarrow} w'$. This is a contradiction. $\square$

We construct the corresponding program for $\mathcal{T}$ as described above and a CAT memory model that contains constraints $\mathit{acyclic}(<_1 \cup co_{<_1} \cup rf \cup fr_{<_1})$... $\mathit{acyclic}(<_k \cup co_{<_k} \cup rf \cup fr_{<_k})$. The encodings of relations $fr_<$ and $co_<$ as well as the sets $\mathcal{E}$ used by the models in the Steinke-Nutt hierarchy are straightforward to encode. The relation $<$ which is refined by a serial view can easily be defined in `cat` for all memory models in this study. The test succeeds if and only if the reachability check succeeds.

4.5 Encoding Reachability

Dartagnan encodes the reachability problem into an SMT formula which is constructed as follows. Formulas ϕ_{CF} and ϕ_{DF} encode the control flow and data flow. Condition $\phi_{\mathcal{M}}$ requires that the executions are *consistent* with the given memory model and ϕ_S ensures the final state of the execution satisfies assertion S. In the remainder of this section we will elaborate each sub-formula separately. The overall BMC encoding is:

$$\phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}} \wedge \phi_S.$$

The tool processes the input as follows. Each loop in the program is unrolled up to a user defined depth k. The program is compiled using a given mapping (see Table 2.1). This results in a directed acyclic graph presenting all possible control flows of the program up to the unrolling depth. We allow each thread to branch and merge again but the program is now acyclic. This means each statement can only be executed once.

Then, Dartagnan constructs an SMT-query. The main idea of the BMC encoding is to guess an execution, which consists of *executed events* and the *rf* and *co* relations. Guessing the executed events fully specifies the control flow of the candidate execution, while guessing *rf* and *co* specifies the data-flow of the candidate execution. It is easy to see that the data flow of the local registers is uniquely determined by the execution. Note that the encoding of the program semantics is basically the encoding of the weakest possible memory model expressible in `cat`. All widely used models are additional restrictions of this. This part of the encoding is independent off the memory model. It is similar to established BMC encodings of concurrent programs [57].

We model the heap by encoding a new memory location for each variable and a set of locations for each memory allocation of an array. Every location x has an address $ad(x)$, which is an integer variable and its value is chosen by the solver. In an array, the locations are required to have consecutive addresses. The same arithmetic operations are allowed for memory addresses as for regular integer values. Instances of instructions are modeled as events, most notably writes (to the shared memory), reads (from the shared memory), thread-local assignments and fences. A write event has two expressions as parameters, the value it writes and the address it writes to. A read event contains the address it reads from and the register in which it writes the value.

A memory model defines relations among the events in the program. We encode relations by associating pairs of events with Boolean variables. Whether the pair (e_1, e_2) is contained in relation r is indicated by the variable $\mathsf{r}(e_1, e_2)$. Encoding of binary operators on relations $\mathsf{r}_1 \cap \mathsf{r}_2$, $\mathsf{r}_1 \cup \mathsf{r}_2$, $\mathsf{r}_1 ; \mathsf{r}_2$, $\mathsf{r}_1 \setminus \mathsf{r}_2$ is then straightforward. For recursively defined and (reflexive and) transitive relations, DARTAGNAN lets the user choose between two methods for computing fixed points by setting the appropriate parameter (see Section 4.2). The integer-difference logic (IDL) method encodes a Kleene iteration by means of integer variables (one for each pair of events) representing the step in which the pair was added to the relation. The Knaster-Tarski encoding simply looks for a post fixpoint. We will show that this is sufficient for reachability analysis.

4.5.1 Encoding Control Flow

The encoding of the control flow relies on two boolean variables per event: cf_e and $\mathsf{exec}(e)$. Variable cf_e indicates whether an event e has been reached in the control flow. Variable $\mathsf{exec}(e)$ shows if an event has been executed. Since we don't know which events are executed at the time of the encoding, the event sets are not restricted to executed events here. For simple events like basic memory accesses, writes to registers, or if conditionals, the values of cf_e and $\mathsf{exec}(e)$ are equivalent. However, some types of event may deviate from this rule. For example, in conditional read-modify-write atomic events, whether the write component is executed depends on a condition. We elaborate further on the encoding of RMW instructions in Section A.1 of the appendix.

Instead of representing the branching of the program with a tree [62], we use a direct acyclic graph (DAG) capturing the branches of the program and how those merge again. This allows us to keep the size of the control-flow encoding linear w.r.t the unrolled program size as opposed to a tree representation which is exponential in the number of if statements. We encode this in the following formula:

$$\bigwedge_{t \in \mathcal{ID}} (\phi_{CF}(first(t)) \wedge cf_{first(t)}).$$

Together with the encoding of the condition of $\mathsf{exec}(e)$ for any event e, this forms the control flow encoding ϕ_{CF}. For a thread $t \in \mathcal{ID}$, the first event in its control flow is given by $first(t)$. For an event e, the formula $\phi_{CF}(e)$ denotes the behavior of the events that are reached by the control flow if e is reached In other words it forms the encoding of the sub-graph of the control flow that consists of the events reachable from e. Since the control flow graph of the unrolled program is acyclic, we can define $\phi_{CF}(e)$ recursively. For a sequence of events $e_1; e_2$, we define $e_2 \coloneqq suc(e_1)$. It holds that e_2 is reached by the control flow iff e_1 is. If an event e has no successor, then we set $\phi_{CF}(suc(e)) = true$. If an event e is a local computations, a read, a write, or a fence, then it imposes no further restriction on the control-flow:

$$\phi_{CF}(e) = (cf_e \leftrightarrow cf_{suc(e)}) \wedge \phi_{CF}(suc(e)).$$

Given an if event $e_{\text{if}} \coloneqq \texttt{if}\ b\ \texttt{then}\ e_1; \ldots; e_n\ \texttt{else}\ e'_1; \ldots; e'_m$, it holds that if e_{if} is executed, then so is its successor. For the recursive definitions of the control flow, we do not rely only on its successor but on the first events in the if and else branches as well (4.1). Either condition b and thus variable $\mathsf{df}(b)$ (defined by the data flow encoding) is satisfied and e_1 is reached or b is not satisfied and thus e'_1 is reached (4.2).

$$\phi_{CF}(e_{\text{if}}) = \phi_{CF}(e_1) \wedge \phi_{CF}(e'_1) \wedge \phi_{CF}(suc(e_{\text{if}})) \wedge (cf_{e_{\text{if}}} \leftrightarrow cf_{suc(e_{\text{if}})}) \quad (4.1)$$
$$\wedge (cf_{e_1} \leftrightarrow (\phi_{DF}(b) \wedge cf_{e_{\text{if}}})) \wedge (cf_{e'_1} \leftrightarrow (\neg\phi_{DF}(b) \wedge cf_{e_{\text{if}}})) \quad (4.2)$$

4.5.2 Encoding Data Flow

The encoding of the data flow consists of the following components: the thread external data flow via shared locations in the relations rf and co, the internal data flow using registers in idd and $\mathsf{addrDirect}$, the model of the heap ϕ_{Heap}, the final values of locations ϕ_{FVL} and registers ϕ_{FVR} after the execution, and the correct evaluation of expressions and assignments ϕ_{Eval}.

$$\phi_{DF} \coloneqq \phi_{rf} \wedge \phi_{co} \wedge \phi_{\mathsf{idd}} \wedge \phi_{\mathsf{addrDirect}} \wedge \phi_{Heap} \wedge \phi_{FVL} \wedge \phi_{FVR} \wedge \phi_{Eval}.$$

We encode the data flow using relations. For any pair of events $e_1, e_2 \in \mathbb{E}$ and relation $r \subseteq \mathbb{E} \times \mathbb{E}$ we use a Boolean variable $\mathtt{r}(e_1, e_2)$ denoting whether $(e_1, e_2) \in r$ holds.

We ensure that the data flow follows the reads-from (rf) and coherence order (co) relations in the shared locations. The encoding also checks that all executed guards are satisfied, and that all executed data manipulation statements are correctly evaluated.

The rf relation requires that a read gets its value from exactly one write to the same variable. The encoding of reads from ϕ_{rf} ensures that this has to be the case for every read (see Equation 4.3). For every read and write event e, we introduce variables $\mathsf{ad}(e)$ and $\mathsf{val}(e)$ denoting the address in the memory it references and the value it reads from or writes to the address. In Section 4.8, we use an alias analysis to construct for every read event e_r a set of events $Alias(e_r)$ that could access the same locations as e_r. Exactly one writes e_w in that set is related to e_r with rf (see Equations 4.4 and 4.5). If e_r accesses the value written by e_w, then they both access the same address and value (see Equation 4.6).

$$\phi_{rf} := \bigwedge_{e_r \in \mathbb{R}} \mathsf{exec}(e_r) \rightarrow \phi_{rf}(e_r) \tag{4.3}$$

$$\phi_{rf}(e_r) := \bigvee_{e_w \in Alias(e_r) \cap \mathbb{W}} (\mathsf{exec}(e_w) \wedge rf(e_w, e_r)) \tag{4.4}$$

$$\wedge \bigwedge_{\substack{e_w, e'_w \in Alias(e_r) \cap \mathbb{W} \\ e_w \neq e'_w}} \neg(rf(e_w, e_r) \wedge rf(e'_w, e_r)) \tag{4.5}$$

$$\wedge \bigwedge_{e_w \in Alias(e_r) \cap \mathbb{W}} rf(e_w, e_r) \rightarrow (\mathsf{ad}(e_r) = \mathsf{ad}(e_w) \wedge \mathsf{val}(e_r) = \mathsf{val}(e_w)) \tag{4.6}$$

The relation co forms a complete order between all writes to the same address. Its encoding ϕ_{co} is straightforward and we omit its exact description.

Concerning the data flow of the local registers ϕ_{idd} and $\phi_{\mathsf{addrDirect}}$, we require that the value of a register can't originate from just any write to the register. It has to come from the last previously executed event of the same thread that wrote to the register. Let $\mathbb{REG}$ be the set of registers used by the program and for a given register r, we denote the set of events that read from r as $\mathrm{read}(r)$. For every event e that writes to a register, we introduce the Integer variable $\mathsf{reg}(e)$ that denotes the value it writes to the register. For every event e that reads from a register r, we introduce the variable r_e. Since we don't know a priori which events are executed, we need to traverse the acyclic control flow backwards. For every register r and event e that reads from it, we construct a sequence of events $e_0, \ldots, e_n$ that write to r and occur before e in the control flow. The sequence is required to be consistent with the control flow: If $i < j \leq n$, then e_i and e_j are either concurrent or e_i precedes e_j in the control flow graph. Note that e_0 is a dummy event that denotes the initial value of r (and is always executed), so the sequence is never empty. The relation idd connects every event that accesses a register in a computation of a value (in contrast to addrDirect which applies to computations of addresses) to the corresponding event that writes the accessed value to the register. It is encoded as follows. We construct a formula $\phi_{\mathsf{idd}}(e, r)$ that traverses the list backwards until the first executed event

e_i and ensure that the value r_e accessed by e is the same one that was written by e_i denoted by $\mathsf{reg}(e_i)$. We do this recursively:

$$\begin{aligned}
\phi_{\mathsf{idd}} &\coloneqq \bigwedge_{r \in \mathbb{REG}} \bigwedge_{e \in \mathrm{read}(r)} \mathsf{exec}(e) \rightarrow \phi_{\mathsf{idd}}(e, r) \\
\phi_{\mathsf{idd}}(e, r) &\coloneqq \phi_{\mathsf{idd}}(e, r, e_n) \\
\phi_{\mathsf{idd}}(e, r, e_0) &\coloneqq \mathsf{reg}(e_0) = r_e \land \mathsf{idd}(e_0, e) \\
\phi_{\mathsf{idd}}(e, r, e_i) &\coloneqq (\mathsf{exec}(e_i) \land \mathsf{idd}(e_i, e) \land \mathsf{reg}(e_i) = r_e) \\
&\quad \lor (\neg \mathsf{exec}(e_i) \land \neg \mathsf{idd}(e_i, e) \land \phi_{\mathsf{idd}}(e, r, e_{i-1}))
\end{aligned}$$

The corresponding relation addrDirect does the same for registers that are accessed in the computation of an address. Its encoding $\phi_{\mathsf{addrDirect}}$ is analogue.

The encoding of the evaluation of expression and assignments ϕ_{Eval} is straightforward. For any expression, we create a similar term over the corresponding integer or boolean variables. For any assignment (occurring in local computations, reads, or writes), we ensure that both sides have the same value.

4.5.3 Encoding the Heap

The structure of the shared memory is encoded in ϕ_{Heap}. DARTAGNAN identifies memory locations via addresses rather than via variable names. This approach facilitates pointer arithmetics, where the same memory address can be assigned to multiple variables. Memory addresses are encoded as integer constants, and their values are chosen by a solver. The only constraints are that the values must greater than zero and they must be distinct. For a set of heap addresses H, these constraints are encoded as

$$\bigwedge_{i=0}^{|H|} \left(H[i] \geq 0 \land \bigwedge_{j=0}^{i-1} H[i] \neq H[j] \right). \tag{4.7}$$

In addition to the basic variables, DARTAGNAN allows modeling arrays of integers. For each element of an array, it generates an address constant which is encoded in the same way as the addresses of regular variables. In order to force consecutive addresses within arrays, the tool applies the following encoding to each ordered list of array addresses A

$$\bigwedge_{i=1}^{|A|} \left(A[i] = A[i-1] + 1 \right). \tag{4.8}$$

DARTAGNAN does not explicitly model different versions and views of the heap. Instead, it encodes a binding of an event updating a value in the heap with an event reading this value. The binding is determined by the edges of the communication relations rf, idd and addrDirect. If an execution contains such edge between some events e_1 and e_2, it implies that the value observed by e_2 is equivalent to the value stored by e_1.

4.5.4 Encoding Final States and Assertions

In litmus tests, assertions are already defined as an SMT formula (see Section 2.1.1), so no additional conversion is needed. In order to allow the state predicate formula ϕ_S to be expressed in terms of the locations and registers of the original input program, the data flow encoding relates the final state of the unrolled compiled program to the original program.

The final values of global and local variables are determined by the communications relations. For values in the shared memory, the final value is equivalent to a value in the write event which addresses the location in question, is executed, and has no outgoing edge of the coherence order relation. This constraint is encoded as follows:

$$\phi_{FVL} \coloneqq \bigwedge_{w_1 \in \mathbb{W}} \mathsf{last}(w_1) \Leftrightarrow (\mathsf{exec}(w_1) \wedge \bigwedge_{\substack{w_2 \in \mathbb{W}: \\ (w_1, w_2) \in M(co)}} \neg \mathsf{co}(w_1, w_2)) \tag{4.9}$$

$$\wedge \bigwedge_{l \in \mathbb{L}} \bigwedge_{\substack{w \in \mathbb{W} \\ l \in PTS(w, ad(w))}} \Big(\mathsf{last}(w) \wedge \big(\mathsf{addr}(l) = \mathsf{addr}(w)\big)\Big) \Rightarrow \Big(\mathsf{final}(l) = \mathsf{val}(w)\Big). \tag{4.10}$$

Note that we already make use of static analysis to minimize the encoding. In Constraint 4.9, we only check the pairs (w_1, w_2) in the set $M(co)$ that may occur in co according to our relation analysis (see Section 4.6). In Constraint 4.10, we only need to check those writes w with address $ad(w)$ that may point to the location l according to our alias analysis given in Section 4.8.

For registers, the final value is denoted by the last event in the control flow that is executed and writes to the register. This is analogue to ϕ_{idd} in Section 4.5.2. We construct for every register r a sequence of events $e_0, \dots, e_n$ writing to r that is consistent with the control flow. The last executed event denotes the final value of r.

$$\phi_{FVR}(r) \coloneqq \phi_{FVR}(r, e_n) \tag{4.11}$$

$$\phi_{FVR}(r, e_0) \coloneqq \mathsf{reg}(e_0) = \mathsf{final}(r) \tag{4.12}$$

$$\begin{aligned}\phi_{FVR}(e, r, e_i) \coloneqq &(\mathsf{exec}(e_i) \wedge \mathsf{reg}(e_i) = \mathsf{final}(r)) \\ &\vee(\neg\mathsf{exec}(e_i) \wedge \phi_{FVR}(r, e_{i-1}))\end{aligned} \tag{4.13}$$

Assertion can also be used freely throughout the code as assert instructions, rather than being limited to the end of the execution. To encode this, each instructions `assert(exp)` in the program is transformed to a local computation `f` $\leftarrow$ `exp` where the fresh variable `f` $\in$ `F` stores the value of `exp` at the corresponding point of the execution. We refer to the formula $\bigvee_{\mathtt{f} \in \mathtt{F}} \neg\mathtt{f}$ as reachability condition.

4.5.5 Encoding Memory Models

A memory model defined in the `cat` language (see Fig. 2.5) is a constraint system over so-called derived relations together with constraints. The language defines a number of base relations. Their encodings can be obtained directly from the source code of the program (e.g., the program order *po*), from statements corresponding to the synchronization primitives of the used architecture (e.g., memory fences *mfence* on TSO) or they are part of the execution (the *rf* and *co* relations). The encodings of the base relation describes the conditions given in Fig. 2.2 and Fig. 2.3 and ensures that the relations are according to the program semantics. We defined the encoding of *rf* in Section 4.5.2 but we will omit the encodings of the remaining base relations as they are rather straightforward.

Derived relations are built from relations using operators such as union, intersection, difference, composition, transitive closure, etc. We use new Boolean variables to represent the derived relations. Most of the operators can be encoded in SMT in a fairly simple manner.

For the target architecture we need to encode memory model consistency, namely that an execution is consistent. This means that all acyclicity, irreflexivity and emptiness constraints of the memory model are satisfied. We encode acyclicity of a relation r by using non-Boolean variables in our SMT encoding for compactness reasons. We assign each event e a numerical variable $\Psi_e \in \mathbb{N}$ and require that if an event e is related to e' then the numerical value Ψ_e assigned to e is less than the value $\Psi_{e'}$ assigned to e'.

In order to keep the encoding compact, we don't examine every pair of events. It is sufficient to consider only those pairs that might contribute to a constraint violation and could occur in the relation. We call these pairs of events the active set $A(r)$ of relation r. We will show how compute such an active set in Section 4.6. Acyclicity of a relation r is encoded as

$$\mathit{acyclic}(r) \Leftrightarrow \bigwedge_{(e_1,e_2)\in A(r)} (\mathtt{r}(e_1,e_2) \Rightarrow (\Psi^{\mathtt{r}}_{e_1} < \Psi^{\mathtt{r}}_{e_2})).$$

Note that we can impose a total order with all $\Psi^{\mathtt{r}}_{e_1} < \Psi^{\mathtt{r}}_{e_2}$ constraints iff there is no cycle. Our encoding is the same as the SAT + IDL encoding used in [85] where SAT modulo acyclicity is discussed further. The irreflexive and emptiness constraints are encoded as:

$$\begin{aligned}\mathit{irreflexive}(r) &\Leftrightarrow \bigwedge_{(e,e)\in A(r)} \neg\mathtt{r}(e,e)\\ \mathit{empty}(r) &\Leftrightarrow \bigwedge_{(e_1,e_2)\in A(r)} \neg\mathtt{r}(e_1,e_2).\end{aligned}$$

$$
\begin{array}{llll}
\mathtt{r}_1 \cup \mathtt{r}_2(e_1,e_2) \Leftrightarrow \mathtt{r}_1(e_1,e_2) \vee \mathtt{r}_2(e_1,e_2) & r^+(e_1,e_2) & \Leftrightarrow \mathtt{r}(e_1,e_2) \vee r^+;r^+(e_1,e_2) \\
\mathtt{r}_1 \cap \mathtt{r}_2(e_1,e_2) \Leftrightarrow \mathtt{r}_1(e_1,e_2) \wedge \mathtt{r}_2(e_1,e_2) & \mathtt{r}^{-1}(e_1,e_2) & \Leftrightarrow \mathtt{r}(e_2,e_1) \\
\mathtt{r}_1 \setminus \mathtt{r}_2(e_1,e_2) \Leftrightarrow \mathtt{r}_1(e_1,e_2) \wedge \neg \mathtt{r}_2(e_1,e_2) & \mathtt{r}^*(e_1,e_2) & \Leftrightarrow \mathtt{r}^+(e_1,e_2) \vee (e_1 = e_2)
\end{array}
$$

$$
\mathtt{r}_1;\mathtt{r}_2(e_1,e_2) \Leftrightarrow \bigvee_{\{e_3 \in \mathbb{E} \mid (e_1,e_3) \in A(r_1) \wedge (e_3,e_2) \in A(r_2)\}} : \mathtt{r}_1(e_1,e_3) \wedge \mathtt{r}_2(e_3,e_2)
$$

Figure 4.3: Derived relations.

4.5.6 Encoding Derived Relations.

Similar to the basic relations, we use new Boolean variables to represent the derived relations. We only encode edges that are relevant to the memory model according to the relation analysis (see Section 4.6) as denoted by the set of active pairs $A(r)$. For any derived relation $r \subseteq \mathbb{E} \times \mathbb{E}$ and active pair of events $(e_1, e_2) \in A(r)$ we use a Boolean variable $\mathtt{r}(e_1, e_2)$ representing the fact that $(e_1, e_2) \in r$ holds. The encoding determines their values as summarized in Fig. 4.3. For the union (resp. intersection) of two relations, at least one of them (resp. both of them) should hold; set difference requires that the first relation holds and the second one does not; reflexive closure checks if the events are the same or related by r^+; for the composition of relations we iterate over a third event e_3 and check if it belongs to the range of the first relation and the domain of the second. Note that we only have to consider an event e_3 if the edges (e_1, e_3) and (e_3, e_2) may actually occur in r_1 and r_2 respectively.

Computing a reverse relation requires reversing the events. Relation $\mathsf{r}^{-1}(e_1, e_2)$ is equivalent to $\mathsf{r}(e_2, e_1)$. The reflexive and transitive closure r^* checks if the events are either the same or are related by r^+. Finally, we define the transitive closure of r recursively. Here, events are either related according to r or the recursive case holds with a relation composition.

4.5.7 Encoding Recursive Relations

Recursive definitions are supported by `cat`. The semantics of such recursively defined relations are the least fixpoint solution to this system of monotone equations on relations. We argue that for reachability, it is sufficient to encode any fixpoint, not necessarily the least one. The constraints of the memory model (acyclicity, irreflexivity and emptiness) are monotone in the following sense: If a relation fulfills an constraint, all of its subsets will fulfill the constraint as well. The `cat` operators on relations are also monotone (except set difference which is not applied to recursive relations on the right side): Consider $r := (r; r) \cup r_0$, where the operator ";" represents relation composition. If relation r_0 is enlarged or reduced, then so is r.

These observations allow us to apply the Knaster-Tarski Theorem [160]. This is a key insight; we use it to simplify the SMT encoding of `cat` models. We can let the SMT solver freely pick any fixpoint that satisfies all the constraints, as it always contains the least fixpoint, which also satisfies all the constraints. It removes the need to encode the least fixpoints of the `cat` language exactly. We call this the Knaster-Tarski encoding. The encoding of r is thus simply

$$\mathtt{r}(e_1, e_2) \Leftrightarrow \mathtt{r};\mathtt{r}(e_1, e_2) \vee \mathtt{r}_0(e_1, e_2).$$

Theorem 10. *The Knaster-Tarski encoding is correct.*

Proof. We argue that for reachability queries, the Knaster-Tarski encoding is correct by comparing it with an encoding of the least fixpoint. Assume a least fixpoint encoding of a reachability query has a satisfying assignment. Naturally, the least fixpoint also satisfies the Knaster-Tarski encoding as it is a fixpoint.

Assume the least fixpoint encoding is unsatisfiable. Every execution violates some constraint. Any violated acyclicity constraint implies a cycle. Since larger fixpoints only add edges to relations and the operators on relations are monotone, the relations in the constraints can only become larger when switching to the Knaster-Tarski encoding, not smaller. The cycle remains for all larger relations (the constraint is monotone). It follows that a acyclicity constraint violated in an least fixpoint encoding remains violated in our Knaster-Tarski encoding. Similar reasoning holds for irreflexivity and emptiness violations since they are monotone as well. Hence, our Knaster-Tarski encoding is also unsatisfiable. The Knaster-Tarski encoding is satisfiable iff a least fixpoint encoding is satisfiable.

It remains to prove that a satisfying assignment of the Knaster-Tarski encoding does indeed give an execution that is consistent with the memory model. We assume the Knaster-Tarski encoding is satisfied. The assignment contains an execution and correct definitions of basic relations. For any derived relation r, it contains a corresponding definition of a relation that we denote r_{KT}. The encoding of the operators in the Knaster-Tarski encoding only differs from a least fixed point encoding by using some fixpoint instead of the least fixpoint. In addition, all operators on relations are monotone. It follows that $r \subseteq r_{KT}$ holds for any relation r. This means a constraint satisfied by r_{KT} is also satisfied by the smaller r. Since all constraints in the Knaster-Tarski encoding are satisfied, all constraints are also satisfied by the smaller relations defined by the least fixpoint. It follows that the execution satisfies the memory model. □

4.6 Relation Analysis

To minimize the size of the encoding (and hence the solving times), we found it key to reduce the domains over which we encode the relations. We determine for each relation a static over-approximation of the pairs of events that may be in this relation. Even more, we restrict the relation to the set of pairs that might influence a constraint of the given memory model. These restricted sets are the *relation analysis* information (of the program relative to the memory model). For each relation r, we compute two sets of pairs, $M(\mathsf{r})$ and $A(\mathsf{r})$. The former contains so-called *may pairs*, pairs of events that may be in relation r.

This does not yet take into account whether or not the may pairs occur in some constraint of the memory model. The so-called *active pairs* $A(\mathsf{r})$ incorporate this information, and hence restrict the set of may pairs further. As a result of the relation analysis, we only have to introduce Boolean variables $\mathsf{r}(e_1, e_2)$ with the corresponding constraints for the pairs $(e_1, e_2) \in A(\mathsf{r})$ to the SMT encoding. We do not need to encode the pairs $(\mathbb{E} \times \mathbb{E}) \setminus A(\mathsf{r})$.

Note that since we allow for recursive definitions of relations, the algorithm for constructing the may set and the active set is a fixpoint computation. Unconventionally, the two sets propagate their information in different directions. For $A(\mathsf{r})$, the computation proceeds from the constraints and propagates information down the syntax tree of the `cat` memory model. The sets $M(\mathsf{r})$ are computed bottom-up the syntax tree. Interestingly, in our implementation we do not compute the full fixpoint but let the top-down process trigger the bottom-up computation it needs. We say the algorithm is *constraint-driven.*

Both sets are computed as least solutions to a common system of inequalities. As we work over powerset lattices (relations are sets after all), the order of the system will be inclusion. Moreover, we will understand each set $M(\mathsf{r})$ and $A(\mathsf{r})$ as a variable, thereby identifying it with its least solution. To begin with, we give the definition for $A(\mathsf{r})$. In the base case, we have a relation that occurs in a constraint $\mathit{const}(\mathsf{r})$ of the memory model. The inequality is defined based on the shape of the constraint:

$$\begin{aligned} A(\mathsf{r}) &\supseteq M(\mathsf{r}) && (\mathit{empty}) \\ A(\mathsf{r}) &\supseteq M(\mathsf{r}) \cap \mathsf{id} && (\mathit{irreflexive}) \\ A(\mathsf{r}) &\supseteq M(\mathsf{r}) \cap M(\mathsf{r}^+)^{-1} && (\mathit{acyclic}). \end{aligned}$$

If the constraint requires the relation to be empty, all pairs of events that could be contained in the relation are relevant to the constraint. If the constraint requires irreflexivity, then only the pairs of the form (e, e) matter. We therefore intersect the set of may pairs with the identity relation. If the constraint requires acyclicity, we concentrate on the pairs (e_1, e_2) that could occur in a cycle. This is the case for (e_1, e_2) if they could be in r and there could be a path of r from e_2 to e_1. This means (e_2, e_1) may be in relation r^+. Note how the definition of active pairs triggers the computation of may pairs.

If the relation in the constraint is a composed one, the following inequalities propagate the information about the active pairs down the syntax tree of the `cat` memory model:

$$\begin{aligned} A(\mathsf{r}_1) &\supseteq A(\mathsf{r})^{-1} && \text{if } \mathsf{r} = \mathsf{r}_1^{-1} \\ A(\mathsf{r}_1) &\supseteq A(\mathsf{r}) && \text{if } \mathsf{r} = \mathsf{r}_1 \cap \mathsf{r}_2 \text{ or } \mathsf{r} = \mathsf{r}_1 \setminus \mathsf{r}_2 \\ A(\mathsf{r}_1) &\supseteq A(\mathsf{r}) \cap M(\mathsf{r}_1) && \text{if } \mathsf{r} = \mathsf{r}_1 \cup \mathsf{r}_2 \text{ or } \mathsf{r} = \mathsf{r}_2 \setminus \mathsf{r}_1 \\ A(\mathsf{r}_1) &\supseteq \{x \in M(\mathsf{r}_1) \mid (x; M(\mathsf{r}_2)) \cap A(\mathsf{r}) \neq \varnothing\} && \text{if } \mathsf{r} = \mathsf{r}_1; \mathsf{r}_2 \\ A(\mathsf{r}_1) &\supseteq \{x \in M(\mathsf{r}_1) \mid (M(\mathsf{r}_2); x) \cap A(\mathsf{r}) \neq \varnothing\} && \text{if } \mathsf{r} = \mathsf{r}_2; \mathsf{r}_1 \\ A(\mathsf{r}_1) &\supseteq \{x \in M(\mathsf{r}_1) \mid (M(\mathsf{r}_1^*); x; M(\mathsf{r}_1^*)) \cap A(\mathsf{r}) \neq \varnothing\} && \text{if } \mathsf{r} = \mathsf{r}_1^+ \text{ or } \mathsf{r} = \mathsf{r}_1^*. \end{aligned}$$

The definition maintains the invariant $A(\mathsf{r}) \subseteq M(\mathsf{r})$. If a pair (e_1, e_2) is relevant to relation $\mathsf{r} = \mathsf{r}_1^{-1}$, then (e_2, e_1) will be relevant to r_1. We do not have to intersect $A(\mathsf{r})^{-1}$ with $M(\mathsf{r})^{-1}$ because $A(\mathsf{r}) \subseteq M(\mathsf{r})$ ensures $A(\mathsf{r})^{-1} \subseteq M(\mathsf{r})^{-1}$. We can avoid the intersection with the may pairs for the next case as well. Here, $A(\mathsf{r}) \subseteq M(\mathsf{r})$ holds by the invariant and $M(\mathsf{r}) = M(\mathsf{r}_1) \cap M(\mathsf{r}_2)$ by definition (see below). It follows $A(\mathsf{r}_1) \subseteq M(\mathsf{r}_1)$. For union and the other parameter of set subtraction, the intersection with $M(\mathsf{r}_1)$ is necessary as $A(r) \subseteq M(r_1)$ may not hold. There are symmetric definitions for union and intersection for r_2. For a relation r_1 that occurs in a relational composition $\mathsf{r} = \mathsf{r}_1; \mathsf{r}_2$, the pairs $x = (e_1, e_3)$ become relevant that may be composed with a pair (e_3, e_2) in r_2 to obtain a pair (e_1, e_2) relevant to r. Note that for r_2, we again require the may pairs. The definition for r_2 is similar. The definition for the (reflexive and) transitive closure follows the ideas for relational composition. Here, we only require pairs that may occur in a path of r_1 from an event e_1 to e_2 that may lead to an edge (e_1, e_2) that is relevant for r. Since the active sets do not influence the may pairs, we only need to compute them once.

The definition of the sets of may pairs is straightforward and follows the syntax of the `cat` memory model from the bottom up. With $\oplus \in \{\cup, \cap, \; ; \mid\}$ and $\otimes \in \{+, *, -1 \mid\}$, we have

$$\begin{aligned} M(\mathsf{r}_1 \oplus \mathsf{r}_2) &\supseteq M(\mathsf{r}_1) \oplus M(\mathsf{r}_2) \\ M(\mathsf{r}^{\otimes}) &\supseteq M(r)^{\otimes} \\ M(\mathsf{r}_1 \setminus \mathsf{r}_2) &\supseteq M(\mathsf{r}_1). \end{aligned}$$

The definition executes the corresponding operation on the current approximation. Subtraction $\mathsf{r}_1 \setminus \mathsf{r}_2$ is the exception, it is not sound to use an over-approximation of the subtrahend r_2. Here, an under-approximation of the subtrahend is needed. However, as the form of a relation depends too much on the execution, a non-trivial under-approximation for the events pairs in r_2 does not exist. We do not even know which events will be executed at the time of the relation analysis.

Since the may sets form complete lattices and the set of inequalities form a monotone function, we can apply a Kleene iteration [159] in order to compute the least solution for recursively defined relations.

At the bottom level, the may sets are determined by the base relations. They depend on the shape of the relations and the positions of the events in the control flow. For relations po, data, addr, id, int, ext, fencerel(*fence*), ctrl, and rmw, the value of $\mathsf{r}(e_1, e_2)$ is fully defined by two parameters: positions of the events in the control flow and whether they get executed. We assume that for each branch of an input program exists an execution where the branch is taken and compute may pairs solely based on the positions of the events in the control flow. May sets of these relations are immediate. The relation po may only concern pairs (e_1, e_2) where e_2 is reachable from e_1 in the control flow graph. The relations data, addr, fencerel(*fence*), ctrl, and rmw are further restrictions of this.

The relations loc, co and rf are concerned with memory accesses. What makes it difficult to approximate these relations is our support for pointers and pointer arithmetic. This means the addresses are not known at the time of the encoding. Without further information, we have to conservatively assume a memory event may access any address. Hence, any two memory events may access the same address and thus any two stores may be in coherence order, and any load may receive its value from any store. To improve the precision of the may sets for loc, co, and rf, our fixpoint computation incorporates a *may-alias analysis* (see Section 4.8). We use a control-flow insensitive Andersen-style analysis [26]. It incurs only a small overhead and produces a close over-approximation of the may sets. The analysis returns a set of pairs of memory events $Alias \subseteq (\mathbb{W} \cup \mathbb{R}) \times (\mathbb{W} \cup \mathbb{R})$ such that every pair of events outside $Alias$ definitely accesses different addresses. Recall that $\mathbb{W}$ are the write events in the program and $\mathbb{R}$ the reads. Note that the analysis is control-flow insensitive as the given memory model may be very weak [12]. With this, the base cases are as follows:

$$\begin{aligned} M(\mathsf{loc}) &\supseteq Alias \\ M(\mathsf{co}) &\supseteq (\mathbb{W} \times \mathbb{W}) \cap Alias \\ M(\mathsf{rf}) &\supseteq (\mathbb{W} \times \mathbb{R}) \cap Alias. \end{aligned}$$

We stress the importance of the alias analysis for our relation analysis: loc, co, and rf are frequently used as building blocks of composite relations. Excessive may sets will therefore negatively affect the over-approximations of virtually all relations in a memory model, and keep the overall encoding unnecessarily large.

4.7 Relation Analysis on an Example

Consider the program (in the .litmus format) given in Fig. 4.4. The assertion asks whether there is a reachable state with final values EBX = 1, ECX = 0. We analyze the program under the x86-TSO memory model introduced in Section 2.3:

$Consistent_{TSO}$

$$\begin{aligned} &\mathsf{rfe} := \mathsf{rf} \setminus \mathsf{int} \qquad \mathsf{coe} := \mathsf{co} \setminus \mathsf{int} \qquad \mathsf{ghb\text{-}tso} := \mathsf{po\text{-}tso} \cup \mathsf{com\text{-}tso} \cup \mathsf{implied} \\ &\mathsf{com} := \mathsf{co} \cup \mathsf{fr} \cup \mathsf{rf} \qquad \mathsf{com\text{-}tso} := \mathsf{co} \cup \mathsf{fr} \cup \mathsf{rfe} \qquad \mathsf{po\text{-}tso} := (\mathsf{po} \setminus \mathbb{W} \times \mathbb{R}) \cup \mathsf{fence}(\mathsf{mfence}) \\ &\mathsf{implied} := \mathsf{po} \cap (\mathbb{W} \times \mathbb{R}) \cap ((\mathbb{M} \times \mathbb{A}) \cup (\mathbb{A} \times \mathbb{M})) \qquad \mathsf{po\text{-}loc} := \mathsf{po} \cap \mathsf{loc} \end{aligned}$$

$$acyclic(\mathsf{po\text{-}loc} \cup \mathsf{com}) \qquad acyclic(\mathsf{ghb\text{-}tso}) \qquad empty(\mathsf{rmw} \cap (\mathsf{fre};\mathsf{coe}))$$

The semantics of the program under TSO is a set of executions, graphs where the nodes are events and the edges correspond to the relations defined by the memory model. An overview of the possible executions with the required final values is given in Fig. 4.5. For the sake of compactness, we consider only memory events — thread-local computations are omitted.

The atomic exchange instruction xchg(x, r0) gives rise to a pair of read and write events related by rmw. Such reads and writes belong to the set A of atomic read-modify-write events, as shown in Fig. 4.6

```
X86
{x = 0; y = 0; P0:EAX = 1;}
P0               | P1             ;
xchg [x], EAX    | mov EBX, [y]   ;
mov [y], 1       | mov ECX, [x]   ;
exists (P1:EBX = 1 ∧ P1:ECX = 0)
```

Figure 4.4: An Example program.

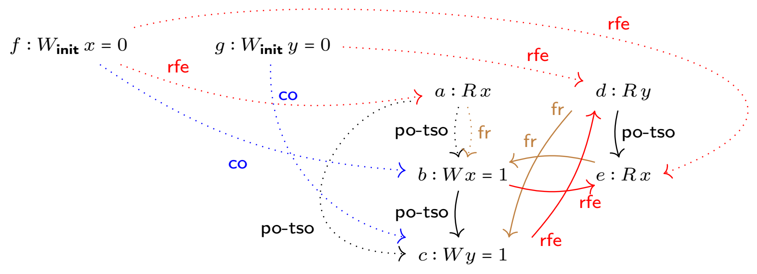


Figure 4.5: Executions under TSO.

Dartagnan encodes the semantics of the given program under the given memory model into an SMT formula. The problem is that each edge (a, b) that may be present in a relation r in a valid execution gives rise to a variable $\mathsf{r}(a, b)$. In the worst case, the number of possible edges of a relation is quadratic in the program size. The goal of our relation analysis is to reduce the number of edges that have to be encoded.

We illustrate this on the constraint $\mathsf{acyclic\ ghb\text{-}tso}$. The graph in Fig. 4.5 shows the 14 (dotted and solid) edges which may contribute to the relation $\mathsf{ghb\text{-}tso}$. They form the *may set* $M(\mathsf{ghb\text{-}tso})$ of the relation (see Section 4.6). Out of those, only the 6 solid edges can occur in a cycle. They form the *active*

$$\begin{tikzcd}
f : W_{\mathsf{init}}\ x = 0 & a : R\,x \arrow[d, "po"] \arrow[d, bend right, "rmw"'] & d : R\,y \arrow[d, "po"] \\
g : W_{\mathsf{init}}\ y = 0 & b : W\,x = 1 \arrow[d, "po"] & e : R\,x \\
 & c : W\,y = 1 &
\end{tikzcd}$$

Figure 4.6: The program order and read-modify-write.

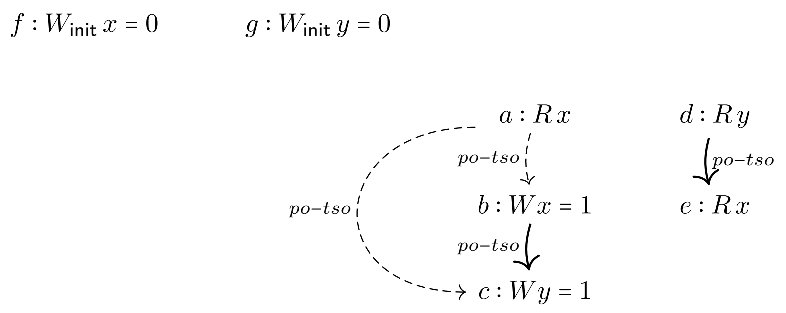


Figure 4.7: $M(\mathsf{po\text{-}tso})$ and $A(\mathsf{po\text{-}tso})$

set $A(\mathsf{ghb\text{-}tso})$ for this constraint. The dotted edges can be dropped from the SMT encoding. Our relation analysis determines the solid edges — edges that may have an influence on a constraint of the memory model. Actually, ghb-tso is a composition of various subrelations (like po-tso or $\mathsf{co} \cup \mathsf{fr}$) that also need to be encoded into SMT. The relation analysis applies to these subrelations as well. Applied to all constraints, it reduces the number of edges that should be encoded in all (sub)relations from 221 to 58. We give a step by step construction of first the may sets and then the active sets.

4.7.1 May Sets

Relation analysis is demonstrated on the constraint acyclic ghb-tso. When the algorithms encounters the constraint, it computes the may pairs for

$$\mathsf{ghb\text{-}tso} = \mathsf{po\text{-}tso} \cup \mathsf{com\text{-}tso} \cup \mathsf{implied}.$$

This triggers calculation of may pairs for its child relations po-tso, com-tso and implied. In Fig. 4.7, we consider relation $\mathsf{po\text{-}tso} = ((\mathsf{po} \setminus \mathsf{W} \times \mathsf{R}) \cup \mathsf{fence(mfence)})$. Recall that the may set consist of both the dashed and solid edges. Since there are no fences, fence(mfence) has an empty may set. The may set of po-tso consists of the may set of po.

The definition of com-tso is $\mathsf{rfe} \cup \mathsf{co} \cup \mathsf{fr}$. An edge of rfe may only start in a write event and end in a read event if both address the same location and do not belong to the same thread. The May set of rfe is a set of all such edges. It is given in Fig. 4.8 Internally, rfe is not a base relation, but an intersection of rf and ext, so, in practice, its May set will be computed as a intersection of May sets for rf and ext.

Edges of co may appear only between writes to the same location. In general, the may pairs of co connect any two writes to the same location in both directions. They are depicted in Fig. 4.9. The exception are the initial writes which cannot have incoming co edges. Here, the two regular write events can only be paired with initial writes.

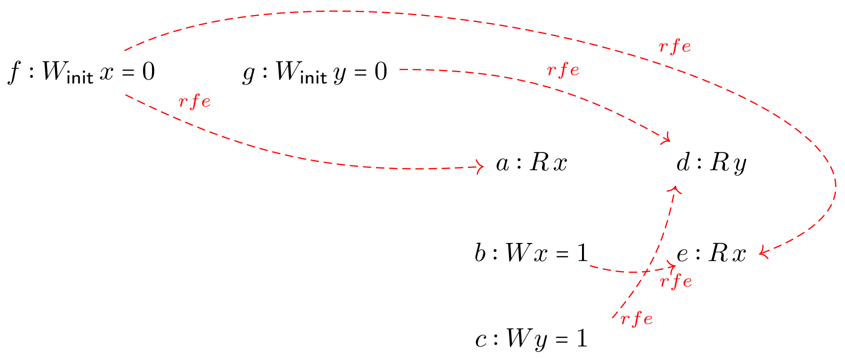


Figure 4.8: $M(\mathsf{rfe})$.

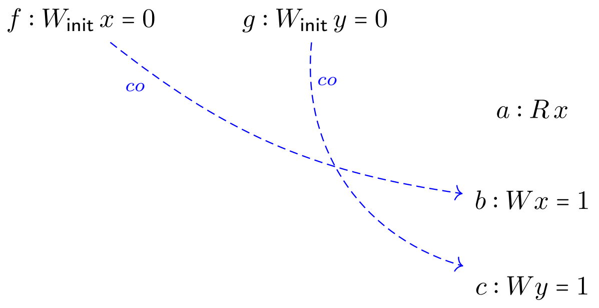


Figure 4.9: $M(\mathsf{co})$.

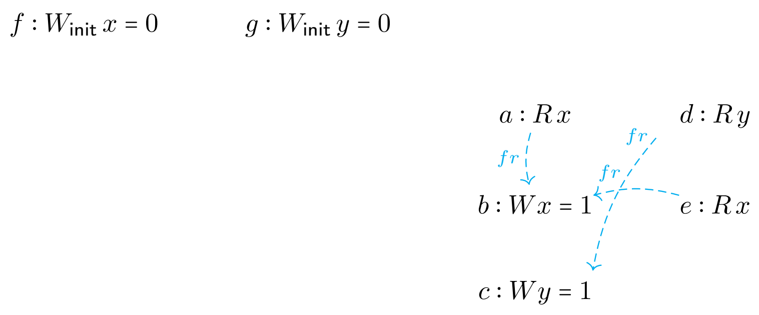


Figure 4.10: $M(\mathsf{fr})$.

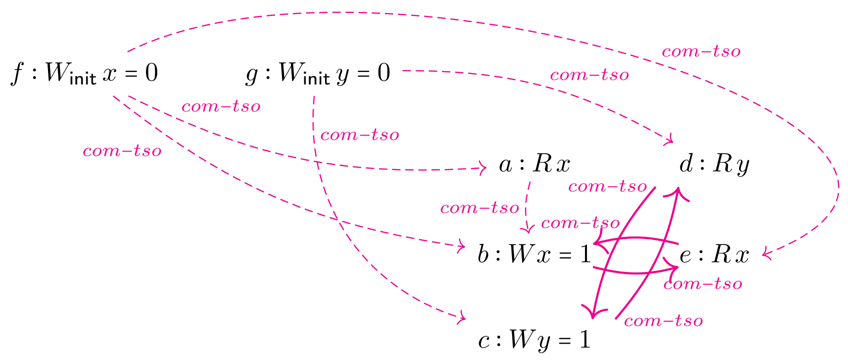


Figure 4.11: $M(\mathsf{com\text{-}tso})$ and $A(\mathsf{com\text{-}tso})$.

An edge of fr (from-read) must start in a read and end in a write to the same location where the value that was observed by the read is overwritten. The may set of $\mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$ (see Fig. 4.10) is computed from the may sets of rf and co.

The union $M(\mathsf{rfe}) \cup M(\mathsf{co}) \cup M(\mathsf{fr})$ forms $M(\mathsf{com\text{-}tso})$ as shown in Fig. 4.11.

The may set of implied is empty since no write is followed by a read in the program order. Finally, Fig. 4.12 shows the may set of ghb-tso, which is a union of $M(\mathsf{po\text{-}tso})$, $M(\mathsf{com\text{-}tso})$ and $M(\mathsf{implied})$.

4.7.2 Active Sets

First, the active set of edges required for the constraint acyclic ghb-tso is computed. The constraint forbids cycles of ghb-tso. The may set of this relation could contain three simple cycles: $(c \to d \to c)$, $(b \to e \to b)$ and $(b \to c \to d \to e \to b)$.

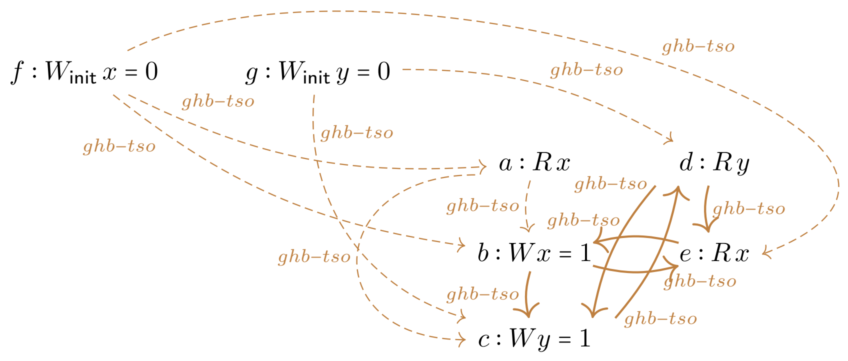


Figure 4.12: $M(\mathsf{ghb\text{-}tso})$ and $A(\mathsf{ghb\text{-}tso})$.

Since ghb-tso is not used in any other constraint, all edges which are not included into these cycles can be safely omitted. The resulting set $A(\mathsf{ghb\text{-}tso})$ is in Fig. 4.12. Recall that active sets are depicted by the solid edges.

Then, the algorithm updates the active sets of the child relations of ghb-tso, namely po-tso, com-tso, and implied. Since ghb-tso is a union, for each child relation r holds

$$A(\mathsf{r}) = A(\mathsf{r}) \cup (A(\mathsf{ghb\text{-}tso}) \cap M(\mathsf{r})).$$

For po-tso, the constraint requires encoding of the two edges out of four that could occur in one of the cycles: $\mathsf{po\text{-}tso}(b, c)$ and $\mathsf{po\text{-}tso}(d, e)$. The active set of po-tso is given in Fig. 4.7. Similarly, for com-tso, the constraint only requires four (out of the original ten) edges as shown in Fig. 4.11.

Note that we only examined one of the three constraints of TSO. For some relations, the active set are obtained by combining the requirements generated by different constraints. For example, it follows from $A(\mathsf{po\text{-}tso})$ that $A(\mathsf{po})$ includes edges (b, c) and (d, e) required by the constraint acyclic ghb-tso. However, the evaluation of acyclic $\mathsf{po\text{-}loc} \cup \mathsf{com}$ additionally requires edge (a, b) to be in $A(\mathsf{po})$.

Table 4.1 shows the number of pairs in the may and active sets of the encoded relations after evaluating all three constraints. Note that since there are seven events, $7^2 = 49$ edges per relation are possible. Out of those, we never need to encode more than six. Note, that the communication relations are not only used in the constraints, they capture the data flow as well. It follows that we always encode their entire may sets. These relations include rf, co and the internal relations (idd and addrDirect) which determine the values of registers.

Relation	May	Active
po	4	3
loc	18	1
po-loc	1	1
co	2	2
rf	6	6
rf^{-1}	6	3
fr	3	3
co ∪ fr	5	3
com	11	6
po-loc∪ com	11	6
rmw	1	0
ext	34	2
fre	2	0
coe	2	0
fre; coe	0	0
rmw ∩ (fre; coe)	0	0
W × R	12	0
po ∖ (W×R)	4	2
mfence	0	0
po-tso	4	2
rfe	5	2
com-tso	10	4
po-tso ∪ com-tso	13	6
po∩(W× R)	0	0
M × A	14	0
A × M	14	0
(M×A)∪ (A × M)	24	0
implied	0	0
ghb-tso	13	6
idd	2	0
addrDirect	0	0
Total	221	58

Table 4.1: The number of edges in May and Active sets.

4.8 Alias Analysis

Recall that we model the heap by encoding a new memory location for each variable and a set of locations for each memory allocation of an array. Each location x has an address $ad(x)$. A memory event has two parameters: the value it writes or reads and the address it accesses. The addresses of locations and events are encoded as Integer variables and their values are chosen by the solver, they are not known at the time of the encoding.

We minimize the encoding size, by over-approximating the domains of relations (see Section 4.6). To compute precise may sets for base relations, we rely on an alias analysis to determine which events could access the same location. We stress the importance of this analysis for our reduction approach. The relations loc, co, and rf are frequently used as building blocks of composite relations. Excessive may sets will therefore negatively affect the over-approximations of virtually all relations in a memory model, and hence keep the overall encoding unnecessarily large.

The reader may argue that assembly will not contain a command $r = \&x$, and in turn C does not have registers. This is one of the challenges in developing a widely-applicable tool: one needs an expressive instruction set. We think of registers as local and locations as global variables.

We offer two versions of an Andersen-style may points-to analysis [26], a control-flow insensitive (CFI) analysis and a control-flow sensitive (CFS) variant. The CFS analysis determines, for every event e in the unrolled program and every register r that occurs in e, a set $PTS(e, r)$ of addresses that the register *may point to* in the event e. Similarly, there are sets $PTS(x)$ that a location x may point to. The fact that the latter set is not decorated by an event means the points-to information for locations is still control-flow insensitive. This is unavoidable, since the data flow of the shared memory locations depends on the memory model. The CFI analysis computes the same points-to set of a register $PTS(e, r)$ for all events e. We elaborate on this below.

Given the points-to sets, the set Alias used in the encoding is not represented explicitly, instead we use an intersection $\mathit{PTS}(e_1, r_1) \cap \mathit{PTS}(e_2, r_2)$ to check whether two registers may alias. For a memory event e, let $ad(e)$ denote the expression that describes the address accessed by the event. In the relation analysis, we want to know which writes might access the same locations as a given memory event e:

$$\mathit{Alias}(e) := \{e' \in \mathbb{E} \mid \mathit{PTS}(e, ad(e)) \cap \mathit{PTS}(e', ad(e')) \neq \varnothing\}.$$

From this, we obtain the set Alias of all event pairs that may alias:

$$\mathit{Alias} := \{(e, e') \mid e \in \mathbb{M} \wedge e' \in \mathit{Alias}(e)\}$$

Technically, the addresses $\mathit{PTS}(e, r)$ are represented symbolically, in terms of their locations. Consider a location x at address 42 and an event e executing $r = \&x$. Then we obtain the information $\&x \in \mathit{PTS}(e, r)$, the address of x is contained in the points-to set. Since the SMT solver chooses the addresses of the locations, address 42 is runtime information that we cannot use for the analysis. In short, we work over the powerset lattice of locations.

We define a system of inequalities in which the $\mathit{PTS}(e, r)$ and $\mathit{PTS}(x)$ act as variables. The analysis information of interest is the least solution to the system. The analysis is triggered by the computation of the may sets.

Although the Andersen analysis is well-understood, the fact that we take the events into account makes it interesting. We present the system of inequalities for the CFS analysis:

$$\begin{array}{rl}
\mathit{PTS}(e, r') \supseteq \mathit{PTS}(e', r') & \text{if } e' \text{ is } r' = \mathit{exp} \text{ and } (e', e) \in M(\mathsf{idd}) \cup M(\mathsf{addrDirect}) \\
\mathit{PTS}(e, r) \supseteq \mathit{PTS}(e, r') & \text{if } e \text{ is } r = r' \\
\mathit{PTS}(e, r) \supseteq \{\&x\} & \text{if } e \text{ is } r = \&x \\
\mathit{PTS}(y) \supseteq \{\&x\} & \text{if } e \text{ is } {}^*y = \&x \\
\mathit{PTS}(e, r) \supseteq \mathit{PTS}(y) & \text{if } e \text{ is } r = {}^*x \text{ and } \&y \in \mathit{PTS}(x) \\
\mathit{PTS}(y) \supseteq \mathit{PTS}(e, r) & \text{if } e \text{ is } {}^*x = r \text{ and } \&y \in \mathit{PTS}(x) \\
\mathit{PTS}(e, r) \supseteq \top & \text{if } e \text{ is } r = \mathit{exp} \text{ with } \mathit{exp} \text{ different from above} \\
\mathit{PTS}(x) \supseteq \top & \text{if } e \text{ is } x = \mathit{exp} \text{ with } \mathit{exp} \text{ different from above.}
\end{array}$$

The first constraint propagates points-to information from one event-register pair to the next. Event e' writes a value defined by the expression exp to the register r' of interest. The information is propagated from e' to the event e that accesses this value. The second inequality forwards points-to information from the right-hand side of the assignment to the left. The third and fourth are the base case of Andersen's analysis. The fifth inequality is for loads that assign to r the value of a location x. As this value may again be the address of a location, we inherit the points-to information. Note that the inequality is required to hold for any location y that x may point to. The fact that the locations do not occur syntactically in the assignment is the reason we treat locations control-flow insensitively. The case of stores is similar. The last two inequalities let a register or location point to anything (top in the lattice) if it is assigned a non-trivial expression (pointer arithmetic). Note, that the fourth and eight constraint only occur in initial events.

We also provide a CFI variant of the alias analysis. Here, the points-to set of a register is the same for all events. Internally, we only need to compute one points-to set per register and require fewer constraints. This leads to a speed up in the alias analysis compared to the CFS version. Semantically, the CFI version only requires us to change the first constraint as follows:

$$PTS(e, r') \supseteq PTS(e', r') \qquad\qquad \text{if } e, e' \in \mathbb{E}.$$

Andersen-style analyses are usually control-flow insensitive, meaning they compute the points-to information independently of the order of instructions. Since we take the events into account, our CFS analysis actually behaves control-flow sensitively on registers (first constraint). This is sound since data and addr dependencies are preserved in cat models.

But what if the given memory model does not preserve the flow of control? To ensure soundness of the analysis, we require the given memory model to preserve the following relations:

$$\mathsf{valDirect}^+ \qquad \mathsf{valDirect}^+;\mathsf{addr} \qquad \mathsf{rf}\ .$$

This is the case for all memory models we encountered so far. Should it not be the case, the user can only rely on a CFI analysis. The fact that static analyses are only allowed to make assumptions about the semantics of a program that are valid wrt. the memory model was observed before and discussed in [12]. Parameterizing the static analysis by the relations of the memory model, however, is a new idea.

Since the encoding is the bottle-neck of the tool, the alias analysis provides a speed-up. This holds particularly for larger programs with more locations. We note that there is a trade-off between the precision of the alias analysis, and hence the size of the resulting encoding, and the time required for the alias analysis. Our CFS analysis is more precise than a standard CFI Anderson analysis. It also has a larger number of variables and constraints and thus takes longer to compute. It can add precision if the control flow restricts the points-to sets further. This is illustrated by the following example:

$$r = \&y;\ *x = r;\ r = \&x.$$

Here, x may point towards y but not itself. The CFS analysis computes the precise set $PTS(x) = \{y\}$ because it differentiates between the points to sets of r in different events. The standard Anderson analysis returns $PTS(x) = \{x, y\}$ since it only computes one points-to set of the register r. . If a thread reads a register containing a pointer and then later in the program order stores other addresses in the same register, then the CFS analysis may add precision.

We gain less precision when there are concurrent accesses to a location. This is the case if we expand our example:

$$r = \&y;\ *x = r;\ r = \&x;\ *z = r \quad || \quad q = *z;\ *x = q.$$

Here, the pointer to y is passed to a second thread via z and then written in x. The CFS analysis computes $PTS(x) = \{x, y\}$ here as well. This may be precise depending on the memory model. We use the control flow insensitive version as our default alias analysis. We recommend the control flow sensitive analysis for larger pointer programs with little concurrency.

4.9 Experiments

We compare Dartagnan against several memory model-aware tools. Herd7 [20] is a tool designed for litmus tests (small programs). It takes `cat` files as an input (and thus supports all memory models used in this section). It enumerates all candidate executions and then filters those accepted by the memory model. Nidhugg [1, 5] performs stateless model checking. It supports TSO, POWER and a simplified version of ARM. CBMC [19] is a Bounded Model Checker with an encoding similar to ours, but it cannot handle recursive definitions efficiently and only supports TSO. We also compare it against the Dartagnan FMCAD-18 version [140, 138], which has no relation analysis and thus also no alias analysis. Finally, we evaluate the impact of the alias analysis on the execution time. The experiments were conducted on an Intel Core i7 CPU with 15,5 GB of RAM.

Benchmarks. For CBMC, Nidhugg, and the FMCAD-18 Dartagnan, we evaluate the performance on 7 classic mutual exclusion algorithms. The benchmarks are executed on TSO (all tools) and a subset of ARMv7 (only Nidhugg and Dartagnan). The results on POWER are very similar to the ones on ARM and thus omitted. We excluded Herd7 from this experiment since it did not scale even for small unrolling bounds [138]. We set a 5 min timeout for Parker, Dekker, and Peterson as this is sufficient to show the trends in the run-times, and a 30 min timeout for the remaining benchmarks.

To compare against Herd7, and to evaluate the impact of the alias analysis, we run 4751 Linux kernel litmus tests[1] with kernel primitives such as RCU on the Linux kernel model. We set a 30 minutes timeout.

4.9.1 Evaluation

The times for CBMC, Nidhugg-ARM, and the FMCAD-2018 version of Dartagnan grow exponentially for Parker. The growth in CBMC and FMCAD-2018 Dartagnan is due to the explosion of the encoding with the program size. For the latter, the solver runs out of memory with unrolling bounds 20 (TSO) and 10 (ARM). For Nidhugg-ARM, the issue is that Parker contains much more executions in ARM than TSO. Thus the tool explores many unnecessary executions. The verification times for Nidhugg-TSO and the current version of

[1]The benchmarks contain all tests from `https://github.com/paulmckrcu/litmus` without Linux spinlock primitives as they are not yet supported by Dartagnan.

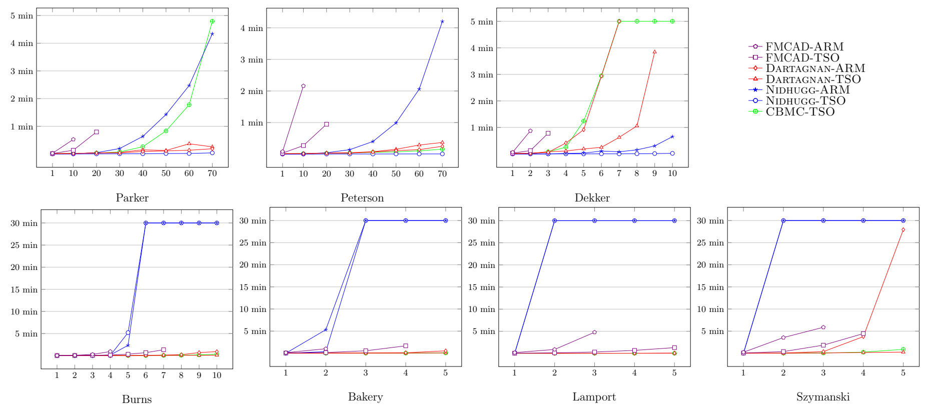


Figure 4.13: Impact of the unrolling bound (x-axis) on the verification time (y-axis).

Dartagnan grow linearly. The latter is due to the optimization by the relation analysis (see Section 4.6) keeping the encoding small. This can result in a speed-up of more than two orders of magnitude in certain cases. For Peterson, the results are similar except for CBMC, which matches Dartagnan's performance.

For Dekker, Nidhugg outperforms both CBMC and Dartagnan. This is because the number of executions grows slowly compared to the explosion of the number of instructions. The executions in both memory models coincide, making the performance on ARM comparable to TSO for Nidhugg. The difference is due to the optimal exploration in TSO, but not in ARM. Our optimizations have some impact on the performance (see FMCAD-2018 vs. Dartagnan), but the encoding size still grows faster than the number of executions.

The benchmarks Burns, Bakery, and Lamport demonstrate the opposite trend: the number of executions grows much faster than the size of the encoding (even without optimizations). In this case, both CBMC and Dartagnan outperform Nidhugg. Notice that for Burns, Nidhugg performs better on ARM than on TSO with unrolling bound 5. This is counter-intuitive since one expects more executions on ARM. However, the number of executions coincide, but the exploration time is higher in TSO (different search algorithm). For Szymanski, similar results hold except for Dartagnan-ARM where the encoding grows exponentially despite the optimizations.

Fig. 4.14 (left) shows the verification times for the current version of Dartagnan with and without alias analysis. Using an alias analysis results in a speed-up of more than two orders of magnitude in benchmarks with several threads accessing up to 18 locations. Fig. 4.14 (right) compares the performance of Dartagnan against Herd7. We used the Knaster-Tarski encoding and alias analysis since they yield the best performance. Herd7 outperforms Dartagnan on small test instances (less than 1 second execution time). This is due to the JVM startup time and preprocessing costs of Dartagnan. However, on large benchmarks, Herd7 times out while Dartagnan takes less than 10 secs.

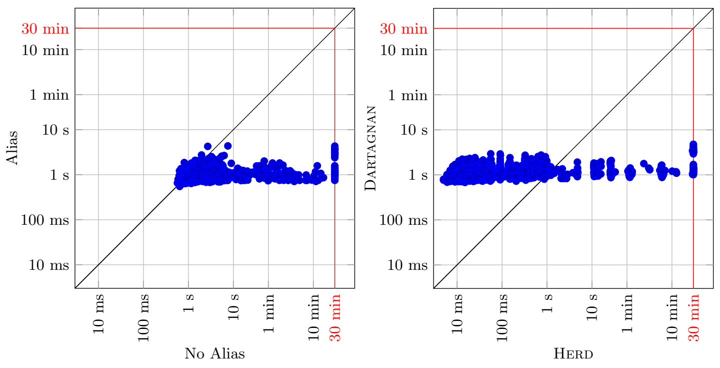


Figure 4.14: Execution times (logarithmic scale) on Linux kernel litmus tests: impact of alias analysis (left) and comparison against Herd7 (right).

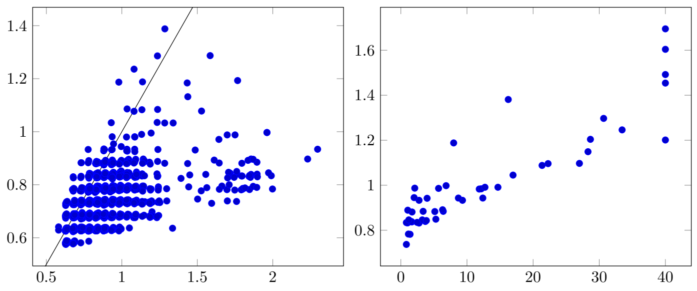


Figure 4.15: Execution times (in secs) of litmus tests with and without alias analysis. The first graph contains the auto litmus tests, the second graph contains manufactured litmus tests to explore the possible speed-up for 1-50 variables.

In Fig. 4.15 (left), we compared the run-time of the tool to its runtime without the pointer analysis on over 2500 litmus tests. On the right, we constructed very small litmus tests that simply pass a pointer between 1-50 threads using 1-50 locations. This shows the best case improvements provided by the alias analysis. We observed that the speed up provided by the alias analysis increases significantly with number of locations.

$Consistent_{SC+atomicity}$

$\mathsf{com} = \mathsf{co} \cup \mathsf{fr} \cup \mathsf{rf} \qquad \mathsf{come} = \mathsf{com} \cap \mathsf{ext}$

$\textcircled{21}\ \mathit{acyclic}(\mathsf{po} \cup \mathsf{com}) \qquad \textcircled{21}\ \mathit{empty}(\mathsf{rmw} \cap (\mathsf{come}; (\mathsf{po} \cup \mathsf{com})^*; \mathsf{come}))$

Figure 4.16: Memory model used for Sv-Comp'20.

4.9.2 Sv-Comp'20 Benchmarks

As there are no competitions on software verification against weak memory models, we entered Dartagnan in the 9th Competition on Software Verification (Sv-Comp 2020) in the category ConcurrencySafety [139]. Our intention was to demonstrate that a universal tool with memory models as input can be competitive even compared against tools specifically developed towards SC.

On the Sv-Comp'20 benchmarks[2], Dartagnan reports only one incorrect result, being beaten in that aspect only by CPAchecker, Divine, Lazy-CSeq and Yogar-CBMC; three of them category winners. The incorrect result has been identified as a (now fixed) bug in the Boogie parser.

We use sequential consistency (SC) with additional support for atomic blocks as the input memory model. Fig. 4.16 shows the memory model used for Sv-Comp'20. To support atomic blocks, Dartagnan adds a specific edge (rmw) for every pair of events between `VERIFIER_atomic_begin()` and its matching `VERIFIER_atomic_end()` or in a `VERIFIER_atomic_` function. We encode atomicity for SC as the empty intersection of rwm and paths starting and ending with an external communication (i.e. between different threads). This means once an atomic block starts, external communications with the block are forbidden until all events in the block have been executed.

On this benchmark, Dartagnan's performance cannot quite match that of other verifiers that were developed specifically towards SC. That is as expected since we have traded off some efficiency for universal memory model support. The main strength of Dartagnan is its fully configurable memory model.

Dartagnan performs particularly poor on benchmarks with big atomic blocks. This is the case for most of the verification tasks in the `pthread-wmm` group which represent 83% of the ConcurrencySafety category. The problem is that Dartagnan adds rmw edges for all pairs in the block. This results in a large encoding (even using relation analysis) and highly impacts its performance.

[2]The results are at `https://sv-comp.sosy-lab.org/2020/results/results-verified/`.

4.10 Conclusion and Outlook

Recall that in Section 3.5, we presented a reduction of testing to SAT. We adapted and expanded on this construction in order to handle more expressive programs instead of the simple tests and the more flexible `cat` language instead of serial view based memory models.

We presented DARTAGNAN, a modular Bounded Model Checker for concurrent programs. The tool is universal: it takes any memory model defined in the `cat` language as input and can check bounded reachability of a program under it. Our method reduces reachability of an acyclic program to satisfiability of a SMT formula using novel encoding techniques.

We have performed experiments to compare our tool to several memory model-aware tools, and find that it scales very well. It competes with tools that are tailored towards specific models. We compared it to the universal tool HERD7 and found it to be faster by an order of magnitude for large programs.

The encoding techniques include compact representations of relations by predicates as well as approximations of operations that are not precise but still sound. We perform a novel static analysis to further reduce the size of the encoding. These techniques can be re-used by any bounded model checking technique reasoning about complex memory models. We will apply them to the portability problem in the next chapter.

We are continuously researching further techniques to reduce the size of the encoding. Currently, we are investigating how symmetry breaking constraints can be utilized towards that goal.

Chapter 5

Program Portability

Equipped with the encoding for reachability, we now tackle the portability problem. We study the problem of porting performance-critical code among hardware architectures. Porting code from one architecture to another is a routine task in system development. Given that no functionality has to be added, porting is rarely considered interesting from a programming point of view. At the same time, porting is non-trivial as the hardware influences both the semantics and the compilation of the code in subtle ways. The unfortunate combination of being routine and yet subtle makes porting prone to mistakes. This is particularly true for performance-critical code that interacts closely with the execution environment. Such code often has data races and thus it exposes the programmer to the details of the underlying hardware. When the architecture is changed, the code may have to be adapted to the primitives and semantics of the target hardware.

Our contribution is the new (and to the best of our knowledge first) tool PORTHOS to fight porting bugs. PORTHOS supports programmers in porting code from one architecture (for which it has been thoroughly validated) to another. It takes as input a concurrent program, and two `cat` memory models (see Section 2.3): one of the source architecture for which the code has been developed and one of the target architecture to which the code is to be ported. PORTHOS automatically checks whether the behavior of the code on the target architecture is the same as on on the source platform. This guarantees that correctness of the program in terms of safety properties (in particular properties like mutual exclusion) carry over to the targeted hardware, and the program remains correct after porting.

The first question is this: What do we understand as program behavior? Is it the possible executions or the reachable states? This leads us to define two notions of portability:

- A program is *trace portable* if the same executions are possible under both memory models.
- It is *state portable* if the same states are reachable under both models.

It is easy to see that trace portability implies state portability. Naturally, the problem is not decidable in the general case for both notions [145]. We again restrict ourselves to bounded unrollings.

Trace Portability We adapt our encoding techniques from the reachability problem in order to construct a reduction of bounded trace portability to UNSAT. There are three new problems that occur when encoding trace portability.

(i) The formulation of trace portability involves an alternation of quantifiers. A program is trace portable if *for every* execution under one model, *there is* an equivalent execution under the other.

(ii) We have to find an exact encoding of the least fixpoint of recursively defined relations. This is necessary since our approach looks for executions that violate a constraint, which is not monotone.

(iii) High-level code may be compiled into different low-level code depending on the architecture.

The first problem is to encode the quantifier alternation underlying the definition of trace portability. Our approach is to adapt the encoding of reachability in order to find equivalent executions that violate trace portability. We look for porting bugs. A porting bug is an execution that is consistent with the target but inconsistent with the source memory model. We capture this alternation with a single existential query. Is there an execution consistent with the source memory model and an equivalent execution not consistent with the target memory model? Consistency is specified in terms of acyclicity (and irreflexivity) of relations. Hence, an execution is inconsistent if a derived relation of the (source) memory model contains a cycle (or is not irreflexive or empty). The naive idea would be to model cyclicity by unsatisfiability. Instead, we reduce cyclicity to satisfiability by introducing auxiliary variables that guess a cycle.

Concerning the second problem, we propose an efficient encoding for derived relations that are defined as least fixed points. Such least fixed points are prominently used in the Power memory model [24] and their computation was identified as a key problem in [169]. To quote the authors *[...] the proper fixpoint construction [...] is much more expensive than a fixed unrolling.* We show that, with our encoding, this is not the case. A naive approach would implement the Kleene iteration in SAT by introducing copies of the variables for each iteration step, resulting in a very large encoding. We show how to employ SAT + integer difference logic [59] to compactly encode the Kleene iteration process. Notably, every bounded model checking technique reasoning about complex memory models defined in CAT will face the problem of dealing with recursive definitions and can make use of our technique to solve it efficiently.

The third problem is that the same high-level program is compiled to different assembly programs depending on the source and the target architectures. Even the number of registers and the semantics of the synchronisation primitives provided by those architectures usually differ. Consider the program from Fig. 5.1, written in C++11 and compiled to x86 and Power. The observation is this. Even if the assembly programs differ, one can map every assembly memory access to the corresponding read or write operation in the high-level code. In the example, clearly "`MOV [y],$1`" and "`stw r1,y`" correspond to "`y.store(memory_order_relaxed, 1)`". This allows us to relate low-level and high-level executions and to compare executions of both assembly programs by checking if they map to the same high-level execution. With this observation, our analysis can be extended by translating an input program into two corresponding assembly programs and making explicit the relation among the low-level and high-level executions.

State Portability A developer is usually not interested in the exact shape of an execution, only in its outcome. This means state portability is the more practically relevant problem. We still find it useful to study trace portability. We compare the two notions of portability and discover the following:

(a) Algorithmically, the complexity of state portability is beyond SAT.

(b) Empirically, there is little difference between trace portability and state portability.

We use these insight to construct an efficient heuristic for the state portability problem. It uses the solution for trace portability to find useful candidate states that might disprove state portablity. The state portability problem asks whether no new (potentially unsafe) states are introduced and whether all reachable states can still be reached (no functionality has been lost). PORTHOS checks this equivalence for two memory models that are given as modules. If equivalence does not hold, it reports a counterexample execution leading to a reachable state allowed by only one architecture. Operating System kernel developers and library designers can use equivalence checks to understand whether a programming idiom, an algorithm, or a data structure that is known to work under one memory model can also be used under another.

State equivalence is checked in the form of inclusions in both directions. Due to the alternation of quantifiers, inclusion is notoriously difficult to check [169]: For every state reachable in one architecture we have to find an execution in the other that leads to the same state. We have tackled the trace inclusion problem and showed that it is easier to solve (in terms of complexity) than state inclusion. Despite that theoretical result, the notions often coincide. We show that state inclusion can be solved practically using a guided search strategy.

The idea is to be pessimistic and try to disprove the inclusion. The analysis looks for a state that is reachable in one but not in the other model (like the one in the **IRIW** example). To find states that may disprove the inclusion, PORTHOS invokes an oracle function. This oracle proposes a series of candidate states for which it gives the following guarantees.

(Progress) The series does not contain the same state twice.

(Soundness) If the oracle has no more states to propose, then the inclusion indeed holds.

Progress is certainly desirable and soundness is indispensable for verification. The interesting thing to note is that soundness leaves it to the oracle to terminate early if it finds out, by whatever reasoning, that the inclusion holds.

We require the implementation of an oracle in SMT which makes progress, is sound, and may terminate early. The idea is to look for so-called *delta executions*: Executions that are inconsistent with one memory model but consistent with the other. Finding a delta execution corresponds to solving the trace inclusion problem. A state resulting from a delta execution is clearly a candidate to violate the inclusion. Moreover, if there are no more states resulting from delta executions, the oracle can conclude that the inclusion holds — even if not all reachable states have been considered.

Outline: This chapter is structured as follows. First, we illustrate portability on an example. We present related work in Section 5.2. An overview of our solution for trace portability is in Section 5.3 In Section 5.4 we describe the design of PORTHOS followed by the encoding of trace portability. Section 5.6 contains a complexity study of the trace and state portability and shows the practical differences on our experiments. We give our method for solving state portability in Section 5.7 This is followed by our experimental results for state portability and finally our conclusion.

The results of this chapter have been published in [137] and [138]. In this work, the experiments for state portability have been updated to include relation analysis.

5.1 Portability Analysis on an Example

Consider program **IRIW** in Fig. 5.1, written in C++11 and using the atomic operator `memory_order_relaxed` which provides no guarantees on how memory accesses in different threads are ordered. When porting, the program is compiled to two different architectures. The corresponding low-level programs behave differently on x86 and on IBM's Power. Note that while the assembly programs are different, there are clearly corresponding memory instructions for each store and load of the high-level program. This means we can map an execution of the high-level program to equivalent executions of the assembly programs.

On TSO, the memory model implemented by x86, each thread has a store buffer of pending writes. A thread can see its own writes before they become visible to other threads (by reading them from its buffer), but once a write hits the memory it becomes visible to all other threads simultaneously: TSO is a multi-copy-atomic model [58]. Power on the other hand does not guarantee that writes become visible to all threads at the same point in time. Think of each thread as having its own copy of the memory. With these two architectures in mind, consider the execution in Fig. 5.1. Thread t_2 reads $x = 1, y = 0$ and thread t_3 reads $x = 0, y = 1$, indicated by the solid edges *rfe* and *rf*. Since under TSO

```
thread t0                             thread t1
y.store(memory_order_relaxed, 1)      x.store(memory_order_relaxed, 1)

thread t2                             thread t3
r1 = x.load(memory_order_relaxed);    r1 = y.load(memory_order_relaxed);
r2 = y.load(memory_order_relaxed)     r2 = x.load(memory_order_relaxed)
```

x86 Assembly

thread t_0	thread t_1	thread t_2	thread t_3
`MOV [y],$1`	`MOV [x],$1`	`MOV EAX,[x]`	`MOV EAX,[y]`
		`MOV EAX,[y]`	`MOV EAX,[x]`

Power Assembly

thread t_0	thread t_1	thread t_2	thread t_3
`li r1,1`	`li r1,1`	`lwz r1,x`	`lwz r1,y`
`stw r1,y`	`stw r1,x`	`lwz r3,y`	`lwz r3,x`

$$\begin{tikzcd}
Iy0 \arrow[r, "\mathit{co}"] \arrow[rrd, bend right, "\mathit{rf}"'] & Wy1 \arrow[rr, bend left, "\mathit{rfe}"] & Rx1 \arrow[d, dashed, "\mathit{po}"] & Ry1 \arrow[d, dashed, "\mathit{po}"'] & Wx1 \arrow[ll, bend right, "\mathit{rfe}"'] & Ix0 \arrow[l, "\mathit{co}"'] \arrow[lld, bend left, "\mathit{rf}"] \\
 & & Ry0 \arrow[ul, "\mathit{fr}"] & Rx0 \arrow[ur, "\mathit{fr}"'] & &
\end{tikzcd}$$

Figure 5.1: Portability of program **IRIW** from TSO to Power.

every execution has a unique global view of all operations, no interleaving allows both threads to read the above values of the variables. Under Power, this is possible. Our goal is to automatically detect such differences when porting a program from one architecture to another, here from TSO to Power.

The CAT formalisation of TSO is given in Fig. 2.8. Since we can disregard any constraints concerning atomicity and *rmw*, the model forbids executions forming a cycle over $\mathit{rfe} \cup \mathit{fr} \cup (\mathit{po} \setminus (\mathbb{W} \times \mathbb{R}))$. The red edges in Fig. 5.1 yield such a cycle; the execution is not consistent with TSO. Power further relaxes the program order (Fig. 2.10), the dotted lines are no longer considered for cycles and thus the execution is consistent. Hence, **IRIW** has executions consistent with Power but not with TSO. It is not trace portable. Since the execution given in Fig. 5.1 is the only one that leads to this state, it is not state portable either.

5.2 Related Work

A problem less general than portability is robustness against TSO which ensures trace inclusion between SC and TSO, thus under-approximating state inclusion. Verifying robustness against TSO is proven to be **PSPACE**-complete [40] and solved by TRENCHER [41]. It reduces the problem to state reachability under the SC semantics in an instrumented program and a minimal number of fences is synthesized to enforce SC behaviors. MEMORAX shares this functionality and is complete for reachability under TSO [2, 3, 4].

A step further, one can enforce robustness not only against TSO, but also against weaker memory models. The OFFENCE tool [21] does this, but can only analyze litmus test and is limited to restoring SC. CHECKFENCE enforces robustness against an over-approximation of the behavior of some popular memory architectures [45]. Checking the existence of critical cycles (i.e. portability bugs) on complex programs has been tackled in [18], where such cycles are broken by automatically introducing fences. The cost of different types of fences is considered and the task is encoded as an optimization problem. Bouajjani et al. study robustness in transactional storage system and databases, in particular against Snapshot Isolation [32] and Transactional Causal Consistency [33]. For many memory models, robustness is found to be **PSPACE**-complete [69, 68].

The MUSKETEER tool analyzes C programs and has shown to scale up to programs with thousands lines of code, but the implementation is also restricted to the case were the source model is SC. ROCKER is a prototype tool which checks robustness against the release/aquire semantics of C11 [109] The tool is noteworthy for comparing the reachable states against SC instead of the executions themselves. Fence insertion can also be used to guarantee safety properties (rather than restoring SC behaviors). The FENDER and DFENCE tools [108, 118] can do this for real-world C code, but they are restricted to TSO, PSO, and RMO.

5.3 Solving Trace Portability

Trace Portability from $\mathcal{M}_S$ to $\mathcal{M}_T$ requires that the same executions are consistent with $\mathcal{M}_T$ and with $\mathcal{M}_S$. Recall that $cons_{\mathcal{M}}(P)$ is the set of executions of program P consistent with $\mathcal{M}$. Given a program P and two memory models $\mathcal{M}_S$ and $\mathcal{M}_T$, our goal is to find an execution X which is consistent with the target ($X \in cons_{\mathcal{M}_T}(P)$) but not with the source ($X \notin cons_{\mathcal{M}_S}(P)$) or vice versa. In such a case P is not portable from $\mathcal{M}_S$ to $\mathcal{M}_T$.

Definition 11 (Trace Portability). *Let $\mathcal{M}_S$, $\mathcal{M}_T$ be two memory models. A program P is* trace portable from $\mathcal{M}_S$ to $\mathcal{M}_T$ *if $cons_{\mathcal{M}_T}(P) = cons_{\mathcal{M}_S}(P)$.*

We introduce a bounded analysis for trace portability implemented in PORTHOS

https://github.com/hernanponcedeleon/Dat3M.

First, the program is unrolled up to a user-specified bound. Within this bound, Porthos is guaranteed to find all portability bugs. It will neither see bugs beyond the bound nor will it be able to prove a cyclic program portable. The unrolled program, together with the CAT models, is transformed into an SMT formula where satisfying assignments correspond to bugs.

A bug is an execution consistent with the target memory model $\mathcal{M}_T$ but inconsistent with the source $\mathcal{M}_S$ or vice versa. We express this combination of consistency and inconsistency with only one existential quantification. The key observation is that the derived relations, which may differ in $\mathcal{M}_T$ and $\mathcal{M}_S$, are fully defined by the execution. Hence, by guessing an execution we also obtain the derived relations (there is nothing more to guess). Checking consistency for $\mathcal{M}_T$ is then an acyclicity, irreflexivity or emptiness constraint on the derived relations that immediately yields an SMT query. Notably, inconsistency for $\mathcal{M}_S$ requires cyclicity. The trick is to explicitly guess the cycle. We introduce Boolean variables for every event and every edge that could be part of the cycle.

In Fig. 5.1, if $Rx1$ is in the cycle, indicated by the variable $\mathtt{C}_r(Rx1)$ being set, then there should be one incoming and one outgoing edge also in the cycle. Besides the incoming edge shown in the graph, $Rx1$ could read from the initial value $Ix0$. Since there are two possible incoming edges but only one outgoing edge, we obtain $\mathtt{C}_r(Rx1) \Rightarrow ((\mathtt{C}_{rfe}(Wx1, Rx1) \vee \mathtt{C}_{rf}(Ix0, Rx1)) \wedge \mathtt{C}_{po}(Rx1, Ry0))$. If a relation is in the cycle, then both end-points should also be part of the cycle and the relation should belong to the execution: $\mathtt{C}_{po}(Rx1, Ry0) \Rightarrow (\mathtt{C}_r(Rx1) \wedge \mathtt{C}_r(Ry0) \wedge \mathtt{po}(Rx1, Ry0))$. Finally, at least one event has to be part of the cycle: $\mathtt{C}_r(Ix0) \vee \mathtt{C}_r(Wx1) \vee \mathtt{C}_r(Rx1) \vee \mathtt{C}_r(Rx0) \vee \mathtt{C}_r(Iy0) \vee \mathtt{C}_r(Wy1) \vee \mathtt{C}_r(Ry1) \vee \mathtt{C}_r(Ry0)$. The execution in Fig. 5.1 contains the relations marked in red and forms a cycle which violates Constraint (17) in TSO. The assignment respects the constraints of Power (Fig. 2.10), showing the existence of a portability bug in **IRIW** from TSO to Power.

The other challenge is to capture relations that are defined recursively. The Kleene iteration process [159] starts with the empty relation and repeatedly adds pairs of events according to the recursive definitions. We encode this into (quantifier-free) integer difference logic [59]. For every recursive relation $\mathtt{r}$ and every pair of events (e_1, e_2), we introduce an integer variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ representing the iteration step in which the pair entered the value of $\mathtt{r}$. A Kleene iteration then corresponds to a total ordering on these integer variables. Crucially, we only have one Boolean variable $\mathtt{r}(e_1, e_2)$ per pair rather than one per iteration step. We illustrate the encoding on a simplified version of the preserved program order for Power defined as $ppo := ii \cup ic$ (cf. Fig. 2.10 for the full definition). The relation is derived from the mutually recursive relations $ii := data \cup ic$ and $ic := ctrl \cup ii$, where $data$ and $ctrl$ represent data and control dependencies. Call $Rx1$ and $Ry0$ respectively e_1 and e_2. The encoding consists of the Knaster-Tarski encoding (see Section 4.5.7) together with

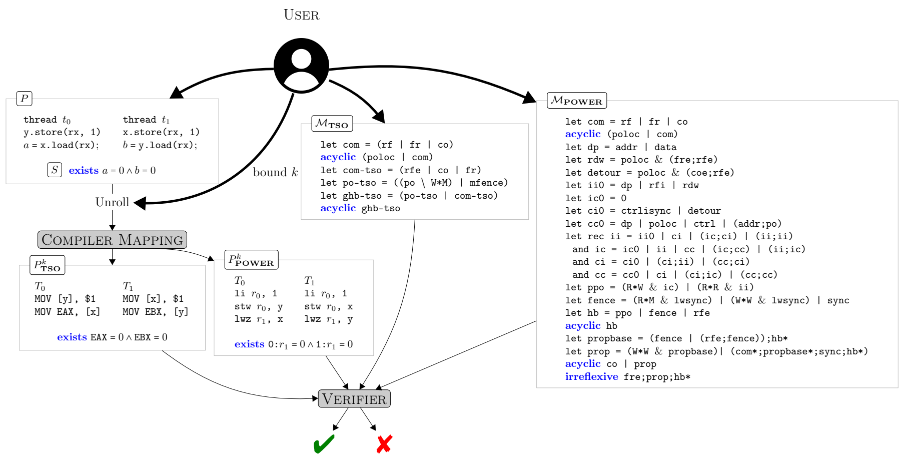
$$\mathtt{ii}(e_1, e_2) \Leftrightarrow (\mathtt{data}(e_1, e_2) \wedge (\Phi^{\mathtt{ii}}_{e_1,e_2} > \Phi^{\mathtt{data}}_{e_1,e_2})) \vee (\mathtt{ic}(e_1, e_2) \wedge (\Phi^{\mathtt{ii}}_{e_1,e_2} > \Phi^{\mathtt{ic}}_{e_1,e_2}))$$

$$\mathtt{ic}(e_1, e_2) \Leftrightarrow (\mathtt{ctrl}(e_1, e_2) \wedge (\Phi^{\mathtt{ic}}_{e_1,e_2} > \Phi^{\mathtt{ctrl}}_{e_1,e_2})) \vee (\mathtt{ii}(e_1, e_2) \wedge (\Phi^{\mathtt{ic}}_{e_1,e_2} > \Phi^{\mathtt{ii}}_{e_1,e_2})).$$

The pair (e_1, e_2) that belongs to relation *data* in step $\Phi^{\texttt{data}}_{e_1,e_2}$ of the Kleene iteration can be added to relation *ii* at a later step $\Phi^{\texttt{ii}}_{e_1,e_2} > \Phi^{\texttt{data}}_{e_1,e_2}$. As $ii := data \cup ic$, the disjunction allows us to also add the elements of *ic* to *ii*. Since *data* and *ctrl* are empty for **IRIW**, the relations *ii* and *ic* have to be identical. Identical non-empty relations will not yield a solution: the integer variables cannot satisfy $(\Phi^{\texttt{ii}}_{e_1,e_2} > \Phi^{\texttt{ic}}_{e_1,e_2})$ and $(\Phi^{\texttt{ic}}_{e_1,e_2} > \Phi^{\texttt{ii}}_{e_1,e_2})$ at the same time. Hence, the only satisfying assignment is the one where both *ii* and *ic* are the empty relation, which implies that *ppo* is empty. This is consistent with the preserved program order of Power for **IRIW**.

Note that without the additional Knaster-Tarski encoding, the solver may choose the "wrong" integer variables that do not satisfy the corresponding restraint. This means it could compute a smaller relation than the least fixed point. We fix this by additionally ensuring that the relation is a fixpoint using the Knaster-Tarski encoding.

5.4 User Interface



```
thread t0              thread t1
y.store(rx, 1)         x.store(rx, 1)
a = x.load(rx);        b = y.load(rx);

exists a = 0 ∧ b = 0
```

```
T0                     T1
MOV [y], $1            MOV [x], $1
MOV EAX, [x]           MOV EBX, [y]

exists EAX = 0 ∧ EBX = 0
```

```
T0                     T1
li r0, 1               li r0, 1
stw r0, y              stw r0, x
lwz r1, x              lwz r1, y

exists 0:r1 = 0 ∧ 1:r1 = 0
```

```
let com = (rf | fr | co)
acyclic (poloc | com)
let com-tso = (rfe | co | fr)
let po-tso = ((po \ W*M) | mfence)
let ghb-tso = (po-tso | com-tso)
acyclic ghb-tso
```

```
let com = rf | fr | co
acyclic (poloc | com)
let dp = addr | data
let rdw = poloc & (fre;rfe)
let detour = poloc & (coe;rfe)
let ii0 = dp | rfi | rdw
let ic0 = 0
let ci0 = ctrlisync | detour
let cc0 = dp | poloc | ctrl | (addr;po)
let rec ii = ii0 | ci | (ic;ci) | (ii;ii)
 and ic = ic0 | ii | cc | (ic;cc) | (ii;ic)
 and ci = ci0 | (ci;ii) | (cc;ci)
 and cc = cc0 | ci | (ci;ic) | (cc;cc)
let ppo = (R*W & ic) | (R*R & ii)
let fence = (R*M & lwsync) | (W*W & lwsync) | sync
let hb = ppo | fence | rfe
acyclic hb
let propbase = (fence | (rfe;fence));hb*
let prop = (W*W & propbase)| (com*;propbase*;sync;hb*)
acyclic co | prop
irreflexive fre;prop;hb*
```

Figure 5.2: PORTHOS from the user's perspective.

We present PORTHOS from a user's perspective. Fig. 5.2 illustrates the artifacts that are required for or produced by the tool for checking reachability. For checking *program portability*, PORTHOS expects as input the program P, two memory models $\mathcal{M}_S$ and $\mathcal{M}_T$, and an unrolling bound k. It recursively unwinds all loops in P up to the bound k. The unwound program is then mapped to the assembly dialects of the source and target architectures In Fig. 5.2, program P is a simplified mutex algorithm which is mapped to x86 (P^k_{TSO}) and to POWER (P^k_{POWER}) using the compiler mappings in Table 2.1

The resulting acyclic and annotated assembly programs are handed over to the analysis. The verifier checks whether the program is portable. For trace portability, this means two SMT queries, each checking trace inclusion in one direction. For state portability, this is implemented in a guided search using multiple SMT queries (see Section 5.7). The tool checks whether the reachable executions (resp. states) under $\mathcal{M}_T$ are the same as under $\mathcal{M}_S$. In Fig. 5.2, we use memory models for POWER and TSO where the support for read-modify-write instructions is omitted for brevity. This analysis is performed on the unrolled and mapped programs. In Fig. 5.2, we check if the consistent executions (or reachable states for state portability) by P^k_{POWER} under POWER are the same as the ones by P^k_{TSO} under TSO (which is the case). We process equivalence queries with two inclusion checks. These checks compare either the executions or the reachable states of two assembly versions of the same program running under different memory models.

5.5 Encoding

Our method for solving trace portability finds non-portable executions as satisfying assignments of an SMT formula. We solve trace inclusion in both directions. We check whether there is an execution of the unrolled compiled program P^k_T that is consistent with $\mathcal{M}_T$ such that the equivalent execution of P^k_S is not consistent with $\mathcal{M}_S$.

The solver guesses an execution of an unrolled assembly program and ensures that the corresponding execution of the other unrolled assembly program is equivalent. Recall that an execution is uniquely represented by the set of executed events and the relations *rf* and *co*. All other relations are derived from these guesses, the source code of the program, and the memory models in question. We encode the derived relations of both memory models. Finally, we encode the `cat` constraints on these relations in such a way that the guessed execution is allowed by $\mathcal{M}_T$ (all the constraints stated for $\mathcal{M}_T$ hold) while the same execution is not allowed by $\mathcal{M}_S$ (at least one of the constraints of $\mathcal{M}_S$ is violated). The full SMT formula is of the form

$$\begin{aligned} &\mathit{equiv}(P^k_S, P^k_T) \wedge \phi_{CF}(P^k_T) \wedge \phi_{DF}(P^k_T) \wedge \phi_{\mathcal{M}_T}(P^k_T) \\ &\qquad\qquad \wedge \phi_{CF}(P^k_S) \wedge \phi_{DF}(P^k_S) \wedge \phi_{\neg\mathcal{M}_S}(P^k_S). \end{aligned}$$

Here, ϕ_{CF} and ϕ_{DF} encode the control flow and data flow of the executions, $\phi_{\mathcal{M}_T}$ encodes the derived relations and constraints of $\mathcal{M}_T$, and $\phi_{\neg\mathcal{M}_S}$ encodes the derived relations of $\mathcal{M}_S$ together with a violation of at least one constraint of $\mathcal{M}_S$. We encode the condition that the executions of the two assembly programs are equivalent in $\mathit{equiv}(P^k_S, P^k_T)$.

We encode trace inclusion in the other direction simply by interchanging $\mathcal{M}_T$ and $\mathcal{M}_S$. If both formulas are unsatisfiable, then the program is trace portable.

The control-flow and data-flow encodings as well as the encoding of $\mathcal{M}_T$ are as in Section 4.5. The remainder of the section focuses on the encoding of the execution equivalence and on the encoding of inconsistency with the source memory model.

5.5.1 Encoding Common Executions

We look for an execution consistent with $\mathcal{M}_T$ and inconsistent with $\mathcal{M}_S$. However, we execute two different assembly programs P^k_S and P^k_T. This means we need a way to compare their executions. Intuitively, two executions are equivalent if they represent the same execution of the program P^k. Since the compilation scheme of Table 2.1 implements each atomic memory operation using a single low-level memory access, a one-to-one mapping $\pi : \mathbb{E}_T \to \mathbb{E}_S$ between the events of P^k_S and P^k_T can be defined for each assembly mapping. Given two events e_S and e_T representing instructions accessing memory in the assembly programs, $\pi(e_T) = e_S$ holds if they both represent the same high-level instruction. Note that such a mapping π can always be defined as long as the compiler implements atomic memory operations with a single memory access. Recall that an execution consists of the set of executed events, the relation rf, and the coherence relation co. The variable $\mathsf{exec}_T(e)$ indicates whether event e of P^k_T is executed. The following encoding relates the executions of both assembly programs:

$$\begin{aligned} \mathit{equiv}(\mathbb{E}_T, \mathbb{E}_S) = & \bigwedge_{e \in \mathbb{E}_T} \mathsf{exec}_T(e) \Leftrightarrow \mathsf{exec}_S(\pi(e)) \\ \wedge & \bigwedge_{e_1, e_2 \in \mathbb{E}_T} \mathtt{rf}(e_1, e_2) \Leftrightarrow \mathtt{rf}(\pi(e_1), \pi(e_2)) \\ \wedge & \bigwedge_{e_1, e_2 \in \mathbb{E}_T} \mathtt{co}(e_1, e_2) \Leftrightarrow \mathtt{co}(\pi(e_1), \pi(e_2)). \end{aligned}$$

5.5.2 Encoding Least Fixed Points

Recall that some relations (e.g. *ii* and *ic* of Power) can be defined mutually recursively and that the Knaster-Tarski encoding does not necessarily compute the least fixed point but some larger relation. While this does not destroy correctness of reachability, Dartagnan also allows for the drawing of execution graphs. Here, we want the correct relations with the least fixed point semantics.

More importantly, this is necessary for trace portability where we need to use the least fixed point semantics for cyclicity constraints: If we were to use the Knaster-Tarski encoding for trace portability, a larger fixpoint could be chosen by the solver with more edges and thus new cycles could be created.

A classical approach for solving such equations is the Kleene fixpoint iteration. The algorithm starts from the empty relations as initial approximation and on each round computes a new approximation until the (least) fixed point is reached. Such an iterative algorithm can be easily encoded into SAT. The problem of such an encoding is the potentially large number of iterations needed, and thus the resulting formula size can grow to be quite large.

A better way to encode this is an approach that has been already used in earlier work on encoding mutually recursive monotone equation systems with nested least and greatest fixpoints [94]. We use an extension of SAT with integer difference logic (IDL), a logic that is still NP complete and supported by the SMT solver natively. A SAT encoding is also possible but incurs an overhead in the encoding size: if the SMT encoding is of size $O(n)$, the SAT encoding is of size $O(n \log n)$ [94]. We chose IDL since our experiments showed the encoding to be the most time consuming of the tasks.

The encoding is also closely related to [98, 131] which encode logic programming under the stable model semantics. They also use a least fixed point semantics for recursive definitions encoded into integer difference logic and SAT. While the basic encoding idea is not new, the application of the approach to encoding axiomatic memory models is novel, and can be of interest to other researchers for efficiently encoding memory model semantics using an SMT solver.

The basic idea is to guess a certificate per tuple that consists of the number of the iteration in which the tuple would be added to the relation in the Kleene iteration. For this we use additional integer variables and enforce that they locally follow the propagations made by the fixed point iteration algorithm. Thus, for any pair of events $e_1, e_2 \in \mathbb{E}$ and relation $r \subseteq \mathbb{E} \times \mathbb{E}$ we introduce an integer variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ representing the round in which $\mathtt{r}(e_1, e_2)$ would be set by the Kleene iteration algorithm. Using these new variables we guess the execution of the Kleene fixed point iteration algorithm, and then locally check that every guess that was made is also a valid propagation of the fixed point iteration algorithm. For a simple example on how the encoding for the union of relations needs to be modified to also handle recursive definitions, consider a definition where $r_1 := r_2 \cup r_3$ and $r_2 := r_1 \cup r_4$. The encoding is as follows

$$\begin{aligned}
\mathtt{r}_1(e_1,e_2) &\Leftrightarrow (\mathtt{r}_2(e_1,e_2) \wedge (\Phi^{\mathtt{r}_1}_{e_1,e_2} > \Phi^{\mathtt{r}_2}_{e_1,e_2})) \vee (\mathtt{r}_3(e_1,e_2) \wedge (\Phi^{\mathtt{r}_1}_{e_1,e_2} > \Phi^{\mathtt{r}_3}_{e_1,e_2})) \\
\mathtt{r}_2(e_1,e_2) &\Leftrightarrow (\mathtt{r}_1(e_1,e_2) \wedge (\Phi^{\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2})) \vee (\mathtt{r}_4(e_1,e_2) \wedge (\Phi^{\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_4}_{e_1,e_2})).
\end{aligned}$$

Assume a pair (e_1, e_2) is added to r_1 by the Kleene iteration in step $\Phi^{\mathtt{r}_1}_{e_1,e_2}$. It comes from either r_2 or r_3. If it came from r_2 then it is of course also in r_2 and it was added to r_2 in an earlier iteration $\Phi^{\mathtt{r}_2}_{e_1,e_2}$ and thus $(\Phi^{\mathtt{r}_1}_{e_1,e_2} > \Phi^{\mathtt{r}_2}_{e_1,e_2})$. It is similar if it came from r_3. The largest satisfying assignment for the encoding is one where both r_1 and r_2 are the union of r_3 and r_4.

Note that this only encodes *at most* the least fixpoint: Assume a pair (e_1, e_2) is in r_2. A satisfying assignment could also set a value for $\Phi^{\mathtt{r}_1}_{e_1,e_2}$ that is smaller than $\Phi^{\mathtt{r}_2}_{e_1,e_2}$ and thus not add the pair to r_1. We require the additional restriction that the relations are fixpoints:

$$\begin{aligned}
\mathtt{r}_1(e_1,e_2) &\Leftrightarrow \mathtt{r}_2(e_1,e_2) \vee \mathtt{r}_3(e_1,e_2) \\
\mathtt{r}_2(e_1,e_2) &\Leftrightarrow \mathtt{r}_1(e_1,e_2) \vee \mathtt{r}_4(e_1,e_2).
\end{aligned}$$

This combination ensures that a satisfying assignment is at most the least fixpoint and that it is a fixpoint. Thus it is exactly the least fixpoint. We use the Knaster-Tarski encoding from Fig. 4.3 to ensure any relation is a fixpoint.

For a relation r, we define its set of subrelations. For a basic relation, the set is empty. For a relation that consists of a binary operator on relations r_1 and r_2, the set consists of r_1, r_2 and all their subrelations. For a relation that consists of a unary operator on a relation r_1, it consists of r_1 and the subrelations of r_1. For r_1^* the subrelations additionally contain r_1^+ and its subrelations. For r_1^+, the subrelations additionally contain $r_1^+ ; r_1^+$ and its subrelations.

$$
\begin{aligned}
\mathtt{r}_1{\cup}\mathtt{r}_2(e_1,e_2) &\Leftrightarrow \mathtt{r}_1(e_1,e_2) \wedge (\Phi^{\mathtt{r}_1\cup\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2}) \\
&\quad \vee (\mathtt{r}_2(e_1,e_2) \wedge (\Phi^{\mathtt{r}_1\cup\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_2}_{e_1,e_2})) \\
\mathtt{r}_1{\cap}\mathtt{r}_2(e_1,e_2) &\Leftrightarrow \mathtt{r}_1(e_1,e_2) \wedge (\Phi^{\mathtt{r}_1\cap\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2}) \\
&\quad \wedge (\mathtt{r}_2(e_1,e_2) \wedge (\Phi^{\mathtt{r}_1\cap\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_2}_{e_1,e_2})) \\
\mathtt{r}_1{\setminus}\mathtt{r}_2(e_1,e_2) &\Leftrightarrow \mathtt{r}_1(e_1,e_2) \wedge (\Phi^{\mathtt{r}_1\setminus\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2}) \wedge \neg\mathtt{r}_2(e_1,e_2) \\
\mathtt{r}^{-1}(e_1,e_2) &\Leftrightarrow \mathtt{r}(e_2,e_1) \wedge (\Phi^{\mathtt{r}^{-1}}_{e_1,e_2} > \Phi^{\mathtt{r}}_{e_2,e_1}) \\
\mathtt{r}_1;\mathtt{r}_2(e_1,e_2) &\Leftrightarrow \bigvee_{e_3\in\mathbb{E}} (\mathtt{r}_1(e_1,e_3) \wedge (\Phi^{\mathtt{r}_1;\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_3}) \\
&\quad \wedge \mathtt{r}_2(e_3,e_2) \wedge (\Phi^{\mathtt{r}_1;\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_2}_{e_3,e_2})) \\
\mathtt{r}^*(e_1,e_2) &\Leftrightarrow (\mathtt{r}^+(e_1,e_2) \wedge (\Phi^{\mathtt{r}^*}_{e_1,e_2} > \Phi^{\mathtt{r}^+}_{e_1,e_2})) \vee (e_1 = e_2) \\
r^+(e_1,e_2) &\Leftrightarrow \mathtt{r}(e_1,e_2) \wedge (\Phi^{\mathtt{r}^+}_{e_1,e_2} > \Phi^{\mathtt{r}}_{e_1,e_2}) \\
&\quad \vee (r^+;r^+(e_1,e_2) \wedge (\Phi^{\mathtt{r}^+}_{e_1,e_2} > \Phi^{\mathtt{r}^+;\mathtt{r}^+}_{e_1,e_2}))
\end{aligned}
$$

Figure 5.3: Least fixpoint encoding of operators.

If a relation r is contained in its subrelations, then the relation is recursive and thus there might be multiple fixpoints. In this case, we add the corresponding encoding from Fig. 5.3 to ensure we specify at most the least fixpoint. Note that the marked inequalities in Fig. 5.3 can be omitted if the corresponding relation r_1 or r_2 is not the cause of the recursiveness. The terms marked thus are only necessary if the relation is a subrelation of r_1. They ensure that the corresponding pair was added to r_1 in an earlier iteration. The terms marked like this are only necessary if the relation is a subrelation of r_2. If it is neither a subrelation of r_1 or r_2, then the relation is not mutually recursive to another relation and the encoding in Fig. 4.3 is sufficient.

Theorem 11. *For any recursively defined relation r holds that the IDL-encoding of Kleene has a satisfying assignment and for any satisfying assignment holds $r(e_1,e_2) = true$ iff $(e_1,e_2) \in r$.*

The formal proof of Theorem 11 is given in Section B.1) of the appendix.

5.5.3 Encoding Inconsistency

For the source architecture, we encode inconsistency in $\phi_{\neg\mathcal{M}_S}(P^k_S)$. This means that one of the derived relations does not fulfil its constraints. On the top level this can be encoded as a simple disjunction over all the constraints of the source memory model $\mathcal{M}_S$, forcing at least one of the irreflexivity or acyclicity constraints to be violated. Recall that it is sufficient to consider the active $A(r)$ set of a relation r (see Section 4.6). Encoding violations of irreflexive and emptiness constraints is straightforward:

$$
\begin{aligned}
non-irreflexive(r) &\Leftrightarrow \bigvee_{(e,e)\in A(r)} \mathtt{r}(e,e) \\
non-empty(r) &\Leftrightarrow \bigvee_{(e_1,e_2)\in A(r)} \mathtt{r}(e_1,e_2).
\end{aligned}
$$

What remains to be encoded is $\mathit{cyclic}(r)$, which requires the relation r to be cyclic. Here, we give an encoding that uses only Boolean variables. We add Boolean variables $\mathtt{C}_r(e)$ and $\mathtt{C}_\mathtt{r}(e_1, e_2)$, which guess the edges and nodes constituting the cycle. Given an active set $A(r)$, we denote the set of events that occur in the event pairs given by $A(r)$ as $\mathbb{E}_{A(r)}$ We ensure that for every event in the cycle, there should be at least one incoming edge and at least one outgoing edge that are also in the cycle:

$$c_n = \bigwedge_{e_1 \in \mathbb{E}_{A(r)}} (\mathtt{C}_r(e_1) \Rightarrow (\bigvee_{(e_2,e_1)\in A(r)} \mathtt{C}_\mathrm{r}(e_2, e_1) \wedge \bigvee_{(e_1,e_2)\in A(r)} \mathtt{C}_\mathrm{r}(e_1, e_2))).$$

If an edge is guessed to be in a cycle, the edge must belong to relation r, and both events must also be guessed to be on the cycle:

$$c_e = \bigwedge_{(e_1,e_2)\in A(r)} (\mathtt{C}_\mathtt{r}(e_1, e_2) \Rightarrow (\mathtt{r}(e_1, e_2) \wedge \mathtt{C}_r(e_1) \wedge \mathtt{C}_r(e_2))).$$

A cycle exists, if these formulas hold and there is an event in the cycle:

$$\mathit{cyclic}(r) \Leftrightarrow (c_e \wedge c_n \wedge \bigvee_{e \in \mathbb{E}_{A(r)}} \mathtt{C}_r(e)).$$

5.6 Trace vs State Portability

One motivation to check portability is to make sure that safety properties of $\mathcal{M}_S$ carry over to $\mathcal{M}_T$. Safety properties only depend on the values that can be computed, not on the actual executions. Therefore, we now study a more liberal notion of so-called *state portability*: $\mathcal{M}_T$ may admit different executions as long as they do not compute different states. Admitting more executions means we require less synchronization (fences) to consider a ported program correct, and thus state portability promises more efficient code. The notion has been used in [108].

The main finding in this section is negative: a polynomial encoding of state portability to SMT does not exist (unless the polynomial hierarchy collapses). Phrased differently, state portability does not admit an efficient bounded analysis (like our method for trace portability). We remind the reader that we restrict our input to acyclic programs.

Fortunately, our experiments indicate that new executions often compute new states. This means trace portability is not only a sufficient condition for state portability but, in practice, the two are almost equivalent. Combined with the better algorithmics of portability, we see a good motivation to use solutions for trace portability in order to approximate state portability.

Recall that the set of executions of a program P consistent with $\mathcal{M}$ is denoted by $\mathit{cons}_\mathcal{M}(P)$ and that an execution X computes the state given by $\mathit{state}(X)$ (see Section 2.2). We lift this definition to sets. Given a set of executions $\mathbb{X}$, we denote the states that are computed by the executions in $\mathbb{X}$ as $\mathit{state}(\mathbb{X})$. The relationship between the portability notions is as in Lemma 7.

Definition 12 (State Portability)**.** *Let $\mathcal{M}_S$, $\mathcal{M}_T$ be memory models. Program P is* state portable from $\mathcal{M}_S$ to $\mathcal{M}_T$ *if state*$(cons_{\mathcal{M}_T}(P))$ = *state*$(cons_{\mathcal{M}_S}(P))$.

Lemma 7. *(1) Trace portability implies state portability. (2) State portability does not imply trace portability.*

To illustrate Lemma 7.(2) on an example, consider a variant of **IRIW** (Fig. 5.1) where all written values are 0. The program is trivially state portable from Power to TSO since it can only compute one state where all values are 0. It is not trace portable since it allows the same executions as **IRIW**.

We turn to the hardness argumentation. To check state portability, every $\mathcal{M}_T$-computable state seems to need a formula checking whether some $\mathcal{M}_S$-consistent execution computes it and vice versa. The result would be an exponential blow-up or a quantified Boolean formula, which is not practical. But can this exponential blow-up or quantification be avoided by some clever encoding trick? The answer is no! Theorem 12 shows that state portability is in a higher class of the polynomial hierarchy than portability. So state portability is indeed harder to check than portability.

The polynomial hierarchy [158] contains complexity classes between NP and PSPACE. Each class is represented by the problem of checking validity of a Boolean formula with a fixed number of quantifier alternations. We need here the classes co-NP = $\Pi_1^P \subseteq \Pi_2^P$. The *tautology problem* (validity of a closed Boolean formula with a universal quantifier $\forall x_1 \dots x_n : \psi$) is a Π_1^P-complete problem. The higher class Π_2^P allows for a second quantifier: validity of a formula $(\forall x_1 \dots x_n \exists y_1 \dots y_n : \psi)$ is a Π_2^P-complete problem. Theorem 12 refers to a class of common memory models that we define in a moment. Moreover, we assume that the given pair of memory models $\mathcal{M}_S$ and $\mathcal{M}_T$ is *non-trivial* in the sense that $cons_{\mathcal{M}_T}(P) = cons_{\mathcal{M}_S}(P)$ fails for some program, and similar for state portability.

Theorem 12. *Let $\mathcal{M}_S, \mathcal{M}_T$ be a non-trivial pair of common memory models. (1) Trace Portability from $\mathcal{M}_S$ to $\mathcal{M}_T$ is Π_1^P-complete. (2) State portability is Π_2^P-complete.*

The formal proof of this theorem is rather extensive with multiple technical lemmas. It is given in Section B.2 and B.3 of the appendix. We now present the idea behind the proof.

The first part says that our trace portability analysis is optimal. We focus on this lower bound to give a taste of the argumentation: given a non-trivial pair of memory models, we know there is a program that is not trace portable. Crucially, we do not know the program but give a construction that works for any program. By Theorem 12.(2), state portability cannot be solved efficiently. The proof of this is along similar lines but more involved. Our proof technique requires the memory models to satisfy some properties common to real-world architectures.

Definition 13. *We call an memory model* common[1] *if*

(i) the inverse operator is only used in the definition of fr,

(ii) the constructs int, ext, loc, and $\langle set \rangle \times \langle set \rangle$ *are only used to restrict other relations (in a conjunction),*

(iii) the emptiness constraint is only used in Constraint $\textcircled{18}$*,*

(iv) it satisfies uniproc (Constraint $\textcircled{16}$*) , and*

(v) every program is portable from this model to SC.

We explain the definition. We call the *fr* together with the base relations except for those listed in (*ii*) *non-trivial* relations. When formulating a memory model, one typically forbids well-chosen cycles of non-trivial relations.

To this end, derived relations are introduced that capture the paths of interest, and acyclicity constraints are imposed on the derived relations. The operators inverse and constructions involving $\langle set \rangle \times \langle set \rangle$, *int*, *ext*, or *loc* may do the opposite, they add relations that do not correspond to paths of non-trivial relations. Besides stating what is common in memory models, properties *(i)* to *(iii)* help us compose programs (cf. next paragraph). The uniprocessor correctness condition (uniproc) is a fundamental property of processor architectures [88]. Since the purpose of a memory model is to capture SC relaxations, we can assume a memory model to be weaker than SC. The memory model in this study satisfy these properties and, to the best of our knowledge, so do all memory architectures currently used. Properties *(iv)* and *(v)* guarantee that the program P_ψ given below is portable between any common memory models.

The crucial property of common memory models is the following. For every pair of events e_1, e_2 in a derived relation, (1) there are (potentially several) sequences of non-trivial relations that connect e_1 and e_2, and (2) the derived relation only depends on these sequences. The property ensures that if we append a program P' to a location-disjoint program P, there is no non-trivial edge from P' to P. This means consistency of composed executions is preserved.

It remains to prove Π_1^P-hardness of portability. We first introduce the program P_ψ that generates some assignment and checks if it satisfies the Boolean formula $\psi(x_1 \dots x_m)$ (over the variables $x_1 \dots x_m$). The program $P_\psi := t_1 \parallel t_2$ consists of the two threads t_1 and t_2 defined below. Note that we cannot directly write a constant i to a location, so we first assign i to register $r_{c,i}$.

thread t_1	*thread* t_2
$r_{c,0} \leftarrow 0;\ r_{c,1} \leftarrow 1;\ r_{c,2} \leftarrow 2$	$r_{c,1} \leftarrow 1;$
$x_1 := r_{c,0} \dots x_m := r_{c,0};$	$x_1 := r_{c,1} \dots x_m := r_{c,1};$
$r_1 \leftarrow x_1 \dots r_m \leftarrow x_m;$	
if $\psi(r_1 \dots r_m)$ **then**	
$\quad y := r_{c,2};$	
else $y := r_{c,1};$	

[1]Notice that all memory models considered in [24] and in this paper are common ones.

				Deadness	
	✗✗	✓✓	✗✓	✓✓	✗✓
SC-TSO	27	898	75	933	40
SC-PSO	27	777	196	836	137
SC-RMO	27	737	236	780	193
SC-Alpha	27	846	127	887	86
TSO-PSO	0	833	67	883	27
TSO-RMO	0	760	240	798	202
TSO-Alpha	0	877	133	912	88
PSO-RMO	0	831	169	844	156
PSO-Alpha	0	968	32	973	27
RMO-Alpha	0	999	1	999	1
Alpha-RMO	0	856	144	864	136
	0.98%	85.29%	13.73%	88.26%	10.73%
SC-Power	1477	898	52	936	14
TSO-Power	917	1132	378	1166	344
PSO-Power	502	1880	45	1892	33
RMO-Power	40	2227	160	2239	148
Alpha-Power	0	2427	0	2427	0
	24.20%	70.57%	5.23%	71.35%	4.45%

Table 5.1: Trace Portability vs. State Portability on litmus tests.

We reduce checking whether $\forall x_1 \dots x_m : \psi(x_1 \dots x_m)$ holds to trace portability of a program $P_{\forall\psi}$. The idea for $P_{\forall\psi}$ is this. First P_ψ is run, it guesses and evaluates an assignment for ψ. If ψ is not satisfied ($y = 1$), then some non-portable program P_{np} is executed. The program $P_{\forall\psi}$ is trace portable iff the non-portable part is never executed. This is the case iff ψ is satisfied by all assignments.

Let $\mathcal{M}_S$, $\mathcal{M}_T$ be common and non-trivial. By non-triviality, there is a program $P_{np} = t'_1 \parallel \cdots \parallel t'_k$ that is **n**ot trace **p**ortable from $\mathcal{M}_S$ to $\mathcal{M}_T$. We can assume P_{np} has no registers or locations in common with P_ψ. Program $P_{\forall\psi}$ prepends P_ψ to the first two threads of P_{np}. Once $y = 1$, P_{np} starts running. Formally, let t_1 and t_2 be the threads in P_ψ and let t_i be empty sequences for $3 \leq i \leq k$. We define $P_{\forall\psi} := t''_1 \parallel \cdots \parallel t''_k$ with $t''_i := t_i;\ r \leftarrow y;\ \textbf{if}(r = 1)\ \textbf{then}\ t'_i$.

We show that $P_{\forall\psi}$ is portable iff ψ is satisfied for every assignment by proving the following: if $P_{\forall\psi}$ is not portable then ψ has an unsatisfying assignment and vice versa. The key idea behind this proof is that properties *(i)*-*(iii)* ensure that no event of P_{np} is related to an event of P_ψ in an execution of $P_{\forall\psi}$. This means that any cycle of a relation occurs in P_{np}.

5.6.1 Experiments

We evaluate the practical difference between the two portability notions on benchmark programs on a wide range of well-known memory models. We employ Porthos to verify trace portability and Herd7 for state portability. For SC, TSO, PSO, RMO, and Alpha (henceforth called traditional architectures) we use the formalizations from [13] (see Section 2.3 for SC and TSO). We use the definition in Fig. 2.10 for Power. Note that we only port from a stronger to a weaker model so we only have to check for new executions or states in the target model.

$$\textcircled{22}\ \mathit{domain}(\mathit{ctrl}) \subseteq \mathit{range}(\mathit{rf})$$
$$\textcircled{23}\ \mathit{imm}(\mathit{co}); \mathit{imm}(\mathit{co}); \mathit{imm}(\mathit{co}^{-1}) \subseteq \mathit{rf}^{?}; (\mathit{po}; (\mathit{rf}^{-1})^{?})^{?}$$

$$\mathit{imm}(r) := r \setminus (r; r^{+})$$

Figure 5.4: Syntactic Deadness [169].

Our experiments contain two test suites: TS_1 contains 1000 randomly generated litmus tests in x86 assembly (to test traditional architectures) and TS_2 contains 2427 litmus tests in Power assembly taken from [123]. Each test contains between 2 and 4 threads and between 4 and 20 instructions. Table 5.1 (right) reports the number of non-portable (w.r.t. both definitions) litmus tests (✗✗), the number of trace portable and state portable litmus tests (✔✔) and the number of litmus tests that are not trace portable but are still state portable (✗✔). In the last case the new executions allowed by the target memory model do not result in new computable states of the program. We show that in many cases both notions of portability coincide. For TS_1 on traditional architectures, the amount of non state portable tests is very low (0.98%), while the non trace portability of the program does not generate a new computable state in 13.73% of the cases. For TS_2 from traditional architectures to Power, the number of non state portable litmus tests rises to 24.20%, while only in 5.24% of cases the two notions of portability do not match because the new executions do not result in a new computable state for the program.

In order to remove some executions that do not lead to new computable states, PORTHOS optionally supports the use of syntactic deadness which has been proposed in [169]. Dead executions are either consistent or lead to not computable states. Formally an execution X is dead if $X \notin \mathit{cons}_{\mathcal{M}}(P)$ implies that $\mathit{state}(X) \neq \mathit{state}(Y)$ for all $Y \in \mathit{cons}_{\mathcal{M}}(P)$. Instead of looking for any execution which is not consistent for the source architecture, we want to restrict the search to non-consistent and dead executions of $\mathcal{M}_S$. This is equivalent to checking state portability. As shown by Wickerson et al. [169], dead executions can be approximated with constraints $\textcircled{22}$ and $\textcircled{23}$ given in Fig. 5.4 where $r^{?}$ is the reflexive closure of r. These constraints can be easily encoded into SAT. Our tool has an implementation which rules out quite a few executions not computing new states. The last two columns of the table show that by restricting the search to (syntactic) dead executions, the ratio of litmus tests the tool reports as non portable, but are actually state portable is reduced to 10.73% for traditional architectures and to 4.44% for Power.

The experiments above show that in most of the cases both notions of portability coincide, specially when using dead executions. The cases where such differences are manifested is very low, specially when porting to Power. The results of a trace portability check for mutual exclusion methods under multiple memory models is given in Section B.4 of the appendix.

Algorithm 4: Incremental SMT Solving for State Inclusion

```
Input: Program $P$, memory models $\mathcal{M}_S, \mathcal{M}_T$, Int $k$
Result: Decides whether state inclusion holds.
1  $P^k \leftarrow \textsc{unroll}(P, k)$;
2  $P_S^k \leftarrow \textsc{Compile}(P^k, \mathcal{M}_S)$;
3  $P_T^k \leftarrow \textsc{Compile}(P^k, \mathcal{M}_T)$;
4  $\phi_{\text{RCH}} \leftarrow \phi_{CF}(P_S^k) \wedge \phi_{DF}(P_S^k) \wedge \phi_{\mathcal{M}_S}(P_S^k)$;
5  while $\textsc{Oracle}().\mathit{hasState}()$ do
6      $s \leftarrow \textsc{Oracle}().\mathit{getState}()$;
7      if $\phi_{\text{RCH}} \wedge \phi_s$ is UNSAT then
8          return false ;
9      end
10 end
11 return true
```

5.7 Solving State Portability

We show how to efficiently check state portability. We check *state inclusion* in both directions. The inclusion requires that for all states reachable in the target memory model $\mathcal{M}_T$ there has to be an execution in the source memory model $\mathcal{M}_S$ computing the same state. Such a $\forall\exists$-alternation of quantifiers is notoriously difficult to handle for verification tools [169]. A naive approach would iterate over all reachable states. We propose to use an oracle guiding the search by providing relevant candidate states. We present an implementation of the oracle that iterates over far fewer states but preserves completeness. The key observation is that new states always correspond to new executions. Therefore we only need to consider states coming from executions consistent with the target but inconsistent with the source memory model.

The main procedure is described by Algorithm 4. It takes as input a program, two memory models $\mathcal{M}_S, \mathcal{M}_T$[2], and a bound k. The program is first unrolled up to the bound k and converted to to the acyclic assembly programs P_S^k and P_T^k using the mappings from Table 2.1. The procedure might perform several reachability queries for $\mathcal{M}_S$. Therefore, we construct a formula defining its consistent executions in Line 4. The formulas ϕ_{CF}, ϕ_{DF} and $\phi_{\mathcal{M}_S}$ are given in Section 4.5.

The algorithm then enters a loop iterating over a sequence of states which can be thought of as candidates for violating inclusion. These candidate states are provided by an oracle, a black box providing two functions. Function *hasState()* returns a Boolean judging whether there is still a candidate state to consider. If so, function *getState()* provides the candidate. The oracle has to meet the following specification.

[2]The latter is needed to implement the oracle. However in Algorithm 4 we consider the oracle a black box object.

(O1) If *hasState()* returns false, then state inclusion holds.

(O2) If *hasState()* returns true, an invocation of *getState()* returns a state.

(O3) Function *getState()* never returns the same state twice.

(O4) Every state returned by *getState()* is reachable in $\mathcal{M}_T$.

When the oracle provides a new candidate, the algorithm checks whether it is reachable in $\mathcal{M}_S$. If the state is not reachable, state inclusion does not hold and the procedure returns false at Line 8. If it is reachable, the check is repeated with a different state. If every state provided by the oracle is reachable under $\mathcal{M}_S$, state inclusion holds by (O1) and the procedure returns true at Line 11.

A correct but naive implementation of an oracle would list all states reachable under $\mathcal{M}_T$. A more efficient exploration is guaranteed by the following idea.

5.7.1 An Oracle for Efficient Exploration

We present an oracle that lists good candidates likely to violate state inclusion. Moreover, the oracle may be able to guarantee state inclusion early. Finally, the computation of candidate states itself is based on SMT-solving and quite efficient. The idea is to find all executions consistent with $\mathcal{M}_T$ but not $\mathcal{M}_S$, and extract their reachable states. This guarantees (O1) and (O4): When *hasState()* returns false, all states that may violate inclusion have been considered and thus state inclusion holds. Our implementation encodes the oracle as the formula for trace inclusion (see Section 5.5):

$$\begin{aligned}\phi_{\text{ORA}} = \mathit{equiv}(P_S^k, P_T^k) \wedge \phi_{CF}(P_T^k) \wedge \phi_{DF}(P_T^k) \wedge \phi_{\mathcal{M}_T}(P_T^k) \\ \wedge \phi_{CF}(P_S^k) \wedge \phi_{DF}(P_S^k) \wedge \phi_{\neg\mathcal{M}_S}(P_S^k).\end{aligned}$$

Function *hasState()* denotes whether the formula ϕ_{ORA} is satisfiable. In this case, *getState()* extracts a state s from a satisfying assignment and returns it. This guarantees (O2). To ensure (O3), the same state is not returned twice, the formula is iteratively updated to $\phi_{\text{ORA}} \coloneqq \phi_{\text{ORA}} \wedge \neg\phi_s$.

5.8 Experiments

We implemented state portability in PORTHOS which uses Z3 [65] as the backend SMT solver. It allows us to define memory models in the `cat` language.

We compare PORTHOS against HERD7 [20], a memory model-aware tool. HERD7 is designed for litmus tests (small programs) and designed to test reachability. It allows to reason about one memory model at a time and therefore cannot directly be used to test state inclusion. However, HERD7 returns information about all final states. We check state inclusion with HERD7 by computing the reachable states separately for both models (i.e. we run the tool twice) and comparing them afterwards.

Benchmark	TSO ⊆ SC					
	SAT	Herd7	Porthos	Δ	It	s.u.
Parker	✔	**0.51**	0.91	3	1	0.56
Dekker	✔	T/O	**2.31**	-	4	>**1558.44**
Peterson	✔	16.87	**0.95**	12	1	**17.75**
Burns	✔	1059.89	**1.13**	53	2	**937.95**
Bakery	✔	T/O	**2.06**	-	2	>**1747.57**
Lamport	✔	T/O	**1.57**	-	1	>**2292.99**
Szymanski	✔	T/O	**8.31**	-	4	>**433.21**

Benchmark	POWER ⊆ TSO					
	SAT	Herd7	Porthos	Δ	It	s.u.
Parker	✔	**0.49**	0.97	3	1	0.50
Dekker	✘	T/O	**2.35**	-	3	>**1531.91**
Peterson	✘	16.54	**0.94**	0	0	**17.59**
Burns	✘	1035.61	**1.13**	18	2	**916.46**
Bakery	✘	T/O	**25.18**	-	89	>**142.97**
Lamport	✔	T/O	**25.22**	-	75	>**142.74**
Szymanski	✘	T/O	**69.72**	-	17	>**51.63**

Table 5.2: State inclusion of mutual exclusion algorithms.

Our benchmark suite consists of mutual exclusion algorithms. We unrolled loops twice ($k = 2$) which is sufficient to show that our approach scales better than the other tools for programs with several executions. Programs contain either two or three threads. However their size is reported in terms of the number of consistent executions since the performance of the tools strongly depends on this. The execution times are given in seconds. We set a timeout of 1800 secs for Porthos and 3600 secs for Herd7 since the tool is run twice. For entries marked as T/O, the timeout was reached.

We are checking the inclusions TSO ⊆ SC and POWER ⊆ TSO. Inclusion in the other direction (necessary for equivalence) holds by the definition of the memory models. E.g., every state reachable under TSO is also reachable under the weaker model POWER.

The results are given in Table 5.2. The SAT column reports whether a counterexample to inclusion was found (✔) or not (✘). When Herd7 returns a result, we report on the number of delta executions (Δ). This corresponds to an upper bound on the maximal number of iterations Porthos might perform. As it can be seen from Table 5.2, in general this number is several orders of magnitude smaller than the total number of executions. The cases reporting zero iterations correspond to the set of executions coinciding for both memory models. For most of the cases, Porthos is at least one order of magnitude faster than Herd7. For TSO, the speed-up (S.U. column) can be up-to three orders of magnitude.

To some extent, this speed up is due to the relation analysis. In Section B.5 of the appendix, we present experiments with a previous version of PORTHOS that does not utilize relation analysis. One can see that the difference in speed-up is significant. This shows that the relation analysis is very beneficial not only for BMC but for solving portability problems as well.

5.9 Conclusion and Outlook

We have presented PORTHOS, a modular Bounded Model Checker for concurrent programs. The tool can check bounded trace and state portability under any pair of memory models defined in the `cat` language. It is the first completely modular portability checker. Our method reduces trace inclusion to unsatisfiability of a SMT formula in SAT + integer difference logic using novel encoding techniques. To this effect, we propose efficient solutions for two crucial tasks: reasoning about two user-defined memory models with different compilation mappings at the same time and encoding mutually recursively defined relations (needed for e.g. Power) into SMT precisely.

Our experimental results suggest that trace portability largely coincides with the state-based notion of portability. The complexity results show that checking for state portability cannot be done with two SMT solver queries, unlike our approach to trace portability analysis. This leads us to the idea of using our method for trace portability as an oracle for solving state portability.

State equivalence is tested using a guided search. We propose to use an oracle to find relevant candidate states, and show how to implement an efficient oracle based on SMT queries generated by our solution for trace portability. We have performed experiments to compare PORTHOS to HERD7, and find it at least one order of magnitude faster for large programs.

We expanded our encoding for the bounded reachability problem and were able to use it to solve bounded trace portability even though intuitively it seemed to be a harder problem. We then manage to utilize this encoding in a guided search to solve the bounded state portability problem which we have shown to be higher in the polynomial hierarchy.

Our method for encoding not only consistency but also inconsistency of an execution is a key idea for our memory model synthesis. We will present this in the following chapter.

Modern compilers perform various optimizations when mapping high-level code to assembly instructions. We plan to investigate whether such compiler mappings can be extracted from the compilation process, at least approximately.

Chapter 6

Memory Model Synthesis

The previously presented tools DARTAGNAN and PORTHOS both rely on a definition of the system architecture in the `cat` language. They show that axiomatic memory models are well suited for verification purposes. However, for many architectures, the memory models are only informally defined or given as an operational model by the developer. Formulating an axiomatic model for an architecture is a difficult task. It requires an in-depth understanding of both the architecture and the semantics of axiomatic models. Mistakes in the memory model are often subtle and hard to detect.

New processor architectures and programming languages with their own weak memory semantics are constantly being developed [13, 22, 24, 30, 54, 110, 141, 117, 97, 90, 90, 9, 97, 93]. This leads to a growing need of verification methods under new memory architectures. We require techniques to construct new axiomatic memory models that capture these architectures.

We tackle this problem by developing methods to synthesize weak memory models automatically. We present ARAMIS, a memory model synthesis tool. The tool employs SMT-queries based on our previously presented formulas for reachability and portability. Synthesis tasks are known for having a large time complexity. Our encoding techniques presented in the last two chapters is used in a novel guided search inspired by program synthesis techniques in order to keep the synthesis times practical.

We want to synthesize `cat` memory models in a universal way. This means we require an input that describes the behavior of the architecture and can easily be obtained from any kind of model definition. Our input consists of litmus tests (small programs annotated with reachability assertion) and their possible outcomes. In particular, a set of litmus tests that succeed and a set that should fail under the model. Such tests are known to be useful when formulating a memory model by hand [148].

The possible results of a litmus test under an architecture can be obtained from any kind of architecture definition. They can be constructed manually from the description of the architecture, they can obtained by repeated executions of the litmus test under the architecture [23, 15, 148] or by a simulation of executions under the architecture [151, 153, 152].

Extensive libraries of litmus tests sufficient to describe the behavior of weak memory architectures reasonably well already exist.[121, 120] The purpose of litmus tests is usually to highlight the behavior of programs that depends on different non-trivial guarantees of known memory models. This means it is unlikely that very large libraries of litmus tests are required in order to describe an architecture sufficiently [122].

Given some litmus tests with their desired outcomes, Aramis attempts to generate a `cat` memory model under which the litmus tests behave correctly. The tool maintains a list of relations. It repeatedly expands the list by adding a relation and then attempts to combine constraints over the listed relations in order to form memory models until it has found a memory model that behaves correctly. We present two methods of expanding the list of relations.

One is to simply enumerate all possible relations. Here, the challenges are to keep redundancy to a minimum and to determine which relations are useful to a memory model. We keep down redundancy by eliminating semantically equivalent relations. A relation is useful if a constraint over it still allows the correct litmus tests to succeed and forbids at least one execution of a test that is supposed to fail. We show that these properties are partially inherited when an operator is applied on a relation. For instance if a relation r is acyclic, then the intersection of r with another relation is acyclic as well.

The other method is a directed search in the form of a Counter-Example Guided Inductive Synthesis [155, 154] (CEGIS). Here, we obtain relations from satisfying assignments of SMT-queries and add constraints to ensure certain executions are cyclic. We introduce template relations. They are derived relations whose syntax trees have a certain depth and where the operators and base relations are not known during the encoding. They are chosen by the SMT solver in such a way that they are useful.

Aramis checks whether constraints over the listed relations can be combined to a memory model with the correct behavior using a back-tracking search.

Outline: The remainder of the chapter is structured as follows. In Section 6.1, we present our synthesis technique which consists of two tasks: expand the list of relations and check whether a memory model can be constructed from it. The checking phase is given in Section 6.2 and Section 6.3 presents an enumerative method that implements the expansion phase. In Section 6.4, we introduce a CEGIS method for expansion. The following sections discuss how we handle recursively defined relation and the relation analysis. Section 6.7 shows how the synthesis can be aided with additional inputs. Section 6.8 discusses the experimental results. Program synthesis is compared to memory model synthesis in Section 6.9. The following section is an overview of related work. Finally, we give a conclusion.

The results of this chapter have not been published before.

6.1 Solving Memory Model Synthesis

If a program P annotated with a reachability condition S_P has an execution $X \in exec(P)$ that is consistent with a model $\mathcal{M}$ ($X \in cons_{\mathcal{M}}(P)$) and computes a state that satisfies S_P ($state(X) \vDash S_P$), then we say P *succeeds for* $\mathcal{M}$. Otherwise it fails. If a model $\mathcal{M}$ consists of a single acyclicity axiom ($\mathcal{M} = \{acyclic(r)\}$) of a relation r and P succeeds/fails for $\mathcal{M}$ then we say P succeeds/fails for r. Given a set of programs with reachability conditions $Prog$, if all $P \in Prog$ succeed for $\mathcal{M}$ then we say $Prog$ succeeds for $\mathcal{M}$. If none succeed, we say $Prog$ fails for $\mathcal{M}$. Our definitions for success of an execution are analogue.

Note that our syntax of assertions given in Section 2.1.1 includes the quantifiers *not-exists* and *forall*. The corresponding assertions hold only if certain executions do **not** exist. They are implemented in DARTAGNAN by checking whether the negation holds. This is the case if the corresponding SMT encoding is unsatisfiable. For the purpose of memory model synthesis, we are only interested in satisfiability checks and thus we treat those quantifiers as if they were negated.

The semantics of a system architecture $\mathcal{M}$ can be defined by the language $L(\mathcal{M})$ consisting of all annotated programs of which the reachability check succeeds under $\mathcal{M}$. Given a description $\mathcal{M}$ of an architecture, we want to synthesize a memory model $\mathcal{M}'$ in the CAT language such that $L(\mathcal{M}) = L(\mathcal{M}')$. However, the language $L(\mathcal{M})$ is infinite for any non-trivial architecture. In order to limit the size of the input to a finite description, we approximate $L(\mathcal{M})$ with two finite sets POS and NEG such that $\text{POS} \subseteq L(\mathcal{M})$ and $\text{NEG} \subseteq \overline{L(\mathcal{M})}$. This approximation induces equivalence classes of memory models. Given a set of programs PROGS, it holds $\mathcal{M}_1 \approx \mathcal{M}_2$ iff

$$\forall P \in \text{PROGS} : P \in L(\mathcal{M}_1) \Leftrightarrow P \in L(\mathcal{M}_2).$$

If a model $\mathcal{M}$ has the desired behavior, the goal of a synthesis algorithm is to find a member of the equivalence class $[\mathcal{M}]$ induced by $\text{POS} \cup \text{NEG}$. We say such a model is *correct*. Formally, we address the following *synthesis problem*:

Input: Two sets POS and NEG of programs annotated with reachability conditions

Problem: Find a memory model $\mathcal{M}$ such that:

- POS succeeds for $\mathcal{M}$.
- NEG fails for $\mathcal{M}$.

We introduce the prototype memory model synthesis tool ARAMIS[1]. A `cat` memory model may contain three different constraints: acyclicity, irreflexivity, and emptiness. Our first key insight is that we only need to consider the acyclicity constraint since it is sufficient to express the other constraints. Recall that $\mathbb{E}$ denotes the set of executed events and id the (reflexive) identity relation. It

[1] https://github.com/florianfurbach/Aramis

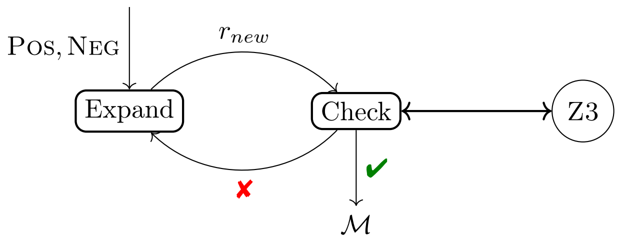


Figure 6.1: An overview of Aramis.

holds that *irreflexive*(r) is equivalent to *acyclic*$(r \cap \mathsf{id})$. Furthermore, *empty*(r) is equivalent to *acyclic*$(r; r^{-1})$ as well as *acyclic*$(r; [\mathbb{E} \times \mathbb{E}])$.

An overview of Aramis is given in Fig. 6.1. The main idea of our synthesis technique is to enumerate all relevant relations r that may occur in the model and consider the corresponding constraints *acyclic*(r). We try to combine these constraints in such a way that they form a correct memory model. This means we maintain a list Rels of relations annotated with additional information. The algorithm alternates between two tasks: expanding the list of relations and checking the current list for suitable relations to form a memory model. We can *expand* the list Rels by simply enumerating relations (Section 6.3) or we can perform a targeted search (Section 6.4).

The algorithm begins with performing an expansion. Once an expansion step is complete and a relation r_{new} was added to the list, the tool performs a *check* to determine whether r_{new} can be used to construct a correct memory model. This is done with a backtracking search. Starting with a memory model $\mathcal{M}$ that contains only the constraint *acyclic*(r_{new}), acyclicity constraints over relations are added until Neg fails for $\mathcal{M}$. If Pos succeeds for $\mathcal{M}$, then the memory model behaves correctly and the tool finishes the synthesis with output $\mathcal{M}$. If Pos does not succeed for $\mathcal{M}$, a backtracking step is performed. The search space can be reduced if the user provides additional data. If the check fails, another expansion is performed.

6.2 Checking

After the expansion has added a new relation r_{new} to the list Rels, the checking phase verifies whether a memory model can be constructed using r_{new} together with other listed relations that succeeds for Pos and fails for Neg. We present a backtracking search that attempts to build such a memory model by repeatedly adding constraints. We disregard recursively defined relations for now. We will discuss how to handle those in Section 6.5.

A memory model consists of a set of constraints. We want to find a set such that Pos succeeds for our synthesized memory model. This means we only need to consider relations for which Pos succeeds. However, it is important to note that even if Pos succeeds for some relations individually, it does not follow that

Pos succeeds for a model that consists of constraints over those relations. It is possible that the constraints invalidate different executions of a program in such a manner that put together, they invalidate all executions that satisfy the assertion of the program and thus it fails.

Let $X(\textsc{Neg}) = \{X_1 \dots X_m\}$ be the executions of all programs $P_- \in \textsc{Neg}$ that compute states which satisfy the corresponding reachability assertions. Since we require Neg to fail for the synthesized model, we only need to consider relations that contribute towards Neg failing for the synthesized model. An acyclicity constraint over such a relation invalidates some execution $X_i \in X(\textsc{Neg})$. We only consider relations that forbid at least one such execution. We say a relation r is *relevant* if Pos succeeds for r and at least one execution $X_i \in X(\textsc{Neg})$ is not consistent with $acyclic(r)$. The following theorem shows that it is sufficient to consider relevant relation to synthesize a memory model.

Theorem 13. *Let $\mathcal{M}_{rel}$ consist only of the constraints over relevant relations of $\mathcal{M}$. If $\mathcal{M}$is correct, $\mathcal{M}_{rel}$ is correct as well.*

Proof. Assume $\mathcal{M}$ is correct. This means Pos succeeds and Neg fails for $\mathcal{M}$. Any execution consistent with $\mathcal{M}$ is also consistent with $\mathcal{M}_{rel}$ since it contains a subset of the constraints of $\mathcal{M}$. It follows that Pos succeeds for $\mathcal{M}_{rel}$.

Assume Neg does not fail for $\mathcal{M}_{rel}$. There is an execution $X_i \in X(\textsc{Neg})$ that is consistent with $\mathcal{M}_{rel}$ but not with $\mathcal{M}$. Since the only difference between the memory models are non-relevant relations, there is a non-relevant relation r such that X_i does not satisfy $acyclic(r)$. This implies that r is relevant which is a contradiction. □

With this in mind, we want to know which relations are relevant. For a relevant relation r, we additionally need to know the executions of $X(\textsc{Neg})$ that are not consistent with $acyclic(r)$. The straightforward approach to acquiring this information is with multiple SMT queries. We will show that some of this information can be obtained faster in the expansion phase in Section 6.3.2 and 6.4. For every execution $X_i \in X(\textsc{Neg})$, we compile a set $rel(X_i)$ of relevant relations that are acyclic in X_i. This list can easily be compiled as a byproduct of identifying which relations are relevant.

Let r_{new} be the relation added in the previous expansion phase. If r_{new} is not relevant, then it is not useful for the synthesis and the checking phase can be skipped. If it is relevant, then the checking phase is initiated by calling the function Check($\mathcal{M}$) given in Algorithm 5 with the parameter $\mathcal{M} = \{acyclic(r_{new})\}$.

The idea is to target the search by looking for executions in $X(\textsc{Neg})$ that are supposed to be inconsistent with the synthesized model but consistent with the current one. Then, we try out constraints over relation that are sure to forbid those executions. The back tracking search only considers memory models that contain the relation r_{new} that was newly added in the expansion phase. Any model without r_{new} constructed from constraints over relations of Rels has already been considered in previous checking phases.

Algorithm 5: Check($\mathcal{M}$)

```
1  Find the first $X_i \in X(\textsc{Neg})$ such that $X_i \in cons(\mathcal{M})$ ;
2  if there is no such execution then
3      return $\mathcal{M}$ ;
4  end
5  foreach $r \in rel(X_i)$ do
6      if $\neg redundant(r)$ then
7          $\mathcal{M}' := \mathcal{M} \cup \{acyclic(r)\}$;
8          if Pos succeeds for $\mathcal{M}'$ then
9              Check($\mathcal{M}'$) ;
10         end
11     end
12 end
```

First, the algorithm determines whether Neg fails for $\mathcal{M}$ in Line 1. This can be done with multiple SMT queries according to the encoding in Chapter 4. If Neg fails for $\mathcal{M}$, then the synthesis is successful (Line 3).

If Neg does not fail, then there is a counter-example execution X_i found in Line 1. The algorithm then tries out all possibilities to remove this counterexample. For any relation r in $rel(X)$, it adds an acyclicity constraint over r to $\mathcal{M}$ (Line 7). If Pos still succeeds for the resulting model $\mathcal{M}'$ (Line 8), then recursive calls to $Check(\mathcal{M}')$ are used to attempt to add further constraints to $\mathcal{M}'$ until Neg fails for it (Line 9).

Note that this check is indeed necessary. Even though we only consider relevant relations for which Pos succeeds individually, they may forbid different executions of a program $P_+ \in$ Pos and thus P_+ may fail for a model that combines constraints over those relations.

In Line 6, we further restrict the relations we attempt to add to the model by introducing the notion of redundant checks. The idea is that some executions may be forbidden by the same relations. We want to avoid constructing the same memory model more than once by adding constraints on the same relations in different orders. We ensure that those relations are always added according to their order in Rels. We call adding a relation r to $\mathcal{M}$ *redundant* if the following holds: There is a relation r' that was added earlier in a traversal of $rel(X_j)$, r occurs before r' in Rels, and $r \in rel(X_j)$. This redundancy means we do not need to add r. We can try out the same model by adding r in the earlier traversal of $rel(X_j)$ and then add r' afterwards.

It remains to show that the checking algorithm is both correct and complete.

Lemma 8. *Any synthesized memory model $\mathcal{M}$ is correct.*

Proof. It is an invariant of $Checking(\mathcal{M})$ that Pos succeeds for $\mathcal{M}$. This is true in the initial call of the function since r_{new} is relevant and thus Pos succeeds for the parameter $\mathcal{M} = \{acyclic(r_{new})\}$. It holds in any recursive call of the function in Line 9 since it is ensured that Pos succeeds for $\mathcal{M}'$ in Line 8.

The algorithm only returns a model in Line 3 if no execution in $X(\textsc{Neg})$ is consistent with $\mathcal{M}$. This is the case if NEG fails for $\mathcal{M}$. □

Lemma 9. *If there is a correct memory model $\mathcal{M}$ that consists of acyclicity constraints over relations in* RELS*, then the checking phase synthesizes such a memory model.*

Proof. According to Theorem 13, we can assume that $\mathcal{M}$ contains only acyclicity constraints on relevant relations. We can further assume that it is minimal. If we remove one constraint from $\mathcal{M}$, then it is no longer correct.

We examine a path of the back-tracking algorithm that synthesizes $\mathcal{M}$. Let r_{new} be the relation with $acyclic(r_{new}) \in \mathcal{M}$ that occurs last in RELS. After r_{new} is added in the expansion phase, $Checking(\{acyclic(r_{new})\})$ is called. For any execution $X_i \in X(\textsc{Neg})$ we denote $rel(\mathcal{M}, X_i)$ as the set of relations with acyclicity constraints in $\mathcal{M}$ that forbid X_i. Since NEG fails for $\mathcal{M}$, the set is not empty for any execution in $X(\textsc{Neg})$. The algorithm iterates through $X(\textsc{Neg})$ from X_1 to X_m. We examine the path that is as follows. When processing an execution X_i that is not yet forbidden by the model, we look at the recursive method call where the relation that is added is the one from $rel(\mathcal{M}, X_i)$ that occurs first in RELS.

It is easy to see that adding this relation r is never redundant. Assume there is an earlier X_j $(j < i)$ with $r \in rel(X_j)$ and a relation r' that was added when traversing $rel(X_j)$. It follows $r, r' \in rel(\mathcal{M}, X_j)$. Since r' was added instead of r, it holds that r' occurs before r in RELS.

Since $\mathcal{M}$ is correct, it satisfies the condition in Line 2. No relation that is added is redundant and $\mathcal{M}$ is minimal. It follows that the checking phase either constructs $\mathcal{M}$ or returns another correct memory model before that. Either way, the synthesized model is correct and consists of acyclicity constraints over relations in RELS. □

6.3 Expansion by Enumeration

ARAMIS maintains and continuously expands a list of relations RELS. In the checking phase, the tool combines constraints over those relations in order to synthesize a memory model. There are three main goals in expanding RELS.

- The relations should be useful to the checking phase.
- The expansion should be fast.
- Additional information on the relations should be provided, such as relevance and which executions contain cycles.

Any expansion method presents a trade off between those goals. We present expansion by enumeration, the easiest way to construct RELS. It simply enumerates all possible relations. It is very fast as it generates new relations in constant time. However, this synthesis method is not directed. Whether an added relation is useful is not a factor. We will show in Section 6.3.2 that some additional information can be provided.

Initially, we add the base relations to the list. We then repeatedly expand it by applying operators to the relations already listed. This is done as follows. We distinguish between *new* relations to which operators have not been applied yet and *used* relations which have already been used to derive new relations in previous steps. In an expansion step, we pick a new relation r. Then, we apply the unary operators to r, adding relations $r^{\otimes}$ for $\otimes \in \{*, -1, +\}$ to the list. We apply the binary operators to every pairing of r with a used relation r'. This adds $r \oplus r'$and $r' \oplus r$ for $\oplus \in \{\cap, \cup, \setminus, ;\}$. We then mark r as used.

This approach does not yet include recursively defined relations. We can handle a bounded number of recursively defined named relations by adding their names to the initial list along with the base relations (see Section 6.5). This also requires us to add k named relations to the memory model in the checking phase. If we change the definition of a named relation, then all relations that refer to it change as well.

6.3.1 Normal Form of Relations

In order to increase efficiency, we identify some superfluous relations and do not add them to Rels. Although listed relations are syntactically different, they may still be semantically equivalent, meaning they have the same semantics. Formally, we define two relations r and r' as *semantically equivalent* ($r \equiv r'$) if both r and r' consist of the same set of event pairs for any execution. For example the relation $po \cup fr$ is semantically equivalent to $fr \cup po$.

We can replace any relation in a memory model with an equivalent relation without changing the semantics of the memory model. This means if two listed relations are equivalent, we only have to use one of them in the checking phase. Each time we add a relation that is equivalent to an already listed relation, we introduce redundancy which increases the runtime of the synthesis. We want to avoid adding those relations.

In a general sense, we fix the conditions under which associative and distributive operators on relations are applied and the order of relations on which commutative operators are applied. We observe that the operators on relations satisfy the following semantic equivalences:

- Union and intersection are commutative:

$$r_1 \cap r_2 \equiv r_2 \cap r_1, \qquad r_1 \cup r_2 \equiv r_2 \cup r_1.$$

- Union, intersection and composition are associative:

$$\forall_{\oplus \in \{\cap, \cup, ;\}} : \quad (r_1 \oplus r_2) \oplus r_3 \equiv r_1 \oplus (r_2 \oplus r_3).$$

- Composition distributes over intersection and union:

$$r_1; (r_2 \cup r_3) \equiv (r_1; r_2) \cup (r_1; r_3),$$

$$r_1; (r_2 \cap r_3) \equiv (r_1; r_2) \cap (r_1; r_3).$$

- Union and intersection distribute over each other:

$$r_1 \cap (r_2 \cup r_3) \equiv (r_1 \cap r_2) \cup (r_1 \cap r_3),$$
$$r_1 \cup (r_2 \cap r_3) \equiv (r_1 \cup r_2) \cap (r_1 \cup r_3).$$

- The inverse distributes over any operator:

$$\forall_{\oplus \in \{\cap, \cup, \setminus\}} : \quad (r_1 \oplus r_2)^{-1} \equiv r_1^{-1} \oplus r_2^{-1},$$
$$(r_1; r_2)^{-1} \equiv r_2^{-1}; r_1^{-1}, \qquad (r^*)^{-1} \equiv (r^{-1})^*.$$

- Setminus distributes over union and intersection and interchanges them on the second component:

$$(r_1 \cup r_2) \setminus r \equiv (r_1 \setminus r) \cup (r_2 \setminus r),$$
$$(r_1 \cap r_2) \setminus r \equiv (r_2 \setminus r) \cup (r_2 \setminus r).$$
$$r \setminus (r_1 \cup r_2) \equiv (r \setminus r_1) \cap (r \setminus r_2),$$
$$r \setminus (r_1 \cap r_2) \equiv (r \setminus r_1) \cup (r \setminus r_2).$$

We remove any redundancy caused by these properties by restricting the relations added to the list. We only add relations in the following *normal form*:

- Associative Operators can only be nested from the left.
- Commutative Operators are always ordered by the position of the relations in the list.
- A union is never nested in an intersection and an intersection or a union is never nested in a composition.
- A union or an intersection is never nested in a setminus.
- The inverse operator is only applied to base relations.

This use of normal forms increases performance in general, since redundant computations are avoided. In some specific cases however, this can even slightly reduce performance. The order in which the semantically new relation are added may change. For instance, in the synthesis of sequential consistency (SC), we now have to use more expansion steps in order to get the normal form $(((po \cup fr) \cup rf) \cup co)$ than for the equivalent $((po \cup fr) \cup (rf \cup co))$ that would have been added earlier.

Note that this technique does not entirely remove redundancy. For instance the relations rf and $rf \cup (rf \cap po)$ are semantically equivalent and they are both in normal form. Our notion of semantic equivalence induces equivalence classes. In order to avoid adding redundant relations entirely, we would need to ensure that only one relation of each equivalence class is added. This requires a method to ascertain whether a relation we wish to add is equivalent to a relation in the list. However, checking whether two relations are equivalent is not trivial. The trade-off between efficiency gained by avoiding more redundant relations and efficiency lost by longer equivalence checks needs to be considered.

This notion of normal forms presents an under-approximation of an equivalence checker. Since we do not actually compare relations but simply avoid applying operators to certain relations, it is instantaneous. Our normal forms present a fast approximation of an equivalence checker that identifies many equivalent relations.

6.3.2 Annotated Lists

The checking phase requires information about the relations in order to decide whether and when to add it to a memory model. In particular, we need to ascertain whether a relation is relevant and if so, which execution in $X(\textsc{Neg})$ contain cycles of it.

For a relation r, we store a set of programs from Pos for which the check of acyclicity succeeds. In particular, we are interested in whether the check succeeds for Pos. We can determine whether a program succeeds for r with an SMT-query using the encoding from Dartagnan.

We also store a set of executions of Neg that succeed for r. Note that we do not require an SMT query in order to decide whether an execution succeeds for r. For any execution, we can check whether it succeeds for r by computing the set of edges in r on the execution. This is done in a straightforward manner analogue to the computation of the may pairs (see Section 4.6). We then check whether the set contains a cycle. We use our technique of computing the active set of a relation in an acyclicity constraint. If this returns an empty set, then the relation is acyclic in the execution. It follows that the execution succeeds for r.

It is not always necessary to compute all the additional information anew when adding relations. We show that when an operator on relations is applied, then the resulting relation inherits some properties from the relations the operator is applied to. Given two relations r and r' it holds $r \subseteq r'$ iff the following is true for every execution: Any pair of events defined by r is also contained in the set of event pairs given by r'.

Theorem 14. *Assume $r' \subseteq r$.*

1. *A check that succeeds for r also succeeds for r'.*
2. *A check that fails for r' also fails for r.*
3. *A check succeeds for $r_1; r_2$ if and only if it succeeds for $r_2; r_1$.*
4. *A check succeeds for r if and only if it succeeds for r^{-1}.*

Proof. 1. If a check succeeds for r, then r contains no cycle in the corresponding execution X. Since $r' \subseteq r$ holds that the even smaller relation r' contains no cycle in X and thus succeeds at that check as well.

2. If a check fails for r', then any execution X that reaches a state satisfying the reachability assertion contains a cycle of r'. Any such cycle in X is also a cycle of the larger relation r and thus the check fails for r as well.

3. If a check fails for $r_1;r_2$, then any execution X that reaches a state satisfying the assertion contains a cycle of $r_1;r_2$. This is the case exactly if there is a cycle $e_1 \overset{r_1}{\to} e_2 \overset{r_2}{\to} e_3 \overset{r_1}{\to} \dots e_1$ labeled by a word in $(r_1.r_2)^+$. It follows that there is also a cycle $e_2 \overset{r_2}{\to} e_3 \overset{r_1}{\to} \dots e_1 \overset{r_1}{\to} e_2$ labeled by a word in $(r_2.r_1)^+$. We simply have to start the cycle at the second letter. This means there is a cycle of $r_2;r_1$. Thus the check fails for $r_2;r_1$ as well. The other direction is analog.

4. If a check fails for r, then any execution X that reaches a state satisfying the assertion contains a cycle of r. This is the case exactly if there is a cycle $e_1 \overset{r}{\to} e_2 \overset{r}{\to} e_3 \dots e_1$. It follows that there is also a cycle $e_1 \overset{r^{-1}}{\to} \dots e_3 \overset{r^{-1}}{\to} e_2 \overset{r^{-1}}{\to} e_1$. We simply inverse the direction the cycle. This means there is a cycle of r^{-1}. Thus the check fails for r^{-1} as well. The other direction is analog.

□

We use Theorem 14 together with the following properties in order to reuse information about child relations.

$$\begin{array}{cccc}
r \subseteq r \cup r' & r' \supseteq r \cap r' & r' \cup r \supseteq r;r' & r' \cap r \subseteq r;r' \\
r' \subseteq r \cup r' & r \supseteq r \cap r' & r \cup r' \supseteq r;r' & r \cap r' \subseteq r;r' \\
r \subseteq r^* & r \supseteq r \setminus r'. & r \subseteq r^+ &
\end{array}$$

When applying an operator on relations r and r' in order to expand Rels by adding a relation $r \oplus r'$ or $r^{\otimes}$, we can take advantage of the above properties. Assume the operator is $r \cap r'$. It holds $r \cap r' \supset r$ and $r \cap r' \supset r'$, we can apply Theorem 14. It follows that if Pos succeeds for r or r' then it also succeeds for $r \cup r'$.

Some of these properties have an effect in the opposite direction. Assume the operator is $r \cup r'$. It holds $r \cup r' \subset r$ and $r \cup r' \subset r'$. We can again apply Theorem 14. It follows that if X_i contains a cycle on r or r', then it also contains a cycle on $r \cup r'$.

6.4 Expansion by CEGIS

We present a counter-example guided inductive synthesis (CEGIS) method that uses an SMT-solver to produce relevant relations. We introduce *template relations*. They don't specify which operators and basic relations are used. Instead, we let the SMT-solver choose them in such a way that they forbid some executions of Neg and that Pos succeeds for them.

We construct a formula that encodes the Pos programs together with some executions of Neg programs and acyclicity of a template relation. A satisfying assignment consists of an instantiation of the template relation. This relation is generated in such a way that all encoded Pos succeed and every encoded Neg execution contains a cycle. We refine this process by checking whether a $P_- \in$ Neg succeeds for an instance. If it does not succeed, we obtain a counterexample execution and we add its encoding to the formula.

6.4.1 Template Relations

A template relation models all possible relations that are not recursively defined up to a fixed syntax tree depth. In the syntax tree of a template relation, each node represents a relation. The template consists of a complete binary tree. Let $\varphi_{r,n}$ denote the encoding of a named relation r where the name of the node n is the name of the relation as well and let *basic* be the set of basic relations we wish to encode. For a relation r we introduce a Boolean variable $[r]_n$ that denotes whether the SMT solver chooses to encode the relation r in node n. A leaf represents a base relation and an inner node represents a binary operator on the left and right child, or a unary operation on the left child, or no operator on the left child.

Each leaf node encodes a named template relation which can contain any base relation depending on the assignment. The encoding of a leaf n consists of a disjunction over all possible encodings of base relations:

$$\varphi_n = \bigvee_{b \in basic} \left([b]_n \wedge \varphi_{b,n} \wedge \bigwedge_{b' \in basic \setminus \{b\}} \neg [b']_n \right)$$

Exactly one encoding of a base relation $\varphi_{b,n}$ can be assigned to the leaf and the corresponding variable $[b]_n$ is the only one that is true. We can easily determine which base relation b a leaf n encodes in a satisfying assignment by checking which corresponding variable $[b]_n$ is true.

In a similar way we encode each inner node n with children n_1 and n_2 as a choice over all relations that consist of operators applied to n_1 and n_2. We define the encoding φ_n of a node n. The idea behind the formula is as follows. For every relation r obtained by applying a binary operator $\oplus \in \{\cap, \cup, \setminus, ;\}$ to n_1 and n_2, the formula φ_n encodes r in $\varphi_{r,n}$ if and only if the corresponding variable $[r]_n$ holds (Condition 6.1). In addition, exactly one variable $[r]_n$ is required to hold (Condition 6.2 and 6.3). The same goes for every a unary operator $\otimes \in \{*, -1, +, \mathtt{eq}\}$ (see Condition 6.4-6.6).

We define the unary equality operator eq. It is encoded by $\mathsf{eq}(r)(e_1, e_2) \Leftrightarrow r(e_1, e_2)$. It does not change a relation, it only provides it with a new name.

$$\varphi_n := \bigwedge_{\oplus \in \{\cap, \cup, \setminus, ;\}} ((\varphi_{\oplus(n_1,n_2),n} \leftrightarrow [\oplus(n_1, n_2)]_n) \wedge \tag{6.1}$$

$$\bigwedge_{\oplus' \in \{\cap, \cup, \setminus, ;\}} \neg([\oplus(n_1, n_2)]_n \wedge [\oplus'(n_1, n_2)]_n)) \wedge \tag{6.2}$$

$$\bigvee_{\oplus \in \{\cap, \cup, \setminus, ;\}} [\oplus(n_1, n_2)]_n \wedge \tag{6.3}$$

$$\bigwedge_{\otimes \in \{*, -1, +, \mathsf{eq}\}} ((\varphi_{\otimes(n_1),n} \leftrightarrow [\otimes(n_1), n]_n) \wedge \tag{6.4}$$

$$\bigwedge_{\otimes' \in \{*, -1, +, \mathsf{eq}\}} \neg([\otimes(n_1), n]_n \wedge [\otimes'(n_1), n]_n)) \wedge \tag{6.5}$$

$$\bigvee_{\otimes \in \{*, -1, +, \mathsf{eq}\}} [\otimes(n_1), n]_n \tag{6.6}$$

By accessing the different variables of the forms $[\oplus(n_1, n_2)]_n$, $[\otimes(n_1)]_n$, and $[b]_n$, we obtain a relation of a given maximal syntax tree depth from the satisfying assignment of a single quantifier free SMT query. For this relation, we know that all encoded POS succeed and the encoded set of executions from $X(\text{NEG})$ are inconsistent.

The new operator eq is necessary to model syntax trees that are not complete but have leafs on different depths. Assume we use a template relation with a syntax tree depth of two and we want to generate the relation $\xrightarrow{rel} = (po \cup co) \cup rf$. A satisfying assignment can assign r to the root node and $(po \cup co)$ to its left child node. The right child node of the root however is not a leaf and thus cannot give rf directly. Instead, the equality relation eq can be assigned which ensures that the relations is equal to its left child which is a leaf that encodes rf.

6.4.2 The CEGIS Loop

We present a Counter-Example Guided Inductive Synthesis loop that implements the expansion phase. The idea is to encode acyclicity of some template relation for all programs in POS in a formula φ. We extract a relation r that instantiates the template relation from a satisfying assignment of the formula. We expand RELS by adding r. We then keep looking for different executions $X_j \in X(\text{NEG})$ that are acyclic for r. We add constraints such that an instance of the template relation forbids X_j and attempt to extract new instances. This method ensures that the extracted relations are relevant.

We start by constructing a template relation $tRel$ of a given syntax tree depth k. We construct a formula φ that encodes reachability for each program $P_+ \in \text{POS}$ together with the acyclicity constraint on the template relation $\varphi_{acyclic(tRel),P_+}$. The latter ensures reachability is consistent with $acyclic(tRel)$.

$$\varphi := \bigwedge_{P_+ \in \text{POS}} \varphi_{reach(P_+)} \wedge \varphi_{acyclic(tRel),P_+}.$$

Algorithm 6: CEGIS(φ, X_i)

```
1  Solve($\varphi$);
2  if $\varphi$ is satisfied then
3      $r_{new} := extract(\varphi, tRel)$;
4      RELS.add($r_{new}$);
5      Check($\{acyclic(r_{new})\}$);
6      foreach $X_j \in \{X_{i+1} \dots X_m\}$ do
7          if $X_j$ succeeds for $r_{new}$ then
8              CEGIS($\varphi \wedge \varphi_{cyclic(tRel),X_j}, X_j$);
9          end
10     end
11 end
```

We then call the the method *CEGIS* given by Algorithm 6 with parameters $CEGIS(\varphi, X_0)$. If φ has a satisfying assignment, we extract a relation r_{new} from the satisfying assignment of φ that instantiates the template relation *tRel* (Line 3). The relation r_{new} is added to RELS (Line 4) and the checking phase is executed for the new relation in Line 5. Then we enter the next step in the search where we want to ensure that some execution of $X(\text{NEG})$ fails. For some unused $X_j \in X(\text{NEG})$, we check if it succeeds for r (Line 7). Since the execution X_j is given, we recall that the set of pairs in r can be obtained and checked for acyclicity without an SMT query.

If X_j succeeds for r_{new}, then we ensure that the next instantiation of the template relation does not allow X_j. This is done by adding a cyclicity constraint $\varphi_{cyclic(tRel),X_j}$ to φ in Line 8 which ensures that the template relation forbids X_j in the future. Then we go back to computing another relation r_{new} from φ.

If φ becomes unsatisfiable at some point, we backtrack and continue with another execution. Once all possible combinations of executions have been considered, the CEGIS loop is complete. If we require more relations, we start a new CEGIS loop with a higher syntax tree depth for the template relation.

Internally, an execution X_j is represented by a set of event pairs for each relation in *basic*. It is obtained from a satisfying assignment of a reachability query. In order to encode the condition $cyclic(tRel)$ for an execution in $\varphi_{cyclic(tRel),X_j}$, we encode X_j together with the template relation using unique new names. The fresh relation names are necessary since we may encode several executions of the same program P_- together. Those executions occur on the same events and contain the same base relations. We need to distinguish them from each other.

If the CEGIS loop has terminated without synthesizing a memory model, then we restart the loop with a larger syntax tree depth for the template relation. We want to avoid adding relations that are equivalent to those already added with a smaller syntax tree depth. This means we need to ensure that a generated relation r really has tree depth k and not a smaller depth. To this effect, we add a condition to the encoding of the template relation that ensures that in the syntax tree, there is a path from a leaf to the root that does not contain a node encoding the operator eq.

Depending on the size of the input sets Pos and Neg as well as the sizes of the programs themselves, it may not be feasible to encode all programs in Pos immediately. In this case, we can call the solver multiple times with subsets of Pos of a fixed size. These encodings only have to be constructed once in a preprocessing step.

6.5 Recursion

Our synthesis method supports a bounded number k of recursively defined relations in the synthesized model. This is implemented by introducing k relation names. A major challenge of recursively defined relations is that it is not sufficient to try out constraints over named relations which are defined recursively. One also needs to consider constraints over derived relations that refer to recursively defined relations.

If we use enumeration in the expansion phase, then the relation names are treated like base relations during the expansion. In the checking phase, we try out all possibilities of assigning k relations from Rels to the names. We then execute the checking phase for each assignment. Each such assignment creates new definitions of the named relations and thus of all derived relations that refer to them. This means whether a derived relation is relevant needs to be reevaluated for each assignment. This slows down the synthesis considerably.

The CEGIS method of expanding is better suited for synthesizing recursively defined relations. When encoding template relations, we again handle the named relations similar to base relations. We add the k relation names as choices to the encoding of the leaves. We then add k additional template relations to the formula φ in the CEGIS loop. We ensure that those additional template relations support recursively defined instances by assigning the k relation names to the encoding of corresponding root nodes.

6.6 Relation Analysis

In Section 4.6, we presented may pairs and active sets which reduce the size of the encoding. We make use of this technique in the synthesis as well. Note that the may pairs of a derived relation depend on which program is currently examined. Apart from the program they depend on the may pairs of the child relations. It follows that we only have to encode the may pairs for every relation in the list and for each program once, they do not change. We can use dynamic programming and store the may pairs for every relation and each program in Pos. Note that this presents a trade-off between time and space complexity of the method. In our implementation Aramis, the feature is optional and depends on whether the input parameter -dynamic is set.

The active pairs however depend on the way a relation is used. To illustrate this, consider a relation r that occurs only in the constraint $\mathit{acyclic}(r)$. Its active set contains all pairs that could occur in a cycle of r. If we add a constraint $\mathit{acyclic}(r \cup r')$, then the active pairs also contain those pairs of r that could occur in a cycle that consists of both r and r'. We need to recompute active sets any time a relation is encoded.

6.7 Sketch Based Synthesis

Since memory model synthesis is a complex task that does not scale well, it is often crucial that the user provides additional information about the model to be synthesized in order to complete the synthesis in a reasonable time.

An upper limit on the number of constraints k may be provided. Since the number of constraints in a model is usually very small, a reasonable upper bound could be guessed even with little or no knowledge of the synthesis task. This is easily implemented by modifying Algorithm 5 such that it performs a back-tracking step if either there is no next relation in $\mathit{rel}(X)$ or the model already contains k constraints. This ensures that no more than k constraints are synthesized.

Often, some guarantees of the model to be synthesized are already known. They can be provided to the tool in form of a memory model $\mathcal{M}_w$ that is weaker than $\mathcal{M}$. This is implemented by initializing the checking phase with the parameter $\mathit{Checking}(\{\mathit{acyclic}(r_{new})\} \cup \mathcal{M}_w)$. Additionally, we restrict the set $X(\textsc{Neg})$ to executions that are consistent with $\mathcal{M}_w$. If $\mathcal{M}_w$ contains some named relations, we attempt to use them as building blocks in further derived relations. This is incorporated in the expansion phase by treating them like base relations. Naturally we still initialize the checking phase with $\mathcal{M}_w$. We also support manual input of the relations that form the set *basic* and are used as building blocks for further relations. We define the set of relations that may be subtracted by the set-minus operator separately.

6.8 Experiments

We implemented memory model synthesis in the prototype tool Aramis which uses Z3 [65] as the back-end SMT solver. It allows us to synthesize memory models in the `cat` language. We attempt to synthesize SC using expansion by enumeration. We run Aramis on inputs that are sets of litmus tests randomly chosen from the library of Herd7, sorted into Pos and Neg depending on whether the litmus test succeeds under SC. The experiments were performed on a 1,7 GHz Intel Core i7 with 8GB memory. The execution times are given in minutes and seconds.

Since the purpose of litmus tests is to characterize weak memory behavior, there are only few litmus tests that succeed under SC. When examining the size of an input, we focus on the crucial factors: the number of programs in Pos

Pos	Neg	X(Neg)	*basic*	`Dyn`	SC	Z3	Rels	Time
21	30	110	4	on	✔	20273	2504	44
21	50	327	4	on	✔	20273	2504	46
30	82	432	4	on	✔	28906	2504	1:03
21	30	110	4+1	on	✔	83592	9198	2:17
21	50	327	4+1	on	✔	83829	9198	2:40
30	82	432	4+1	on	✔	122315	9198	3:56
21	50	327	5+1	on	✔	98889	10519	3:06
30	82	432	5+1	on	✔	141164	10519	4:29
21	30	110	6	on	-	-	-	115:35
21	30	327	6	off	-	-	-	205:30

Table 6.1: Synthesis of SC using enumeration.

and the number of executions of programs in Neg. Further parameters allow us reduce the run-time and improve the quality of the synthesized model. We provide the number of base relations used in the expansion. An entry $x + y$ in the *basic* column denotes a set of size x containing the base relations that are combined into derived relations and y many relations that may be subtracted using set-minus. We also specify whether we use dynamic synthesis in order to store the may pairs in column `Dyn`. To evaluate an execution of Aramis, we list the number of SMT-queries to Z3 produced by the tool, the number of relations produced by the expansion phase, as well as the synthesis times.

We can give a bound on the maximal number of constraints in the synthesized model as an input parameter. Since there is only one constraint in SC and the corresponding relation $po \cup co \cup fr \cup rf$ immediately produces the correct result in the checking phase, the parameter does not have much of an impact for SC and thus we omit it.

The results are given in Table 6.1. The run-time of the tool fluctuates strongly with the input and the given parameters. In particular the number of base relations has a huge effect on the efficiency of the synthesis. For the simple SC model, even small inputs of 51 litmus tests lead to a correct synthesis. An increase in the number of input programs does not lead to a significantly longer synthesis time. This confirms that litmus tests are well suited to describe a memory model and that Aramis is able to handle even larger sets of litmus tests as input. This is the case even though the litmus tests in our experiments are randomly chosen.

In the second to last entry, the tool reaches the limit of the java garbage collector overhead and stops. We suspect this is due to the fact that Aramis stores too many may sets in order to avoid computing them multiple times. The last entry shows that the tool runs indeed longer when the may sets are not stored. However, it still insufficient to synthesize a result, another garbage collector overhead error is produced.

We expect that constructing new relations in a targeted way using the expansion by CEGAR will provide a significant speed-up to these results and

Program Synthesis	**Model Synthesis**
Specification φ	Tests (Pos, Neg)
Expression e	Model $\mathcal{M}$
Function f	A decider which test succeeds
Background theory T	Semantics of executions and consistency
Grammar for a set of expressions L	Restricted version of `cat`
Input-output pair example	Execution of some $P_- \in$ Neg

Table 6.2: Program Synthesis vs. Model Synthesis.

enable us to tackle harder instances of the synthesis problem. The implementation of this module in Aramis is left as a topic for future work. Our results here serve as a proof of concept that our approach to memory model synthesis can produce practical results.

6.9 Memory Model vs Syntax Guided Synthesis

While memory model synthesis is a new area of study, program synthesis is a more established field. We adapted methods and insight from syntax guided program synthesis (SYGUS) [25] in order to synthesize memory models. In this chapter, we study the SYGUS problem and relate it to the memory model synthesis problem. We present several established concepts and methods of program synthesis and propose corresponding approaches for memory model synthesis. The relation is summarized in Table 6.2.

Program Synthesis The core computational problem of program synthesis is defined as follows: Find an expression e in a background theory T such that e is an implementation of some function f that satisfies a given specification φ.

This problem translates to memory model synthesis in a straightforward way. In memory model synthesis, the expression e we want to synthesize is the memory model $\mathcal{M}$. The background theory is the semantics of program executions and consistency with CAT memory models. The function f we want to implement is a decider whether an execution is consistent. The specification φ is given by Pos and Neg.

Syntax Guided Synthesis Problem If the search space for program synthesis contains all possible expressions e in the background theory T, then it is very large. In syntax guided synthesis, we assume that we have some vague knowledge about the shape of the synthesized program. This allows us to restrict the search space with some template. Formally, we introduce a grammar that defines a set L of expressions built from the background theory T. The language

L is chosen such that it contains the expression we attempt to synthesize ($e \in L$).

In memory model synthesis, we restrict the search space to the core of the CAT language (see Fig. 2.5) and to acyclicity constraints. We can further adapt the template by specifying which fences and base relation are used and how many axioms or recursive definitions are used. In Section 6.7 we give an overview over possible restrictions to the memory model. Memory model synthesis has an advantage over program synthesis since we can check a single constraint against the specification in isolation first and obtain useful information about which models might satisfy the specification. We only need to consider relevant relations where all Pos succeed and at least some Neg behavior is inconsistent (see Section 6.2). We only need to try out memory models where every execution of a $P_- \in$ Neg that satisfies the reachability condition is forbidden by a constraint. These properties significantly restrict the language of memory model candidates.

6.9.1 CEGIS

Counter-Example Guided Inductive Synthesis [155, 154] is an active learning method. Here, the learning algorithm uses a *concept class* which consists of a set of examples. An example is an interpretation of f in the background theory. If it is consistent with the specification, then we say it is positive. In program synthesis, an example consists usually of an input-output pair. The learning algorithm continually constructs expressions that are consistent with the concept class and queries an oracle to verify them. The oracle usually consists of an SMT solver, a constraint solver, or a model checker. If the expression is not consistent with the specification, then the oracle provides a counterexample. The examples are used to improve the synthesis in further iterations.

In memory model synthesis, an example is an execution. A counterexample can be a consistent execution of some $P_- \in$ Neg that succeeded. This is implemented in Aramis. The other possibility is that some $P_+ \in$ Pos failed. Here, we can provide some inconsistent execution that reaches the state. However there may be multiple inconsistent executions that reach the state and we don't know which one is the right choice. Ideally, the input programs are chosen such that the different executions of a program that satisfy the reachability assertion all describe the same property of the memory architecture. In this case such an execution is still useful even if we can't guarantee that it is a correct counterexample. We have to allow for "false" counterexamples in the learning algorithm. There are multiple possible approaches to dealing with this problem:

- We can remove those counterexamples from the concept class after a time.
- Instead of executions, we can use the programs of Pos as counterexamples.
- We could only compute relations for which Pos succeeds.
- We can simply disregard executions of Pos as counterexamples.

Our method for expansion by enumeration disregards executions of Pos as counterexamples while the expansion using CEGAR only computes expressions where Pos succeeds.

The following learning techniques can be used to synthesize expressions.

6.9.2 Enumerative Learning

The method enumerates expressions that satisfy all counter examples, namely values for which the expression violates φ. When a new expression is found that conforms with the counterexamples, it is verified against φ. If it is not verified, a new counterexample is found and the enumeration is started from scratch with the new set of counterexamples.

Our algorithm presented in Section 6.7 enumerates expressions. Unfortunately it does not support examples, it simply enumerates all constraints (partial expressions) and checks them against the specification.

6.9.3 Stochastic Search

In a stochastic search, each expression is given a score depending on how many concrete examples it satisfies. A random expression is picked and a subexpression is mutated with different sub-trees of the same size with a probability that depends on their scores. The method also allows for the size of the expression to be changed with a small probability. This converges at the correct size. Since a stochastic search traverses the search space randomly, different runs of the search are not expected to intersect much. This means the method is well suited for multiple independent parallel runs.

This method could be adapted to memory model synthesis as follows. We could give a relation a score depending on how many programs (or alternatively how many executions) it handles correctly. It might be beneficial to adapt the expansion step by applying operators to relations with a higher score first. However, such a score only makes sense with relation that don't simply allow all or none of the checks to succeed.

6.9.4 Constraint Based Learning

Constraint based learning is similar to enumerative learning. However, expressions are not simply enumerated and checked against counterexamples. Instead, a constraint solver is given the counterexamples and finds the next expression that satisfies them. It builds expressions from libraries that contain operators, the libraries are expanded if the constraint solver was unsuccessful. Our application of constraint based learning to model synthesis is given by the CEGIS loop presented in Section 6.4.

6.10 Related Work

We have given an overview of the related program synthesis in the last section. Being a new field of study, there are not many results on synthesis for weak memory. To the best of our knowledge, the only paper about memory model synthesis is [39]. They present the tool MemSynth, which accepts as input positive and negative reachability queries as well as a sketch of a memory model. A sketch is an axiomatic memory model with so called expression holes in it, i.e. relations that are not defined. Similar to our approach, MemSynth encodes the problem in a formula and passes it to a solver. In their approach the solver is a version of Rosette with an additional layer that handles expression holes. The key aspect of MemSynth is that it can solve the entire synthesis problem with one bounded relational logic query. Here, a hole is encoded by a nondeterministic derivation tree of a bounded depth d. This approach is used to solve not only model synthesis but also model ambiguity, model equivalence and program verification. They claim to be able to synthesize TSO in 2 seconds and Power in 12 seconds. We suspect that these short synthesis times are due to the very restrictive nature of the sketches.

Another synthesis tool based on `cat` constructs programs that differentiate two memory models [169]. This is of interest to memory model designers.

6.11 Conclusion and Future Work

We have presented Aramis, a synthesis tool for weak memory models in the `cat` language. The input is a set of program reachability queries that are supposed to succeed under the memory model and a set that is supposed to fail. It attempts to synthesize a memory model with the required behavior. The tool uses our encoding techniques for reachability and portability (see Chapter 4 and 5). Our approach consists of an expansion phase where we repeatedly construct new relations and a checking phase where we attempts to combine the relations to a memory model.

We present two methods of expansion: enumeration and a CEGIS loop. Since the approach of using SMT queries in a counterexample guided search was useful for solving state portability, we construct a more involved CEGIS method for the memory model synthesis. We introduce template relations, a concept that allows us to extract useful relations from satisfying assignments of SMT queries instead of executions.

We perform experiments on the prototype tool with expansion by enumeration which suggest that our approach to memory model synthesis can produce correct results for practical inputs in manageable time.

We discuss the more established field of program synthesis and how the prevalent approaches can be adapted for memory model synthesis. The application of our encoding techniques for reachability and portability to memory model synthesis shows that they can be used for more than program verification tasks. They could be applied to solve all manner of problems in the field of concurrency under weak memory. For instance, it is useful to study the differences of similar memory models in order to develop memory models. In the future, we hope to explore how our techniques can be applied to synthesize a program that succeeds for only one of two given memory models.

In Section C of the appendix, we present an idea how one could built on our synthesis method in order check whether an input (Pos, Neg) is sufficient to define a semantically unique CAT memory model. We also discuss some synthesis techniques in Section C that we hope to implement in future work. Namely, a stronger method of checking equivalence of relations and a more targeted synthesis of relations where cycles of base relations are identified in executions that should fail and a relation is constructed from these cycles. Another method we would like to implement is the stochastic search examined in Section 6.9.

Chapter 7

Petri Net Invariant Synthesis

Most program verification task consist of proving safety properties. We want to show that the reachable configurations are safe, meaning they do not show certain behavior that we want to exclude. Since most tasks in program verification are undecidable and even simple tasks are very complex [112], the program behavior is usually approximated. This is often done by generating an invariant, which can be understood as a property that holds true for every reachable configuration [75, 96, 36, 146]. We are looking for an invariant that acts as a separator between reachable and undesired configurations. The challenge is to find a type of invariant that is both not too hard to generate and expressive enough to show the required properties. Inductive invariants are a prominent approach for this[34, 60, 156]. An inductive invariant that is satisfied by a configuration always stays satisfied after an execution step.

We examine Petri nets, a well established approach to modeling concurrent systems [134, 130]. Their formal definition is given in Section 7.2. Petri nets are a powerful tool for mathematical and graphical modeling with a wide array of applications. Program verification tasks are usually expressed as either a Petri net reachability problem or a coverability problem. Both problems are decidable [125, 142, 51]. However, the reachability problem is EXPSPACE-hard [133, 38] and recently proven to be non-elementary hard [61]. The coverability problem is EXPSPACE-complete [142, 51]. It was first solved with minimal coverability sets by Karp and Miller [100]. This approach has been studied and improved extensively [99, 166, 87, 144], but it still exceeds practical limits. This indicates that the harder reachability problem should not be solved in an exact manner, but rather it should be approximated.

Well known invariants for Petri nets are place invariants [52], transition invariants, traps, siphons and the state equation [134]. *Place invariants* are defined as follows. Let n be the number of places. The vector $k \in \mathbb{Z}^n$ is a place invariant if there is a constant $c \in \mathbb{Z}$ such that for every reachable marking $m \in \mathbb{N}^n$ it holds $k \cdot m = c$. They are relatively easy to generate but not very expressive.

We examine the use of a more general version of place invariants in the form of inequalities by weakening the condition $k \cdot m = c$ to $k \cdot m \geq c$. Since the constant c is no longer uniquely determined by $k \cdot m$ due to the weaker condition "≥", we expand on the previous invariant definition: A vector $k \in \mathbb{Z}^n$ and a constant $c \in \mathbb{Z}$ form an invariant (k, c) if every reachable marking $m \in \mathbb{N}^n$ satisfies $k \cdot m \geq c$. Since each inequality $k \cdot m \geq c$ forms a half space over the n-dimensional space of markings, we refer to a tuple (k, c) as a *half space*. These inequalities are powerful enough to express standard place invariants, traps, siphons and more [162]. Our study of these half spaces encompasses their structural properties as well as the decision problem whether a given half space is inductive and culminates in a synthesis method for inductive half spaces.

Many inductive invariants are rather difficult to synthesize, Common approaches often employ continuous approximations in order to gain efficiency. Most prominently, linear programming is faster than mixed or integer programming [150] and the Farkas Lemma provides a powerful tool for solving a system of linear inequalities in the continuous case but not for integers. This leads to inexact methods that are not able to compute some invariants (see Section 7.1).

Our approach to synthesizing inductive invariants employs only discrete methods, it uses a Counter Example Guided Abstraction Refinement (CEGAR) [56] loop with an efficient algorithm and employs an SMT-solver for solving linear constraints. The main contributions of this chapter are as follows:

1. We perform a comprehensive study of half spaces as inductive invariants of Petri nets and derive linear descriptions of their structural properties.

2. We analyze the complexity of the decision problem whether a given half space is an inductive invariant. We find it co-NP-complete and in CSL.

3. We present a novel algorithm that solves the decision problem using dynamic programming. We prove its correctness and optimality as well.

4. We introduce a CEGAR method that efficiently synthesizes inductive invariants in the form of half spaces that separate the reachable markings from a given set of undesirable markings.

5. We implement the synthesis method in the prototype tool INEQUALIZER[1] and perform experiments in which we compare it to MIST [135], a safety checker for Petri Nets.

Given a Petri net N and a marking m_f, we study *linear safety verification* where we want to separate m_f from the reachable markings by an inductive half space. This comes in two variants: the reachability version with only one undesirable state m_f and the coverability version with a set of undesirable markings $\uparrow m_f$ that cover m_f.

Our CEGAR method for the reachability version attempts to synthesize an inductive invariant that is satisfied by the initial marking but not by m_f

[1]`https://github.com/florianfurbach/Inequalizer`

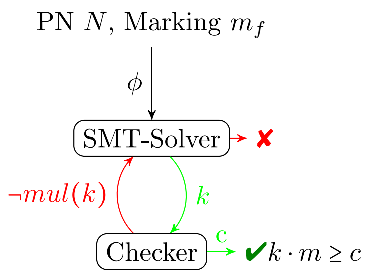


Figure 7.1: The CEGAR method.

(see Fig. 7.1). It can be easily adapted for the coverability version. It starts out by generating an SMT formula ϕ that consists of linear inequalities describing necessary properties of invariants. If there is no satisfying assignment, then there is no separating invariant. A satisfying assignment consists of a vector k. We now have to determine whether there is a constant c such that $k \cdot m \geq c$ is inductive. Our method for solving the decision problem whether a given half space is an inductive invariant can be applied here as well. We adapt it slightly in order to generate a suitable constant c instead of testing a given c. If we find a suitable c, then we have successfully synthesized a separating invariant (k, c). If not, we update the formula using a constraint $mul(k)$ which holds for any multiple of the vector k. We set $\phi := \phi \wedge \neg mul(k)$ to ensure that we discard multiples of k in the future.

Outline: The remainder of the paper is structured as follows. Related work is discussed below. In Section 7.2 we introduce the necessary formalisms. Next, we separate the space of inequalities into those that are trivialy inductive (Section 7.3) and those for which inductivity is harder to determine (Section 7.4). In Section 7.5 we describe necessary properties of t-inductive half spaces in a formula ϕ. An analysis of the decision problem and an algorithm that solves it are presented in Section 7.6. The experimental results are given in Section 7.7. Finally, we present a conclusion and outlook.

The results of this chapter are based on research done together with Peter Chini and Roland Meyer and have not been published before.

7.1 Related Work

Verification problems of Petri nets, in particular reachability and coverability, have been the subject of considerable study for some time [100] and efficient solutions keep getting developed [106]. Algorithmic solutions to coverability analysis for vector addition systems were first introduced by Karp and Miller [100]. There are many different approaches to this. Most techniques are based on a forward or backward state-space exploration [80, 86, 99, 101, 166].

A method that has drawn interest are unfoldings [73, 127, 71, 113]. Notably, Abdulla et al [7] solve coverability by constructing an unfolding that represents backwards reachable states and analyzing it using a SMT formula (of exponential size). Approximation refinement is used in [125].

Profiting from recent advances in SMT-solving techniques, deriving program properties by constraint solving has become a popular method [92, 7, 72, 156]. In [136], ranking functions are synthesized by solving linear inequalities in order to prove termination of unnested programs.

Petri net methods for reachability or coverability based on constraint solving often simplify the problem by using continuous values or disregarding the transition guards. This allows for simpler definitions of Petri nets or inductivity and provides tools like the Farkas Lemma but it also leads to fewer invariants. Esparza et al [72] build linear constraints from the marking equation and trap properties in order to disprove co-linear properties. Then, they apply the Farkas Lemma in order to generate an inductive invariant that disproves the property.

There is even already existing work studying the synthesis of half spaces by Sipma et al [147]. Their approach provides a closed-form characterization of inductive half spaces. However, they interpret Petri nets as transition systems over real values and thus their notion of inductivity does not allow for invariants that we call non-trivial. For instance, they cannot handle our example in Fig. 7.2. Our method builds on their approach by adapting their closed form to allow for non-trivial inequalities and using it inside a CEGAR loop.

Marvin Triebel examines the semantic properties of half spaces and shows that some inequalities describe more general versions of established inductive invariants like traps, siphons, and s-invariants [162].

7.2 Linear Safety Verification

We formulate the problem of safety verification for Petri nets by means of half spaces and introduce the necessary formalisms.

Petri Nets. A *Petri Net* is a tuple $N = (P, T, F)$, where P is a finite set of *places*, T is a finite set of *transitions*, and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a *flow function.* We denote the number of places $|P|$ by n. The places are numbered. For convenience, we use a place p_i and their numeric value i interchangeably: Given a vector $x \in \mathbb{N}^n$ we denote its i-th component as both $x(i)$ and $x(p_i)$. For a transition $t \in T$ we define vectors $t^-, t^+ \in \mathbb{N}^n$. The i-th component of t^-, with $p_i \in P$, is defined to be $F(p, t)$, written $t^-(i) = t^-(p_i) := F(p_i, t)$. Similarly, $t^+(p) := F(t, p)$. The vector t^Δ captures the difference $t^\Delta := t^+ - t^-$.

The semantics of a Petri Net N is defined in terms of markings. A *marking* m is a vector in $\mathbb{N}^n$. Intuitively, it puts a number of *tokens* in each place. A marking is said to *enable* a transition t if $m(p) \geq t^-(p)$ for each place $p \in P$, written $m \geq t^-$. The set of all markings that enable t is called the *activation space* of t and denoted by $\mathrm{Act}(t)$. Note that $\mathrm{Act}(t) = \{t^- + v \mid v \geq 0\}$. If $m \in \mathrm{Act}(t)$, then t can be *fired*, resulting in the new marking $m' = m + t^\Delta$. This constitutes the *firing relation*, written as $m[t\rangle m'$. We lift the relation to sequences of transitions $\sigma = t_1 \dots t_k \in T^*$ where convenient, writing $m[\sigma\rangle m'$. A marking m_f

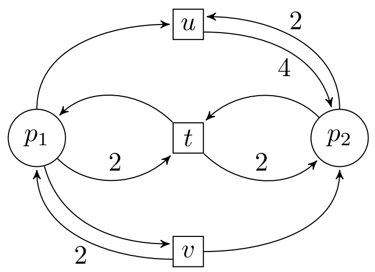


Figure 7.2: Petri Net with places p_1, p_2 and transitions u, t, v. The edges correspond to entries of the flow function F. We omit the label if it is 1.

is called *reachable* from a marking m_0 if there is a sequence of transitions σ such that $m_0[\sigma\rangle m_f$. We use $post^*(m_0)$ to denote the markings reachable from m_0 and $pre^*(m_f)$ are the markings from which m_f is reachable. A marking m_f is *coverable* from m_0, if there is a sequence of transitions σ and a marking $m \in \uparrow m_f$ such that $m_0[\sigma\rangle m$. The upward closure of m_f is $\uparrow m_f = \{m \in \mathbb{N}^n \mid m \geq m_f\}$.

(Inductive) Half Spaces. We describe sets of markings by means of half spaces. Let $N = (P, T, F)$ be a Petri net, $k \in \mathbb{Z}^n$, and $c \in \mathbb{Z}$. The *half space* defined by k and c is $\mathrm{Sol}(k, c) = \{m \in \mathbb{Z}^n \mid k \cdot m \geq c\}$. Here, $k \cdot m = \sum_{p \in P} k(p) \cdot m(p)$ is the usual scalar product. Note that we could also define half spaces via $k \cdot m \leq c$. This is of course equivalent since $k \cdot m \geq c$ if and only if $-k \cdot M \leq -c$. We are interested in half spaces that are inductive in the sense that they cannot be left by firing transitions. A half space (k, c) is *t-inductive* if for any $m \in \mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$ we have $m + t^{\Delta} \in \mathrm{Sol}(k, c)$. A half space (k, c) is *inductive* if it is t-inductive for all $t \in T$. We use IHS as a shorthand for inductive half space.

This means a half space (k, c) is not t-inductive iff there is a marking m in the half space $(k \cdot m \geq c)$, from which t can be fired $(m \geq t^-)$, and by firing t, the half space is left $(k \cdot (m + t^{\Delta}) < c)$. It holds $m \geq t^-$ iff there is a vector $x \in \mathbb{N}^S$ such that $m = t^- + x$. Together that gives us the following property that describes inductivity:

Theorem 15. *A half space (k, c) is t-inductive iff*

$$\nexists x \in \mathbb{N}^n : c \leq k \cdot x + k \cdot t^- < c - k \cdot t^{\Delta}$$

The formal proof is straightforward and given in Section D.1 of the appendix. This theorem provides us a way to disprove inductivity by finding a vector that satisfies two linear inequalities. It is a key insight behind our method for checking inductivity in Section 7.6

Example We provide some geometric intuition about the introduced notions. Consider the Petri net depicted in Figure 7.2. Focus on transition t. The vectors describing t are $t^- = (2, 1)$ (illustrated by the incoming edges of

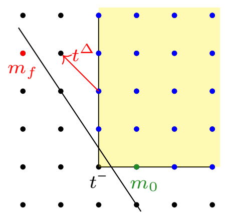


Figure 7.3: Geometric interpretation of (k, c) in $\mathbb{Z}^2$.

t), $t^+ = (1, 2)$ (the outgoing edges of t), and their difference $t^\Delta = (-1, 1)$. The activation space of t is the set $\mathrm{Act}(t) = \{(2, 1) + (x, y) \mid x, y \in \mathbb{N}\}$. It is visualized in Figure 7.3 by the yellow area. Let $m_0 = (3, 1)$ and $m_f = (0, 4)$ be two markings of the net. Consider the half space defined by $k = (3, 2)$ and $c = 9$. In Figure 7.3, it is indicated by the line $k \cdot x = c, x \in \mathbb{R}^2$: $\mathrm{Sol}(k, c)$ is the set of vectors that lie above the line. Clearly, $m_0 \in \mathrm{Sol}(k, c)$ and $m_f \notin \mathrm{Sol}(k, c)$ and thus the inequality is separating. The markings in $\mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$ are colored blue in Figure 7.3. The half space is t-inductive: if $m \in \mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$, firing t^Δ does not lead to a marking below the line. As we will see in the following section, (k, c) is also u and v-inductive. Hence, (k, c) is a proof for non-reachability of m_f from m_0.

Linear Safety Verification Our goal is to find inductive half spaces that disprove reachability or coverability. Formally, the first algorithmic problem we will study is LSV(R), *linear safety verification (reachability version)*:

> **Given:** A Petri net N and two markings m_0, m_f.
> **Question:** Is there an IHS (k, c) such that $m_0 \in \mathrm{Sol}(k, c)$ and $m_f \notin \mathrm{Sol}(k, c)$?

Problem LSV(C), *linear safety verification (coverability version)*, is defined by:

> **Given:** A Petri net N and two markings m_0, m_f.
> **Question:** Is there an IHS (k, c) such that $m_0 \in \mathrm{Sol}(k, c)$ and $\uparrow m_f \cap \mathrm{Sol}(k, c) = \varnothing$?

The reader familiar with separability will note that disproving reachability of m_f from m_0 amounts to finding a separator between $post^*(m_0)$ and $pre^*(m_f)$. A separator is a set $S \subseteq \mathbb{N}^n$ so that $post^*(m_0) \subseteq S$ and $S \cap pre^*(m_f) = \varnothing$. The difference between separability and the linear safety verification problem formulated above is that separators are neither required to be half spaces nor required to be inductive.

The choice for half spaces and inductivity is motivated by the constraint-based approach to safety verification that we pursue. Half spaces can be given in terms of (k, c), a format that is computable by a solver. Inductivity yields a local check for separation. Indeed, if (k, c) is inductive and $m_0 \in \mathrm{Sol}(k, c)$, we already have $post^*(m_0) \subseteq \mathrm{Sol}(k, c)$. Similarly, if (k, c) is inductive and $m_f \notin \mathrm{Sol}(k, c)$, then $pre^*(m_f) \cap \mathrm{Sol}(k, c) = \varnothing$. Hence, $\mathrm{Sol}(k, c)$ is indeed a separator. There are, however, separators that are neither half spaces nor inductive. To see the latter, consider a transition that is not enabled in $post^*(m_0)$ but happens to be enabled in the separator.

While reachability and coverability are decidable for Petri nets [100], decidability is still unknown for LSV(R) and LSV(C). The difficulty in linear safety verification is the check for inductivity. We tackle the problem by decomposing the set of half spaces into two classes. So-called trivial half spaces will always be inductive (and so the check is trivial). Non-trivial half spaces are the ones for which inductivity is non-trivial to check. Our main findings are necessary conditions for non-trivial half spaces to be inductive.

7.3 Trivial Half Spaces

We introduce three criteria for half spaces that, if satisfied, each guarantee inductivity. Intuitively, a half space is trivial if firing a transition always enters the half space or firing a transition in the half space either leads deeper into the half space or is not possible. It has to satisfy one of the three criteria. The idea for the criteria is due to [147, 162], we do not claim any novelty for this section.

The first case in which inductivity of a half space (k, c) wrt. a transition t is trivial is if k and t point in the same direction. More formally, the half space (k, c) is *oriented towards transition* t if $k \cdot t^{\Delta} \geq 0$. The scalar product gives information about the angle between the vectors k and t: if it is non-negative, the angle is at most 90 degrees. Hence, firing the transition moves the current marking away from the border of the half space. To give an example, the half space in Figure 7.3 is oriented towards the transitions u and v. It is not, however, oriented towards transition t. Indeed, firing that transition means moving closer to the border of the half space. It remains to argue that a half space which is oriented towards a transition t is t-inductive.

Lemma 10. *Let (k, c) be oriented towards t. If $m \in \mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$, then $m + t^{\Delta} \in \mathrm{Sol}(k, c)$.*

Proof. The lemma holds by $k \cdot (m + t^{\Delta}) = k \cdot m + k \cdot t^{\Delta} \geq k \cdot m \geq c$. The first equality is by distributivity of the scalar product, the following inequality is by the definition of orientation towards t, and the last inequality is by $m \in \mathrm{Sol}(k, c)$. Finally, $(m + t^{\Delta}) \in \mathrm{Sol}(k, c)$ follows from $k \cdot (m + t^{\Delta}) \geq c$ according to the definition of $\mathrm{Sol}(k, c)$. ☐

The lemma constitutes a first step in synthesizing inductive inequalities. If one wants to find a t-inductive inequality (k, c) that t-separates two markings m_0 and m_f and is oriented towards t, it will be a solution of the linear system

$$k \cdot m_0 \geq c \wedge k \cdot m_f < c \wedge k \cdot t^{\Delta} \geq 0.$$

For other inequalities we can, however, not hope for a similar result. There are half spaces that fail to be inductive for certain transitions. In fact, it turns out that deciding inductivity for a half space is *NP*-hard (see Section 7.6). As an example, consider the inequality from Figure 7.3 again. If we replace $c = 9$ with $c' = 8$, we get (k, c') which is not t-inductive. We have $k \cdot t^- = 8 = c'$ but $k \cdot (t^- + t^\Delta) = 7 < c'$. When firing t from marking t^-, we leave the solution space of (k, c') with $t^- \in \mathrm{Sol}(k, c')$.

Next, we consider inductivity of a half space (k, c) with respect to a transition t if $k \geq 0$ holds. This condition implies that $k \cdot m$ describes a monotone function. Increasing the values of m can only cause $k \cdot m \geq c$ to become or remain satisfied. Note that a violation of inductivity implies a value $k \cdot m$ that becomes smaller than c after a transition is fired. In this case, it suffices to check inductivity for the marking m in $\mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$ with the least value $k \cdot m$. Determining the least marking relative to unknown (k, c), however, does not admit a linear formulation. As a linear approximation, we require inductivity for t^-, the least marking in $\mathrm{Act}(t)$. Formally, we call (k, c) *monotone for t* if $k \geq 0$ and $k \cdot (t^- + t^\Delta) \geq c$ hold.

Lemma 11. *Let (k, c) be monotone for t. If $m \in \mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$, then $m + t^\Delta \in \mathrm{Sol}(k, c)$.*

Proof. Note that membership $m \in \mathrm{Act}(t)$ implies $m = t^- + v$ with $v \in \mathbb{N}^n$. We have $k \cdot (m + t^\Delta) = k \cdot (t^- + t^\Delta + v) = k \cdot (t^- + t^\Delta) + k \cdot v \geq c$. The first equality rearranges the terms of $m + t^\Delta$, the second is distributivity, the third is by the fact that $k \cdot (t^- + t^\Delta) \geq c$ and $k \cdot v \geq 0$. The former holds by the definition of monotonicity for half spaces, the latter is by the fact that $k, c, v \geq 0$. It holds $m + t^\Delta \in \mathrm{Sol}(k, c)$ since $k \cdot (m + t^\Delta) \geq c$. □

The last case in which it is easy to check (k, c) for being inductive wrt. t is that $k \leq 0$ and thus (k, c) is antitone in the marking. Now, if the half space (k, c) does not contain a marking that enables t, it is t-inductive. Not containing such a marking means $\mathrm{Act}(t) \cap \mathrm{Sol}(k, c) = \varnothing$. Since $k \leq 0$, this is equivalent to checking that t^-, the least vector in $\mathrm{Act}(t)$, does not satisfy $k \cdot t^- \geq c$. To sum up, we call (k, c) *antitone for t* if $k \leq 0$ and $k \cdot t^- < c$.

Lemma 12. *If (k, c) is antitone for t, then $\mathrm{Act}(t) \cap \mathrm{Sol}(k, c) = \varnothing$.*

Proof. Note that membership $m \in \mathrm{Act}(t)$ means $m = t^- + v$ for some $v \in \mathbb{N}^n$. Now $k \cdot m = k \cdot (t^- + v) = k \cdot t^- + k \cdot v < c$ follows. The inequality is by $k \cdot t^- < c$ and $k \cdot v \leq 0$. The former holds by the definition of antitone half spaces, the latter by $k \leq 0$ and $v \geq 0$. □

The following definition summarizes the three criteria developed above into the notion of trivial half spaces.

Definition 14. *Half space (k, c) is* trivial wrt. transition t *if (k, c) is oriented towards t, (k, c) is monotone for t, or (k, c) is antitone for t.*

We use the term trivial since the half spaces are trivially inductive. This is stated in the following theorem and follows directly from the above lemmas.

Theorem 16. *([162]) If (k, c) is trivial wrt. t, it is t-inductive.*

7.4 Non-Trivial Half Spaces

We now concentrate on half spaces that are inductive but not trivially so. This means the half spaces are neither oriented towards the transition of interest, nor monotone, nor antitone. By understanding these half spaces, we comprehend to which degree triviality is a restrictive proof principle. The first concern is this. If the half space is not oriented towards the transition, triviality requires $k \geq 0$ or $k \leq 0$. This means we could be missing inductive half spaces where some entries of k are positive while others are negative.

The main finding in this section is that such half spaces do not exist. Instead, triviality is complete in the following sense.

Theorem 17. *If (k, c) is t-inductive but not oriented towards t, then either $k \geq 0$ or $k \leq 0$ holds.*

Proof. The proof makes use of so-called *syzygies*, a concept well-known in commutative algebra [91]. Given a finitely generated module M over a ring R and a set $k_1, ..., k_n$ of generators, a syzygy of M is an element $(s_1, ..., s_n) \in R^n$ for which $s_1 k_1 + \cdots + s_n k_n = 0$ holds. For our purpose, the intuition behind a syzygy is that it is a vector that returns 0 if it is multiplied with k. This means we can add a syzygy to a marking without changing the scalar product of the marking and k.

We show that any half space (k, c) that is not oriented towards t and where $k(i) > 0$ and $k(j) < 0$ for some $i, j \in [1, n]$ cannot be t-inductive. Let (k, c) be such an inequality and $u \in \mathbb{Z}^n$ a vector in $\mathrm{Sol}(k, c)$. Note that such a vector always exists. We construct from u a vector v close to the border of the half space. Then the idea is to lift v to a marking m by adding non-negative syzygies. The resulting marking lies in $\mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$ and shows that (k, c) cannot be t-inductive by satisfying $m + t^{\Delta} \notin \mathrm{Sol}(k, c)$.

The vector v is defined by $v = u + \lfloor \frac{c - k \cdot u}{k \cdot t^{\Delta}} \rfloor \cdot t^{\Delta} \in \mathbb{Z}^n$. Since $k \cdot t^{\Delta} < 0$ by the fact that (k, c) is not oriented towards t, we get that $k \cdot v \geq c$:

$$k \cdot v = k \cdot u + \lfloor \frac{c - k \cdot u}{k \cdot t^{\Delta}} \rfloor \cdot k \cdot t^{\Delta} \geq c + c - k \cdot u - k \cdot t^{\Delta} \geq c.$$

The first equality is by the definition of v, the following inequality is by $\lfloor x \rfloor \geq x - 1$ and the last inequality is by $k \cdot u \geq c$ and $k \cdot t^{\Delta} < 0$. Similarly, we get $k \cdot (v + t^{\Delta}) < c$:

$$k \cdot (v + t^{\Delta}) \leq k \cdot u + \frac{c - k \cdot u}{k \cdot t^{\Delta}} \cdot k \cdot t^{\Delta} + k \cdot t^{\Delta} = c + k \cdot t^{\Delta} < c.$$

The first inequality is by $\lfloor x \rfloor \leq x$ and the definition of v, the last inequality is by $k \cdot t^{\Delta} < 0$. This means it holds $v \in \mathrm{Sol}(k, c)$ and $v + t^{\Delta} \notin \mathrm{Sol}(k, c)$. However, this is not yet a counter example to t-inductivity. We cannot ensure that v is a marking that activates transition t.

Such a marking can be constructed by adding syzygies to v. For $p \in P$, let e_p denote the p-th unit vector. For any $p \in P$, we construct a syzygy s_p, depending on $k(p)$. If $k(p) > 0$, we set $s_p = -k(j) \cdot e_p + k(p) \cdot e_j$. If $k(p) < 0$, we define

$s_p = -k(p) \cdot e_i + k(i) \cdot e_p$. For the case $k(p) = 0$, we simply set $s_p = e_p$. Note that all vectors s_p are elements of $\mathbb{N}^n$ and each satisfies $k \cdot s_p = 0$.

The syzygies s_p allow for adding non-negative values to each component of v without changing the scalar product with k. It is easy to see that this is true: Assume $k(p) > 0$, for any vector v and constant μ holds

$$k \cdot (v + \mu \cdot s_p) = k \cdot v - \mu \cdot k(p) \cdot k(j) + \mu \cdot k(j) \cdot k(p) = k \cdot v.$$

The argument for $k(p) < 0$ is analogue. Hence, there exist $\mu_p \in \mathbb{N}$ such that $v + \sum_{p \in P} \mu_p \cdot s_p \geq t^-$. If we set $m = v + \sum_{p \in P} \mu_p \cdot s_p$, we obtain a marking $m \in \mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$ such that $m + t^\Delta \notin \mathrm{Sol}(k, c)$. □

An alternative proof of the theorem without syzygies is given in Section D.4 of the appendix. The theorem does not state that triviality captures all inductive half spaces. In the monotone case, we still approximate the least element in $\mathrm{Act}(t) \cap \mathrm{Sol}(k, c)$ by t^-. But except for that, we are precise.

Non-trivial inequalities do not admit a simple structure like trivial inequalities do. In fact, we cannot even hope for a linear constraint system of polynomial size, describing the space of t-inductive non-trivial inequalities. The reason is as follows: if such a system would exist, call it $L(k, c)$, we could decide t-inductivity for each given half space (k, c) in polynomial time. An algorithm for the problem would first test whether (k, c) is trivial or non-trivial. In the first case, t-inductivity would be immediate. In the latter case, the algorithm would plug (k, c) into the system $L(k, c)$ and evaluate. Depending on the evaluation, (k, c) would be t-inductive or not. All the described steps can be carried out in polynomial time. Hence, this contradicts the *NP*-hardness of the problem (in binary encoded input) which we will prove in Section 7.6.

However, we can give a close approximation by using necessary conditions derived from structural properties. A first step towards this is the following lemma. It combines Theorem 17 with the notion of non -trivial inequalities.

Lemma 13. *Any half space (k, c) that is t-inductive and non-trivial for t satisfies either (a) $k \geq 0$ and $k \cdot t^- < c - k \cdot t^\Delta$ or (b) $k \leq 0$ and $k \cdot t^- \geq c$.*

Note that Theorem 17 is essential for proving the lemma. The lemma provides geometric intuition to separate non-trivial from trivial inequalities. In the non-trivial case, the set $\mathrm{Sol}(k, t) \cap \mathrm{Act}(t)$ is a strict non-empty subset of the activation space $\mathrm{Act}(t)$. This stands in contrast to trivial inequalities, where $\mathrm{Sol}(k, t) \cap \mathrm{Act}(t)$ is either empty or (nearly) the whole activation space. Moreover, Lemma 13 is a tool for proving the following necessary criterion.

Lemma 14. *Let (k, c) be non-trivial for t and t-inductive. For any element $k(i)$ of k with $|k(i)|$ denoting its absolute value holds*

$$k(i) = 0 \ \vee \ |k(i)| \geq -k \cdot t^\Delta. \tag{7.1}$$

Proof. Assume towards contradiction that there is a $k(i)$ with $0 < |k(i)| \leq -k \cdot t^\Delta$ Since it is non-trivial, we can apply Lemma 13 and either (a) or (b) holds. For both cases, we define a value z such that $c \leq k \cdot t^- + z \cdot k(i) < c - k \cdot t^\Delta$ is satisfied.

Case (a) $k \cdot t^{-} < c - k \cdot t^{\Delta}$ and $k \geq 0$: There is a smallest $z \in \mathbb{N}$ such that $c \leq k \cdot t^{-} + z \cdot k(i)$. If $k \cdot t^{-} \geq c$, then $z = 0$ holds trivially. Note that $k \cdot t^{-} + z \cdot k(i) = k \cdot t^{-} < c - k \cdot t^{\Delta}$ holds by (a). If $k \cdot t^{-} < c$, then z exists since $k(i) > 0$. Assume towards contradiction that $c - k \cdot t^{\Delta} \leq k \cdot t^{-} + z \cdot k(i)$. It holds

$$k \cdot t^{-} + (z - 1) \cdot k(i) \geq c - k \cdot t^{\Delta} - k(i) \geq c.$$

The first inequality is by the assumption $c - k \cdot t^{\Delta} \leq k \cdot t^{-} + z \cdot k(i)$ and the second is by $|k(i)| \leq -k \cdot t^{\Delta}$. This means $z - 1$ also satisfies the condition which is a contradiction to z being smallest. The condition $c \leq k \cdot t^{-} + z \cdot k(i) < c - k \cdot t^{\Delta}$ is satisfied either way.

Case (b) $k \cdot t^{-} \geq c$ and $k \leq 0$: There is a smallest $z \in \mathbb{N}$ such that $k \cdot t^{-} + z \cdot k(i) < c - k \cdot t^{\Delta}$. Since z is smallest and $|k(i)| \leq -k \cdot t^{\Delta}$ holds it follows $c \leq k \cdot t^{-} + z \cdot k(i) < c - k \cdot t^{\Delta}$ as well. The detailed proof is omitted since it is analogue to (a).

We define the vector $x \in \mathbb{N}^n$ as $x = z \cdot e_i$. According to Theorem 15, this implies that (k, c) is not t-inductive which is a contradiction. □

7.5 Generating Invariants

We now combine the collected conditions in order to categorize the inductive inequalities. We use these properties to construct an SMT-formula that consists of linear constraints and gives a good over-approximation of all k that can form inductive invariants. In order to keep the constraints in the formula linear, we only generate the vector k. This means we cannot directly apply properties that require c. However, we can apply them indirectly using bounds of c. Recall that we want a separating invariant, i.e. m_0 should satisfy the inequality but not m_f. It follows $k \cdot m_0 \geq c$ and $k \cdot m_f < c$ and thus $k \cdot m_0 > k \cdot m_f$ holds. This gives us a lower and upper bound of c. We apply these bounds of c to the definition of trivial inequalities and derive the following conditions.

$$k \cdot m_0 > k \cdot m_f \tag{0}$$

$$k \cdot t^{\Delta} \geq 0 \tag{1}$$

$$k \leq 0 \ \wedge \ k \cdot t^{-} < k \cdot m_0 \tag{2}$$

$$k \geq 0 \ \wedge \ k \cdot t^{-} > k \cdot m_f - k \cdot t^{\Delta} \tag{3}$$

Each single condition (1)-(3) together with (0) ensures trivial t-inductivity for some values of c within the limits. If none of these condition hold, then we get non-trivial inequalities. In this case Theorem 17 applies:

$$k \geq 0 \vee k \leq 0. \tag{4}$$

We can also apply Lemma 14 directly and add Condition (5):

$$k(i) = 0 \ \vee \ |k(i)| \geq -k \cdot t^{\Delta}. \tag{5}$$

We collect the linear constraints in the SMT-formula ϕ_t that is satisfied by exactly the vectors k that form a separating t-inductive inequality with some c:

$$\phi_t \coloneqq (0) \wedge ((1) \vee (2) \vee (3) \vee ((4) \wedge (5))).$$

Finally we construct the input formula $\phi \coloneqq \bigwedge_{t \in T} \phi_t$ for the CEGAR loop outlined in Fig. 7.1. Each satisfying assignment consists of a vector k that could be extended to a IHS with a constant c.

As discussed in the previous section, it is not possible to construct a linear constraint system of polynomial size that describes precisely all IHS. However, we give a close approximation. By utilizing the derived structural properties, we obtain a formula containing linear constraints. Its solutions are precisely the vectors k that can form t-inductive inequalities.

Lemma 15. *There is a $c \in \mathbb{Z}$ such that (k, c) is a separating t-inductive inequality iff k is a solution of ϕ_t.*

The proof of this lemma is technical but rather straightforward, it is given in Section D.2 of the appendix. So for each solution k of ϕ, there are values c for every transition t such that (k, c) is t-inductive. The problem is that these c may be different for each transition. The challenge is to find a common value c. For this reason, we prefer solutions k of ϕ_t that allow for many values of c. We prioritize condition (1) where (k, c) is t-inductive for every value of c.

If a vector k generated by ϕcannot form a IHS, we can exclude not only the vector k but all multiples of k in future iterations. We archive this by updating the formula $\phi_t \coloneqq \phi_t \wedge \neg mul(k)$.

Lemma 16. *Let $k' \coloneqq a \cdot k$ with $a \in \mathbb{N}$. If (k', c) is an IHS, then so is $(k, \lceil \frac{c}{a} \rceil)$.*

Proof. We use contraposition. Given a marking m that violates t-inductivity of $(k, \lceil \frac{c}{a} \rceil)$. We show that it violates t-inductivity of (k', c) as well.

If it holds $k \cdot m \geq \lceil \frac{c}{a} \rceil$, then it follows $k \cdot m \geq \frac{c}{a}$ from $\lceil \frac{c}{a} \rceil \geq \frac{c}{a}$ and thus $k' \cdot m \geq c$. If it holds $k \cdot (m + t^{\Delta}) < \lceil \frac{c}{a} \rceil$ then it follows $k \cdot (m + t^{\Delta}) + 1 \leq \lceil \frac{c}{a} \rceil$. Finally, we conclude $k' \cdot (m + t^{\Delta}) < c$ using $\lceil \frac{c}{a} \rceil \leq \frac{c}{a} + 1$. □

The above lemma shows that our method of excluding multiples does not forbid vectors that may form an IHS. If a multiple $k' = a \cdot k$ of k could form an IHS (k', c), then k would have already formed an IHS $(k, \lceil \frac{c}{a} \rceil)$.

The presented approach generates IHS. In order for the method to be a semi-decider for $\mathsf{LSV}(\mathsf{R})$, we need to additionally ensure that any candidate k is generated by the SMT solver at some point. This is achieved by adding a constraint that imposes a bound on the absolute values of k. If the formula becomes unsatisfiable, the bound is increased. It remains to show how our semi-decider for $\mathsf{LSV}(\mathsf{R})$ can be adapted for $\mathsf{LSV}(\mathsf{C})$.

7.5.1 Coverability

Note that Condition (0) ensures the existence of a value c such that m_0 satisfies the inequality while m_f does not. When we check for coverability, we need to additionally ensure that all markings m with $m \in m_f\uparrow$ do not satisfy the inequality, meaning $m_f\uparrow \cap \mathrm{Sol}(k,c) = \emptyset$.

Theorem 18. *Let (k,c) be such that $m_f \notin \mathrm{Sol}(k,c)$. It holds $m_f\uparrow \cap \mathrm{Sol}(k,c) = \emptyset$ iff $k \leq 0$.*

Proof. "$\Rightarrow$:" Let $m_f\uparrow \cap \mathrm{Sol}(k,c) = \emptyset$. We assume towards contradiction that $k(i) > 0$ holds for some $i \leq n$. Let m be such that $m(i) = m_f(i) + (c - k \cdot m_f)$ and $m(j) = m_f(j)$ for all $j \leq n$ with $j \neq i$. Then $k \cdot m = k \cdot m_f + k(i) \cdot (c - k \cdot m_f)$. Since $k(i) > 0$ and $(c - k \cdot m_f) > 0$, it follows $k \cdot m \geq k \cdot m_f + 1 \cdot (c - k \cdot m_f) = k \cdot m_f + c - k \cdot m_f = c \geq c$ and thus $m \in m_f\uparrow \cap \mathrm{Sol}(k,c) \neq \emptyset$. This is a contradiction to $m_f\uparrow \cap \mathrm{Sol}(k,c) = \emptyset$.
"$\Leftarrow$:" Let $k \leq 0$ and $m \in m_f\uparrow$ and thus $m \geq m_f$. It follows $k \cdot m \leq k \cdot m_f < c$ and thus $m \notin \mathrm{Sol}(k,c)$. This means $m_f\uparrow \cap \mathrm{Sol}(k,c) = \emptyset$. □

We can now adapt our CEGAR loop for LSV(C) by applying the above theorem. We add a constraint $k \leq 0$ (6) to the formula $\phi := \phi \wedge (6)$. With this additional condition, we can assure that any any IHS synthesized by our algorithm contains no marking in $m_f\uparrow$ and thus proves that m_f is not coverable.

We will show how to generate a suitable c for a given k in the following section.

7.6 Checking Inductivity

We examine non-trivial half spaces and introduce a procedure that decides t-inductivity of a given half space (k,c). We can then generalize this technique to compute all values of c that form an IHS with a given vector k. We present Algorithm 7 which iterates over values of $k \cdot m$ where $m \in ACT(t)$. If the half space is not t-inductive, then there is a marking $m \in ACT(t)$ that satisfies the half space and $m + t^{\Delta}$ does not. Such a value $k \cdot m$ is located in the *targeted interval* from c to $c - k \cdot t^{\Delta} - 1$. Let $K := \{k(i) \mid i \leq n\}$ be the set of values that occur in vector k. We examine sequences over these values $k_1..k_l \in K^*$ where K^* is the Kleene Star. It is important to note that k_i does not denote the position in the vector (which is written $k(i)$) but rather the position in the given sequence over the values of k. We apply the following theorem and represent the product $k \cdot m$ with the corresponding sum $k \cdot t^- + \sum_{i=1}^{l} k_i$.

Theorem 19. *A half space (k,c) is t-inductive iff*

$$\nexists k_1..k_l \in K^* : c \leq k \cdot t^- + \sum_{i=1}^{l} k_i < c - k \cdot t^{\Delta}$$

This condition of t-inductivity follows from Theorem 15. The formal proof is given in Section D.3 of the appendix.

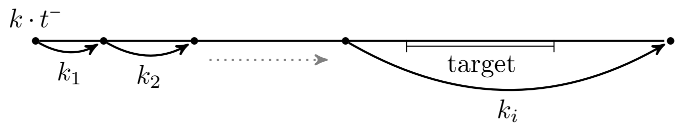


Figure 7.4: This is an example run of the algorithm. Here, the starting point $k \cdot t^-$ is less than c and k is positive. The algorithm adds $k_1, k_2, .., k_i$ until it overshoots the target interval. Since every value is reached at most once, the run-time is polynomial in the input values (See Theorem 24 of the appendix).

Recall that for non-trivial IHS, k cannot contain both positive and negative entries. The algorithm starts at $k \cdot t^-$ and adds values of k until it either has found a value that violates Theorem 19 or it can confirm that none exists. The half space is not t-inductive if the algorithm has "passed" the target interval. The algorithm employs dynamic programming in order to iteratively compute values of partial sums. The partial sum, which is given implicitly by the computation of the algorithm, is extended in a directed way towards the target interval. If a reached value is lower than the target interval, then at least one value has yet to be added. Since addition is commutative, it is sufficient to add the value in the current step: A sketch of an example run of the algorithm is given in Figure 7.4.

We prove that the algorithm is both correct (Lemma 29) and complete (Lemma 30) in Section D.3 of the appendix.

Theorem 20. *Deciding inductivity of a half space is co-NP complete.*

Proof. To show that the problem is in co-NP, we solve it by non-deterministically guessing a vector x that violates Theorem 15. We utilize Theorem 25 in Section D.3 of the appendix which shows that we only have to consider values in a polynomial bound. We reduce the complement of the unbounded subset sum problem to inductivity in order to prove co-NP hardness.

The *Unbounded subset sum problem* [83] is defined as follows: Given $w_1, ..w_n, d \in \mathbb{N}$, are there $x_1, .., x_n \in \mathbb{N}$ such that $\sum_{i=1}^{n} x_i \cdot w_i = d$?

The main difference to deciding t-inductivity is that in inductivity, we have a targeted interval and in the unbounded subset sum problem, we have a targeted value d. The idea is to choose such a value t^{Δ} such that the targeted interval becomes so small that only one value in it can be reached.

We set $k(i) := w_i$ for all $i \leq n$. The *greatest common divisor* (gcd) of an vector k with either $k \geq 0$ or $k \leq 0$ is defined as $\gcd(k) := \gcd(k(1), \dots, k(n))$. Note that the gcd is ≤ 0 if $k \leq 0$. We calculate $\gcd(k)$ as well as a vector $a \in \mathbb{Z}^n$ such that $k \cdot a = \gcd(k)$. This is done in polynomial time using the Euclidean method. We construct the uniquely defined smallest Petri net N with a single transition t such that $t^{\Delta} = -a$. We set $c := d + k \cdot t^-$. If d is not a multiple of gcd then the unbounded subset sum instance is not valid since every sum $\sum_{i=1}^{n} x_i \cdot w_i$ must also be a multiple of gcd. This can be checked in polynomial time, so we

Algorithm 7: Inductivity

```
queue.add($k \cdot t^-$);
reached[$k \cdot t^-$]:= True;
repeat
    $current$:=queue.remove();
    if $c \le current < c - k \cdot t^\Delta$ then
        return Not inductive;
    end
    for $k \in K$ do
        if $(current + k < c - k \cdot t^\Delta \wedge k \ge 0)$
        $\vee(current + k \ge c \wedge k \le 0)$ then
            if $\neg reached[current + k]$ then
                queue.add($current + k$);
                reached[$current + k$]:=True
            end
        end
    end
until $queue.isEmpty$;
return Inductive
```

can assume that d is a multiple of gcd. If this constructed half space is inductive, then we can apply Theorem 19 and it follows:

$$\nexists k_1..k_l \in K^* : c \le k \cdot t^- + \sum_{i=1}^{l} k_i < c - k \cdot t^\Delta$$

$$\Leftrightarrow \nexists k_1..k_l \in K^* : d \le \sum_{i=1}^{l} k_i < d - k \cdot t^\Delta$$

If the constructed half space is not inductive, then it follows:

$$\exists k_1..k_l \in K^* : c \le k \cdot t^- + \sum_{i=1}^{l} k_i < c - k \cdot t^\Delta$$

$$\Leftrightarrow \exists k_1..k_l \in K^* : d \le \sum_{i=1}^{l} k_i < d - k \cdot t^\Delta$$

Since d is a multiple of gcd and $-k \cdot t^\Delta = \text{gcd}$, it holds that there is no number $z \in \mathbb{Z}$ such that $d < z < d - k \cdot t^\Delta$ where z is a multiple of gcd. This means $d = \sum_{i=1}^{l} k_i$ follows from $d \le \sum_{i=1}^{l} k_i < d - k \cdot t^\Delta$. We conclude that that the reduction is indeed correct. □

From co-NP hardness and Theorem 24 of the appendix follows that our algorithm is optimal. In Section D.3 of the appendix, we perform a more comprehensive analysis of the complexity of the decision problem. We will present an alternative algorithm and then show that the problem is in **L** for unary encoded input and fixed dimension of k, in **co-NL** for unary encoded input, in **co-CSL** for binary encoded input, and that it is fixed parameter tractable.

7.6.1 Generating the Constant

It is easy to see how this algorithm can be generalized to compute all values c such that the half space is t-inductive. In this case we do not know where the targeted interval is. Instead, we simply compute the reached values and look for any gap between them that is large enough to contain a targeted interval. If it is, then we have found values for c. The question is now whether the procedure terminates and if so, whether it does so in a certain time limit. Here, we use the so-called Frobenius number [44].

Theorem 21 (Frobenius Number [44])**.** *Let $a_1, \ldots, a_n \in \mathbb{N}$ such that $\gcd(a_1, \ldots, a_n) = 1$ and let a_{max}, a_{min} denote elements of $a_1 \ldots a_n$ with maximal, respectively minimal, absolute value. The largest integer that cannot be represented as a positive linear combination of the a_i is called the* Frobenius number. *It exists and is bounded by $(a_{max} - 1) \cdot (a_{min} - 1)$.*

The proof is given in [44]. Intuitively, this means that at a certain point, our algorithm reaches every multiple value of $\gcd(k) = 1$ and because of that there are no more gaps between reached values that are large enough to contain a targeted interval. As a consequence of the these considerations, we get a new bound on the possible values of c for a non-trivial t-inductive half space. This bound is independent of the Frobenius number and the greatest common divisor.

Theorem 22. *Let $k \cdot m \geq c$ be a non-trivial t-inductive half space and let k_{max}, k_{min} denote entries of k with maximal and minimal absolute values.*

1. *If $k \geq 0$, then it holds that $c < k_{max} \cdot k_{min} + k \cdot t^-$.*
2. *If $k \leq 0$, then it holds that $c \geq -k_{max} \cdot k_{min} + k \cdot t^-$.*

The formal proof is given in Section D.3.1 of the appendix. We can terminate the adapted version of Algorithm 7 as soon as the corresponding bound according to Theorem 22 is reached. Since the bound is polynomial in k, this means our complexity results still apply for the generalized algorithm.

Given a vector k, we can apply the general algorithm for all transitions and check if there is a value c that is in the set of solutions for every transition and that satisfies $k \cdot m_0 \geq c > k \cdot m_f$.

Given a transition t, our technique generates a set of possible values c such that (k, c) is t-inductive. If the intersection of these sets of possible c is not empty, each value c in it gives an inductive half space (k, t). Instead of running the algorithm individually for each transition, our implementation speed up this technique. We are running the method for all transitions in an interleaved manner so that we process the same range of reached values for each transition.

We stop as soon as we have either found a c that is suitable for each transition or when there are no more possible solutions for one of the transitions. The run-time of the algorithm is polynomial in k.

7.7 Experiments

We implemented the CEGAR method in a Java prototype tool called INEQUALIZER[2] which uses Z3 [65] as the back-end SMT solver. Before the tool executes the CEGAR loop, it uses an SMT-query to check whether there is a separating inductive invariant that is trivial for all transitions.

The INEQUALIZER supports two features of the SMT solver, incremental solving and minimization. In incremental solving, the SMT-solver reuses information learned by previous queries in earlier iterations. We use minimization to guide the CEGAR search towards more likely candidates. The idea here is that in the non-trivial case, there may only few suitable value c that ensure t-inductivity. So we prefer an inequality that is trivial w.r.t. as many transitions as possible. We assign the value 0 for every transition where the assignment k only results in trivial inequalities and 1 for the non-trivial ones. Then we request a solution for which the sum over these values is minimal.

We execute the tool on the example given in Fig. 7.2. We find that there are no trivial separating IHS. Using incremental solving, the algorithm takes 3 iterations and returns the non-trivial separating IHS $k = (53, 52)$, $c = 209$. If we use minimization, we only require 2 iterations and get the non-trivial separating IHS $k = (8, 5)$, $c = 22$. In both cases, the running time is under a second. The difference in iterations is due to the fact that we expect minimization to choose possible vectors k that are trivial for more transitions which should increase the chance to find a suitable c. On the other hand, we expect incremental solving to improve the solving time in executions with more iterations.

We evaluate INEQUALIZER for coverability checking on a benchmark suite and compare it with other methods of coverability checking implemented by the tool MIST [135, 86, 82, 81, 66]. The results are given in Table 7.1. The listed Petri nets are all save: the unsafe markings are not coverable. The experiments were performed on a 1,7 GHz Intel Core i7 with 8GB memory. The execution times are given in seconds. For entries marked as T/O, the timeout was reached. Except for the mutual exclusion nets PETERSSON and LAMPORT, our tool found a separating IHS in every case. Each time there was a trivial IHS, the CEGAR loop was never used. We suspect that the two cases where INEQUALIZER timed out are negative instances of $\mathsf{LSV(C)}$ i.e. there is no separating IHS (even though it is not coverable). The execution times are similar to MIST although INEQUALIZER has a small overhead resulting from generating the SMT-query.

[2]`https://github.com/florianfurbach/Inequalizer`

Benchmark	$\lvert P\rvert$	$\lvert T\rvert$	INEQUALIZER	MIST				
				backward	ic4pn	tsi	eec	eec-cegar
BASICME	5	4	0.6	0.1	0.1	0.1	0.1	0.1
KANBAN	16	14	0.7	0.1	0.2	0.8	0.1	0.2
LAMPORT	11	9	T/O	0.1	0.1	0.1	0.1	0.1
MANUFACTURING	13	6	0.6	1.9	0.1	0.1	0.1	0.1
PETERSSON	14	12	T/O	0.2	0.1	0.1	0.1	0.1
READ-WRITE	13	9	0.5	0.1	1	0.1	0.9	0.5
MESH2X2	32	32	1.2	0.3	0.1	48.6	0.8	0.2
MESH3X2	52	54	2.1	2.2	0.2	T/O	T/O	2.2
MULTIPOOL	18	21	0.8	0.3	2	2.2	1	2.3

Table 7.1: INEQUALIZER vs. MIST.

7.7.1 Non-Trivial Petri Nets

Since the benchmark suite did not require non-trivial separating IHS, it did not accurately present our CEGAR method. We would like a better understanding of which Petri nets require non-trivial separating IHS. For this purpose, we construct a simple Petri net that has a non-trivial separating IHS but not a trivial one. We begin by collecting sufficient conditions of a Petri net that ensure non-triviality for any separating IHS.

Lemma 17. *Let $a \in \mathbb{R}_+^m$ be such that $m_f = m_0 + \sum_{i=1}^m a(i) \cdot t_i^\Delta$. For any separating inequality, there is a transition that is not oriented towards it.*

Proof. Since the inequality is separating, it holds $k \cdot m_0 \geq c > k \cdot m_f$ and thus $0 > k \cdot (m_f - m_0) = \sum_{i=1}^m a(i) \cdot k \cdot t_i^\Delta$. One element in the sum has to be negative: $\exists_{i<m} : a(i) \cdot k \cdot t_i^\Delta < 0$. Since $a(i)$ can not be negative, it follows $\exists_{i<m} : k \cdot t_i^\Delta < 0$. So (k, c) is not oriented towards t_i. $\square$

It follows that, for any separating inequality, one of the transitions t_i with an associated value a_i greater than zero is not oriented towards it. In order to ensure that the separating inequality is not trivial, we require two additional properties: it can neither be antitone, nor monotone.

For any t_i with $a_i > 0$ we require $t_i^- \leq m_0$, i.e. the transition is activated in the initial marking. This means $\mathrm{Act}(t) \cap \mathrm{Sol}(k, c) \neq \varnothing$ and thus it is not antitone.

For any t_i with $a_i > 0$ we require $t_i^- + t_i^\Delta \leq m_f$, which means there is a marking from which t_i can be fired in order to reach m_f. Assume there is separating inequality (k, c) that is monotone for t_i. Then it holds $k \geq 0$ and thus $k \cdot m_f \geq k \cdot (t_i^- + t_i^\Delta)$. Since $k \cdot m_f < c$, it follows $k \cdot (t_i^- + t_i^\Delta) < c$. This is a contradiction to monotonicity for t_i.

In summary, if the marking equation has a continuous solution such that the used transitions can all be fired from the initial marking and they can all be fired to reach m_f, then there are no trivial separating inequalities.

This sufficient condition for non-triviality is useful, because it is not much stronger than the following necessary condition for unreachability. If no solution of the marking equation exists where at least one used transition can be fired in the beginning and one in the end, then m_f is unreachable. This condition is

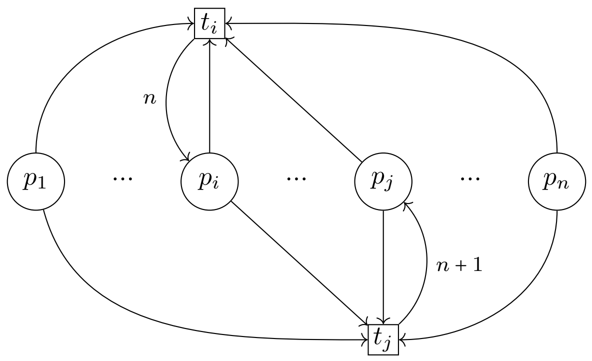


Figure 7.5: A non-trivial Petri net

very easy to check. This comparison suggests that for a Petri net where it is not immediately obvious that m_f is unreachable, a non-trivial inequality is likely to be required.

Example We now construct a minimal non-trivial example for larger dimensions. We introduce a Petri net N_n of size $n \geq 3$ that has a non-trivial separating inequality but no trivial separating inequalities (see Fig. 7.5). Furthermore, if one transition is removed, a trivial separating invariant exists.

We set $m_0 = 1^n, m_f = 2^n$, meaning we start with one token in each place and ask whether we can avoid getting having tokens in each place. Then, we choose some $j \leq n$ and we define n transition $t_1, ...t_n$ such that each transition t_i removes one token from each place and then puts n tokens in place p_i. The exception is t_j which puts $n+1$ tokens into p_j.

Formally, this means $t_i^- = 1^n$ for all $i \leq n$ and $t_i^\Delta(i) = n-1, t_i^\Delta(k) = -1$ for $k \leq n, k \neq i$. For t_j, it holds $t_j^\Delta(i) = -1$ for $i \leq n, i \neq j$ and $t_j^\Delta(j) = n$.

Lemma 18. *The Petri net N_n has a separating non-trivial IHS but no separating trivial IHS.*

Proof. Let $c = -n \cdot (n+1)$, $k(j) = -n$, and $k(i) = -(n+1)$ for $i \neq j, i \leq n$. The inequality is separating, since it holds $k \cdot m_0 = -(n+1) \cdot (n-1) - n = -n \cdot (n+1) + 1 > c$ and $k \cdot m_f = -(n+1) \cdot (n-1) \cdot 2 - 2n = -2(n+1) \cdot n + 2 < c$.

We show that the half space is a IHS using Theorem 19. Since $k \cdot t_j^\Delta = (n+1) \cdot (n-1) - n^2 = -1$ it holds $k \cdot t_j^- = k \cdot m_0 \geq c - k \cdot t_j^\Delta$. If we add any k_i then we get $k \cdot t_j^- + k_i = -n \cdot (n+1) + 1 - n < c$ and if we add k_j or additional values of k we get an even smaller value.

Let (k, c) be any separating inequality. Since $k \cdot m_0 > k \cdot m_f$, it holds $-k_1 \ldots -k_n > 0$. Let $k_l = min(k_1, \ldots k_n)$ be the negative entry of k with the largest absolute value. If $l = j$, then $k \cdot t_l^\Delta = -k_1 ... -k_n + n \cdot k_l + k_l \leq -n \cdot k_l + n \cdot k_l + k_l = k_l < 0$.

$\lvert P\rvert$	Iterations	Time
3	2	0.4
4	113	6.9
5	2	0.4
6	2	0.4
7	6	0.6
8	3	0.6
9	378	205.2
10	2	0.5

Table 7.2: Inequalizer on non-trivial Petri nets.

If $l \neq j$, then $k \cdot t_l = -k_1 ... - k_n + n \cdot k_l = k_l$. If all entries of k are not equal, then it holds $-k_1 + ... - k_n > n \cdot k_l$ and thus $k \cdot t_i < 0$. If all entries of k are equal then they are also negative and it holds

$$k \cdot t_j = -k_1 ... - k_n + n \cdot k_j + k_j = -n \cdot k_j + n \cdot k_j + k_j = k_j < 0.$$

It follows that any separating inequality is oriented towards at least one transition.

Obviously all transitions are enabled at m_0 and the inequality is not antitone. According to $-k_1 \ldots - k_n > 0$, it holds $k \ngeq 0$ and thus it is not monotone either. □

We evaluate the performance of Inequalizer for reachability on the non-trivial Petri nets of sizes three to ten in Table 7.2. We give the number of iterations of the CEGIS loop performed by the tool. We do not include the run-time results of Mist for these Petri nets in the table since they were all well below 0.1 second. We use incremental solving and find that our tool usually computes the IHS quickly using few iterations. There are only two diverging results where the SMT-solver returns a number of unusable vectors k.

7.8 Conclusion and Outlook

Rather than under-approximating the behavior of a program as we did in earlier chapters, we used a complementary approach here. We examined Petri nets which are well suited for modeling concurrent programs and over-approximate their reachable markings with inductive invariants. We extensively studied properties of inductive linear inequalities as well as the decision problem whether an inequality is inductive. This culminates in a counter example guided refinement method (CEGAR) which attempts to solve linear safety verification (LSV) both for reachability and coverability. Here, we use again our approach of SMT-queries combined with a good heuristic in order to achieve practical execution times when solving complex problems. This is implemented in the prototype tool Inequalizer and evaluated experimentally.

We expect that further study of inductive invariants, especially their structural properties will prove useful in improving Inequalizer. In particular, we hope this provides us with options to either reduce the size of the search space or eliminate more vectors than just multiples of a counter example k discovered in the CEGAR loop.

One possible avenue of reducing the search space is using S-invariants. We say a t-inductive half space (k, c) is an *S-invariant for t* iff it holds $k \cdot t^{\Delta} = 0$. Geometrically, this means the vectors k and t^{Δ} are orthogonal to each other and thus t^{Δ} is parallel to the border of the half space. We suspect that the following holds: For any non-trivial t-inductive half space, there is a t-inductive S-invariant (k', c') for t such that

$$\mathrm{Act}(t) \cap \mathrm{Sol}(k, c) = \mathrm{Act}(t) \cap \mathrm{Sol}(k', c').$$

We implemented a procedure that tests the S-invariant conjecture on given examples. To show that the sets are equal, we have to test two inclusions. These can be described by integer programs. We linked our tool with the solver *lp_ solve*[3] to test feasibility of the problems which allowed us to test the conjecture on many examples. We did not find a counter example yet. This suggests that our method might be adapted such that it only examines certain S-invariants in the case of non-triviality.

[3]`https://sourceforge.net/projects/lpsolve/`

Chapter 8

Conclusion

We examined verification under weak memory. For multiple problems, we studied the complexity and gave practical solutions. Beginning by solving the testing problem, we kept expanding our approaches and building on them in order to be able to solve harder problems. Practically, this culminated in the tool suite Dat3M. It consists of the modules Dartagnan, Porthos, and Aramis which solve bounded reachability, portability and memory model synthesis respectively.

We began with the testing problem under weak memory models described using serial views. We showed that even though the execution is already partially known, the problem is still **NP**-complete for almost all practical restricted classes of inputs. We gave a reduction of the problem to SAT.

We expanded on this reduction in order to handle acyclic programs. Instead of serial views, we adapt the encoding to memory models in the more flexible `cat` language as input. We developed different novel techniques to drastically reduce the size of the encoding and implemented the method in the tool Dartagnan.

We added to these encoding techniques the ability to express inconsistency with a memory model and equivalence of executions. We use this to solve trace portability. From trace portability, which we show to be co-**NP**-complete, we move on to the harder state portability, which we show to be Π_2^P-complete. Here, we utilize the encoding for trace portability to generate useful candidates for counterexample executions in a guided search. This method provides an optimal solution for state portability with great experimental results. This is implemented in Porthos.

We use this general concept of a guided search with SMT queries to synthesize memory models. Here, we introduce the concept of template relations which allows us to use SMT queries to generate relations that are cyclic for certain executions and acyclic for all executions of certain litmus tests. We couple this with a back-tracking search in order to combine constraints over the generated relations to candidate memory models.

Finally we use invariants in order over-approximate the behavior of concurrent programs. Recall that in contrast, we used BMC in Dartagnan to under-approximate it. We again employ a guided search that uses SMT queries as a back-end in order to generate useful invariant candidates.

What started out as a complexity study evolved into a set of techniques implemented in the tool suite Dat3M. Our methods proved to be very applicable to solving tasks in the area of verification under weak memory models. Thanks to the recent advances in SMT solvers, our encoding techniques, and directed searches, these methods are efficient and scale well for larger inputs. We hope to apply them to other weak memory verification problems in the future.

Bibliography

[1] Parosh A. Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.

[2] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS*, volume 7460 of *LNCS*, pages 164–180. Springer, 2012.

[3] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *TACAS*, volume 7214 of *LNCS*, pages 204–219. Springer, 2012.

[4] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *TACAS*, volume 7795 of *LNCS*, pages 530–536. Springer, 2013.

[5] Parosh A. Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *CAV*, volume 9780 of *LNCS*, pages 134–156. Springer, 2016.

[6] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. Verification of programs under the release-acquire semantics. In *PLDI*, page 1117–1132. ACM, 2019.

[7] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. Sat-solving the coverability problem for petri nets. *Formal Methods in System Design*, 24(1):25–43, 2004.

[8] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[9] M. Ahamad, R.A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *SPAA*, pages 251–260. ACM, 1993.

[10] J. Alglave. A formal hierarchy of weak memory models. *FMSD*, 41(2):178–210, 2012.

[11] J. Alglave. Weakness is a virtue. $(EC)^2$ Workshop, 2013.

[12] J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig. Soundness of data flow analyses for weak memory models. In *APLAS*, volume 7078 of *LNCS*, pages 272–288. Springer, 2011.

[13] Jade Alglave. *A Shared Memory Poetics*. Thèse de doctorat, L'université Paris Denis Diderot, 2010.

[14] Jade Alglave. Simulation and invariance for weak consistency. In *SAS*, pages 3–22, 2016.

[15] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. Gpu concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, page 577–591. ACM, 2015.

[16] Jade Alglave and Patrick Cousot. Ogre and pythia: An invariance proof method for weak consistency models. In *POPL*, POPL 2017, page 3–18. ACM, 2017.

[17] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language CAT. *CoRR*, abs/1608.07531, 2016.

[18] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, volume 8559 of *LNCS*, pages 508–524. Springer, 2014.

[19] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.

[20] Jade Alglave and Luc Maranget. The diy7 tool suite. `http://diy.inria.fr/`.

[21] Jade Alglave and Luc Maranget. Stability in weak memory models. In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer, 2011.

[22] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *ASPLOS*, pages 405–418. ACM, 2018.

[23] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *TACAS*, pages 41–44. Springer, 2011.

[24] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[25] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8. IEEE, 2013.

[26] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[27] M.F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store buffers in TSO analysis. In *CAV*, volume 6806 of *LNCS*, pages 99–115. Springer, 2011.

[28] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.

[29] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. What's decidable about weak memory models? In *ESOP*, volume 7211 of *LNCS*, pages 26–46. Springer, 2012.

[30] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

[31] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66. ACM, 2011.

[32] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Checking robustness against snapshot isolation. In *CAV*, pages 286–304. Springer, 2019.

[33] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Robustness Against Transactional Causal Consistency. In *CONCUR*, volume 140 of *LIPIcs*, pages 30:1–30:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[34] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394. Springer, 2007.

[35] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[36] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

[37] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In *PPDP*, pages 113–124, 2011.

[38] Michael Blondin, Alain Finkel, Stefan Göller, Christoph Haase, and Pierre McKenzie. Reachability in two-dimensional vector addition systems with states is pspace-complete. In *LICS*, pages 32–43. IEEE Computer Society, 2015.

[39] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *PLDI*, pages 467–481. ACM, 2017.

[40] A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In *ICALP*, volume 6756 of *LNCS*, pages 428–440. Springer, 2011.

[41] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

[42] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In *POPL*, POPL 2017, page 626–638. ACM, 2017.

[43] Ahmed Bouajjani, Constantin Enea, Madhavan Mukund, Gautham Shenoy R., and S. P. Suresh. Formalizing and checking multilevel consistency. In *VMCAI*, volume 11990 of *LNCS*, pages 379–400. Springer, 2020.

[44] Alfred Brauer. On a problem of partitions. *American Journal of Mathematics*, 64(1):299–312, 1942.

[45] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21. ACM, 2007.

[46] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.

[47] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*, volume 6605 of *LNCS*, pages 11–25. Springer, 2011.

[48] C/C++11 mappings to processors. `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`. Accessed: 23.04.2018.

[49] G. Calin, E. Derevenetc, R. Majumdar, and R. Meyer. A theory of partitioned global address spaces. In *FSTTCS*, volume 24 of *LIPIcs*, pages 127–139. Schloss Dagstuhl, 2013.

[50] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):663–671, 2005.

[51] E. Cardoza, R. Lipton, and A. R. Meyer. Exponential space complete problems for petri nets and commutative semigroups (preliminary report). In *ACM*, STOC '76, pages 50–54. ACM, 1976.

[52] Sinan Cayir and Müjgân Uçer. An algorithm to compute a basis of petri net invariants. In *4th ELECO Int. Conf. on Electrical and Electronics Engineering*, 2005.

[53] Peter Chini, Jonathan Kolberg, Andreas Krebs, Roland Meyer, and Prakash Saivasan. On the Complexity of Bounded Context Switching. In *ESA*, volume 87 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[54] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *PLDI*, pages 211–225. ACM, 2018.

[55] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[56] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[57] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, volume 3920 of *LNCS*, pages 182–196. Springer, 2006.

[58] William W. Collier. *Reasoning about parallel architectures.* Prentice Hall, 1992.

[59] Scott Cotton, Eugene Asarin, Oded Maler, and Peter Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS*, volume 3253 of *LNCS*, pages 263–276. Springer, 2004.

[60] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[61] Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for petri nets is not elementary (extended abstract). *CoRR*, abs/1809.07115, 2018.

[62] Daniela Carneiro da Cruz, Maria João Frade, and Jorge Sousa Pinto. Verification conditions for single-assignment programs. In *SAC*, pages 1264–1270. ACM, 2012.

[63] Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *SAS*, volume 7935 of *LNCS*, pages 84–104. Springer, 2013.

[64] Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, volume 8931 of *LNCS*, pages 449–466. Springer, 2015.

[65] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[66] Giorgio Delzanno, Jean-Francois Raskin, and Laurent Van Begin. Towards the automated verification of multithreaded java programs. In *TACAS*, pages 173–187. Springer, 2002.

[67] Brian Demsky and Patrick Lam. Satcheck: Sat-directed stateless model checking for sc and tso. In *OOPSLA*, pages 20–36, 2015.

[68] Egor Derevenetc. *Robustness against Relaxed Memory Models.* doctoralthesis, Technische Universität Kaiserslautern, 2015.

[69] Egor Derevenetc and Roland Meyer. Robustness against Power is PSPACE-complete. In *ICALP*, volume 8573 of *LNCS*, pages 158–170. Springer, 2014.

[70] J. Esparza and P. Ganty. Complexity of pattern based verification for multithreaded programs. In *POPL*, pages 499–510. ACM, 2011.

[71] J. Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. In *TACAS*, volume 1055 of *LNCS*, pages 87–106. Springer, 1996.

[72] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp Meyer, and Filip Niksic. An smt-based approach to coverability analysis. In *CAV*, pages 603–619. Springer, 2014.

[73] Javier Esparza and Stefan Römer. An unfolding algorithm for synchronous products of transition systems. In *In Proceedings of the 10th International Conference on Concurrency Theory (Concur'99*, pages 2–20. Springer Verlag, 1999.

[74] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *TACAS*, volume 5505 of *LNCS*, pages 155–169. Springer, 2009.

[75] R. W. Floyd. Assigning meanings to programs. *Proceedings of a symposium on Applied Mathematics*, 19:19–32, 1967.

[76] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL*, pages 608–621. ACM, 2016.

[77] F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory model-aware testing - a unified complexity analysis. In *ACSD*, pages 92–101. IEEE, 2014.

[78] Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. Memory model-aware testing - A unified complexity analysis. In *ACSD 2014*, pages 92–101, 2014.

[79] Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embedded Comput. Syst.*, 14(4):63:1–63:25, 2015.

[80] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. From many places to few: automatic abstraction refinement for Petri nets. *Fundam. Inform.*, 88(3):275–305, 2008.

[81] Pierre Ganty, Cédric Meuter, Laurent Van Begin, Gabriel Kalyon, Jean-François Raskin, and Giorgio Delzanno. Symbolic data structure for sets of k-uples of integers. 2007.

[82] Pierre Ganty, Jean-Francois Raskin, and Laurent Begin. From many places to few: Automatic abstraction refinement for petri nets. volume 88, pages 124–143, 06 2007.

[83] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[84] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact smt encodings. In *CAV*, pages 355–365. Springer, 2019.

[85] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In *JELIA*, pages 137–151, 2014.

[86] G. Geeraerts, J.-F. Raskin, and L. Van Begin. Expand, enlarge and check: New algorithms for the coverability problem of wsts. *Journal of Computer and System Sciences*, 72:180 – 203, 2006.

[87] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set for petri nets. In *ATVA*, pages 98–113. Springer, 2007.

[88] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, Stanford, CA, USA, 1995.

[89] P.B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[90] J.R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin-Madison, 1991.

[91] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer, 2002.

[92] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 262–276. Springer, 2009.

[93] A. Heddaya and H. Sinha. Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Boston University, 1992.

[94] Keijo Heljanko, Misa Keinänen, Martin Lange, and Ilkka Niemelä. Solving parity games by a reduction to SAT. *J. Comput. Syst. Sci.*, 78(2):430–440, 2012.

[95] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A quantitative Approach*. Morgan Kaufmann, 3 edition, 2003.

[96] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[97] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *ICDCS*, pages 302–309. IEEE, 1990.

[98] Tomi Janhunen and Ilkka Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, pages 111–130, 2011.

[99] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Efficient coverability analysis by proof minimization. In *CONCUR*, pages 500–515. Springer, 2012.

[100] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147 – 195, 1969.

[101] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In *CAV*, pages 158–173. Springer, 2013.

[102] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018.

[103] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the linux kernel's read-copy update (rcu). *STTT*, 21(3):287–306, 2019.

[104] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266. IEEE, 1977.

[105] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988.

[106] Peep Küngas. Petri net reachability checking is polynomial with optimal abstraction hierarchies. In *Proceedings of the 6th International Conference on Abstraction, Reformulation and Approximation*, SARA'05, pages 149–164. Springer-Verlag, 2005.

[107] M. Kuperstein, M.T. Vechev, and E. Yahav. Partial coherence abstractions for relaxed memory models. In *PLDI*, pages 187–198. ACM, 2011.

[108] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, 2012.

[109] Ori Lahav and Roy Margalit. Robustness against release/acquire semantics. In *PLDI*, page 126–141. ACM, 2019.

[110] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI*, pages 618–632. ACM, 2017.

[111] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[112] Leslie Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, pages 347–374. Springer, 1994.

[113] R. Langerak and E. Brinksma. A complete finite prefix for process algebra. In *CAV*, volume 1633 of *LNCS*, pages 184–195. Springer, 1999.

[114] Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. Cerberus-BMC: A principled reference semantics and exploration tool for concurrent and sequential C. In *CAV*, volume 11561 of *LNCS*, pages 387–397. Springer, 2019.

[115] R. Lawrence. A survey of cache coherence mechanisms in shared memory multiprocessors, 1998.

[116] K. Rustan M. Leino. This is Boogie 2. 2008.

[117] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[118] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440. ACM, 2012.

[119] P. Loewenstein, S. Chaudhry, R. Cypher, and C. Manovit. Multiprocessor memory model verification. Automated Formal Methods Workshop (AFM), 2006.

[120] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. In *ASPLOS*, page 661–675. ACM, 2017.

[121] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *CAV*, pages 273–287. Springer, 2010.

[122] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Litmus tests for comparing memory consistency models: How long do they need to be? In *DAC*, page 504–509. ACM, 2011.

[123] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, volume 7358 of *LNCS*, pages 495–512. Springer, 2012.

[124] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016.

[125] E. Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on Computing*, 13(3):441–460, 1984.

[126] Paul E. McKenney and Jack Slingwine. Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[127] K. L. McMillan. A technique of state space search based on unfolding. *Form. Methods Syst. Des.*, 6(1):45–65, 1995.

[128] Robin Morisset and Francesco Zappa Nardelli. Partially redundant fence elimination for x86, ARM, and Power processors. In *CC*, pages 1–10. ACM, 2017.

[129] D. Mosberger. Memory consistency models. *ACM SIGOPS: Operating Systems Review*, 27(1):18–26, 1993.

[130] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[131] Ilkka Niemelä. Stable models and difference logic. *Ann. Math. Artif. Intell.*, 53(1-4):313–329, 2008.

[132] Brian Norris and Brian Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. In *OOPSLA*, OOPSLA 13, page 131–150. ACM, 2013.

[133] Yale University. Department of Computer Science and R.J. Lipton. *The reachability problem requires exponential space.* Research report (Yale University. Department of Computer Science). Department of Computer Science, Yale University, 1976.

[134] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, 1981.

[135] Pierre Ganty. mist - a safety checker for petri nets and extensions. "`https://github.com/pierreganty/mist`.

[136] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation*, pages 239–251. Springer, 2004.

[137] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In *SAS*, volume 10422 of *LNCS*, pages 299–320. Springer, 2017.

[138] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC with memory models as modules. In *FMCAD*, pages 1–9. IEEE, 2018.

[139] Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (competition contribution). In *TACAS*, pages 378–382. Springer, 2020.

[140] Hernán Ponce de León, Natalia Gavrilenko, Florian Furbach, Keijo Heljanko, and Roland Meyer. The Dat3M tool suite. `https://github.com/hernanponcedeleon/Dat3M`.

[141] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL*, 2(POPL):19:1–19:29, 2018.

[142] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223 – 231, 1978.

[143] Zvonimir Rakamaric and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, volume 8559 of *LNCS*, pages 106–113. Springer, 2014.

[144] Pierre-Alain Reynier and Frédéric Servais. Minimal coverability set for petri nets: Karp and miller algorithm with pruning. In *PETRI NETS*, pages 69–88. Springer, 2011.

[145] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[146] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, volume 3385 of *LNCS*, pages 25–41. Springer, 2005.

[147] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. *Petri Net Analysis Using Invariant Generation*, pages 682–701. Springer, 2003.

[148] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.

[149] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, pages 379–391. ACM, 2009.

[150] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[151] M. Senftleben. Operational characterization of weak memory consistency models. Master's thesis, Department of Computer Science, University of Kaiserslautern, 2013.

[152] M. Senftleben. *Modelling Memory Consistency Models for Formal Verification.* PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, 2019. PhD.

[153] M. Senftleben and K. Schneider. Operational characterization of weak memory consistency models. In *International Conference on Architecture of Computing Systems (ARCS)*, volume 10793 of *LNCS*, pages 195–208. Springer, 2018.

[154] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[155] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[156] S. Gulwani S. Srivastava and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292. ACM, 2008.

[157] R.C. Steinke and G.J. Nutt. A unified theory of shared memory consistency. *JACM*, 51(5):800–849, 2004.

[158] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[159] Viggo Stoltenberg-Hansen, Edward R. Griffor, and Ingrid Lindstrom. *Mathematical Theory of Domains.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[160] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[161] Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. In *PLDI*, pages 341–350. ACM, 2010.

[162] M. Triebel and J. Sürmeli. Characterizing stable inequalities of petri nets. In *PETRI NETS*, volume 9115 of *LNCS*, pages 266–286. Springer, 2015.

[163] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.

[164] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220. ACM, 2015.

[165] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, pages 867–884. ACM, 2013.

[166] Antti Valmari and Henri Hansen. Old and new algorithms for minimal coverability sets. In Serge Haddad and Lucia Pomello, editors, *Application and Theory of Petri Nets*, pages 208–227. Springer, 2012.

[167] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344, 1986.

[168] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual - Version 9.* Prentice-Hall, 1994.

[169] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL*, pages 190–204. ACM, 2017.

[170] Wikipedia contributors. Clang: A c language family frontend for llvm. "`http://clang.llvm.org`.

[171] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*. IEEE Computer Society, 2004.

[172] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Checking causal consistency of distributed databases. In *Networked Systems*, pages 35–51. Springer, 2019.

[173] Rachid Zennou, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Gradual consistency checking. In *CAV*, pages 267–285. Springer, 2019.

Appendices

Appendix A

Bounded Model Checking

A.1 Read-Modify-Write

In DARTAGNAN instructions are modeled as instances of events, with the exception of atomic read-modify-write instructions that are represented as sequences of up to three events. The most important events supported by DARTAGNAN consist of basic events modeling memory stores, loads, expression evaluation, control flow, and fences. In addition there are the RCU (read-copy-update, see [22]) synchronization operations needed by the Linux kernel and different variants of atomic read-modify-write instructions. They are as follows:

Basic	**Basic RMW**
store(ad:expr, val:expr, mo)	rmwStore(load:rmwLoad, ad:expr, val:expr, mo)
reg = load(ad:expr, mo)	reg = rmwLoad(ad:expr, mo)
reg = local(val:expr)	**Conditional RMW**
if(pred:expr) br1 br2	rmwStoreCond(load:rmwLoadCond, ad:expr, val:expr, mo)
fence(type, option)	reg = rmwLoadCond(ad:expr, pred:expr, mo)
RCU	fenceCond(load:rmwLoadCond)
rcuReadLock	**Exclusive RMW**
rcuReadUnlock	rmwStoreExcl(load:rmwLoad, ad:expr, val:expr, mo)
rcuSync	reg = rmwStoreExclStatus(store:rmwStoreExcl).

A.1.1 Read-Modify-Write Events

In order to be compatible with the memory models defined for HERD7, we follow the same principle of modeling Read-Modify-Write (RMW) events. An atomic RMW is modeled as a pair of read and write events with an edge of the *rmw* relation between them. Atomicity of an RMW pair is enforced in a memory model by the constraint `empty rmw & (fre;coe)`, which invalidates executions with an external write event between an RMW pair.

Linux, especially the read-modify-write instructions and the RCU primitives. We represent these primitives by a set of special load, store, and fence events. An (unconditional) atomic read-modify-write instruction, such as atomic xchg, yields two events, an rmwStore and an rmwLoad. They are similar to their non-atomic counterparts except that an rmwStore requires a reference to its rmwLoad partner. This reference is used to define the relation rmw between the load and the store. To enforce atomicity, we rely on the following constraint, which is the de-facto standard definition of atomicity in the `cat` language:

$$\begin{aligned} &\mathsf{empty\ rmw\ \&\ (fre\ ;\ coe)} \qquad &&\mathsf{coe = co\ \&\ ext} \\ &\mathsf{fr = rf^{-1}\ ;\ co} &&\mathsf{fre = fr\ \&\ ext}\ . \end{aligned}$$

It forbids executions where between the rmwLoad and rmwStore another thread writes to the rmw-address. Thread internally, such writes are forbidden by the (also standard) constraint:

$$\mathsf{acyclic\ (po\ \&\ loc)\ |\ co\ |\ rf\ |\ fr}\ .$$

In conditional RMW events, the store is executed only if the value read by the associated load event satisfies the condition of the load. Conditional fences[1] model memory ordering guarantees which depend on a successful comparison, for example in Linux atomic_cmpxchg. In conditional read-modify-write atomic events, whether the write component e is executed depends on a condition *cond*. The value of $\mathsf{exec}(e)$ for the write component is encoded as:

$$\mathsf{exec}(e) \Leftrightarrow cf_e \wedge cond.$$

Exclusive RMW events model ARM LDXR/STXR pairs and their siblings. In these pairs, the STXR instruction is executed only if no write to the same address appeared after its LDXR partner. The instruction may also spuriously fail. STXR returns 0 if the write was successful and 1 otherwise. We map STXR to a pair of events, rmwStoreExcl and rmwStoreExclStatus. The encoding of the execution variables is as follows:

$$\begin{aligned} \mathsf{exec(rmwStoreExcl)} &\rightarrow cf_{rmwStoreExcl} \\ \mathsf{val(rmwStoreExclStatus)} &\leftrightarrow (\mathsf{exec(rmwStoreExcl)}\ ?\ 0\ :\ 1). \end{aligned}$$

The encoding of the atomicity requirement follows the standard scheme in `cat`.

Finally, we use a set of event for modeling Linux RCU as well. The event rcuReadLock is related to an event rcuReadUnlock with the relation crit, while the event rcuSync is used to build relation gp (grace period) defined in the `cat` memory model of Linux.

[1]Described in the Linux kernel docs `https://github.com/torvalds/linux/blob/master/Documentation/core-api/atomic_ops.rst`.

Appendix B

Program Portability

B.1 Correctness of Least Fixpoint Encodings

In Section 5.5.2, we presented our encoding for the least fixpoints of recursively defined relations. We recall the correctness theorem.

Theorem 11. *For any recursively defined relation r holds that the IDL-encoding of Kleene has a satisfying assignment and for any satisfying assignment holds $r(e_1, e_2) = true$ iff $(e_1, e_2) \in r$.*

Proof. First, we show that the encoding has a satisfying assignment. We construct a satisfying assignment of the formula. For any relation r, we set $r(e_1, e_2) = true$ for any tuple $(e_1, e_2) \in r$. Since the semantics of recursively defined relations are the least fixpoint over a finite lattice, they can be determined using the Kleene iteration. For any pair of events (e_1, e_2) in a relation r there is an iteration step i in which it was added to the relation. We set $\Phi^{\mathtt{r}}_{e_1,e_2} = i$ and r_1 accordingly. For any pair $(e_1, e_2) \notin r$ we set $r(e_1, e_2) = false$ and $\Phi^{\mathtt{r}}_{e_1,e_2} = 0$ (r_1, r_2 accordingly). Since the semantics of a relation r is the least fixpoint, it follows that any formula in Fig. 4.3, which requires r to be a fixpoint, is satisfied.

We assume w.l.o.g. that $r = r_1 \cup r_2$ and r is a subrelation of r_1 but not of r_2. The encoding is as follows

$$\mathtt{r}(e_1, e_2) \Leftrightarrow (\mathtt{r}_1(e_1, e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2})) \vee (\mathtt{r}_2(e_1, e_2)) \tag{B.1}$$

$$\mathtt{r}(e_1, e_2) \Leftrightarrow \mathtt{r}_1(e_1, e_2) \vee \mathtt{r}_2(e_1, e_2). \tag{B.2}$$

Condition B.1 is defined in Fig. 5.3 and Condition B.2 in Fig. 4.3.

For any tuple $(e_1, e_2) \in r$, one of two cases holds. Case 1 is as follows: The tuple is contained in r_1 and it was added to it one iteration step before it was added to r and thus $(\mathtt{r}_1(e_1, e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2}))$ holds. Case 2 specifies that the tuple is contained in r_2. Either way, It follows that Condition B.1 is satisfied.

If $(e_1, e_2) \notin r$, then $\Phi^{\mathtt{r}}_{e_1,e_2}$ has the smallest value 0 and thus $(\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2})$ cannot hold. It follows that the constructed assignment is indeed satisfying. The argument is analogue for any other operator and for r being a subrelation of r_2.

It remains to be proven that any satisfying assignment describes the least fixpoint. According to condition B.2, any satisfying assignment describes some fixpoint. We show that for every variable $r(e_1, e_2)$ with $r(e_1, e_2) = true$ in a satisfying assignment, it holds $(e_1, e_2) \in r$. We perform an induction over the values of $\Phi^{\mathtt{r}}_{e_1,e_2}$ for all $r(e_1, e_2) = true$.

Induction hypothesis: For any relation r and tuple (e_1, e_2) with $r(e_1, e_2) = true$ and $\Phi^{\mathtt{r}}_{e_1,e_2} \leq i$ it holds $(e_1, e_2) \in r$.

Induction basis: Let $r(e_1, e_2)$ be a Boolean variable among those with $r(e_1, e_2) = true$ that has the smallest corresponding value $\Phi^{\mathtt{r}}_{e_1,e_2}$. Since the variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ exists, r is a recursive relation and has an encoding according to Fig. 5.3. Since $\Phi^{\mathtt{r}}_{e_1,e_2}$ has the lowest value, there is no satisfied subformula of the form $\mathtt{r'}(e, e') \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r'}}_{e,e'})$. Yet, the encoding of r according to Fig. 5.3 is satisfied. It follows that r must have the form $r_1 \cup r_2$, $r_1; r_2$, r_1^*, or r_1^+ and one of the formulas it depends on is not recursive (or $e_1 = e_2$ in the case of r_1^*). Assume w.l.o.g that $r = r_1 \cup r_2$ and r_2 is not recursive (see B.1). Since $r(e_1, e_2) = true$ it follows $r_2(e_1, e_2) = true$. Since r_2 is not recursive and non-recursive relations are encoded correctly, we know $(e_1, e_2) \in r_2$. This means (e_1, e_2) is added to r in the first iteration of Kleene and thus $(e_1, e_2) \in r$. The other cases are analogue.

Induction step: $(i \to i + k)$ Let $i + k$ be the smallest value of a variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ that is larger than i. We again assume w.l.o.g. $r = r_1 \cup r_2$ and that r_2 is not recursive. Since $r(e_1, e_2) = true$ holds, either $r_2(e_1, e_2)$ or $(\mathtt{r}_1(e_1, e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2}))$ are satisfied. If $r_2(e_1, e_2) = true$, then (e_1, e_2) is added to r in the first iteration of Kleene and thus $(e_1, e_2) \in r$. If $(\mathtt{r}_1(e_1, e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2}))$ is satisfied, then according to the induction hypothesis $(e_1, e_2) \in r_1$ holds. It follows from $r = r_1 \cup r_2$ that $(e_1, e_2) \in r$ holds. The argument is analogue for other definitions of r.

□

B.2 Complexity of Trace Portability

We recall the main theorem and the program $P_\psi := t_1 \parallel t_2$ from Section 5.6:

Theorem 12. *Let $\mathcal{M}_S, \mathcal{M}_T$ be a non-trivial pair of common memory models. (1) Trace Portability from $\mathcal{M}_S$ to $\mathcal{M}_T$ is Π_1^P-complete. (2) State portability is Π_2^P-complete.*

We defined $P_{np} = t'_1 \parallel \cdots \parallel t'_k$ and $P_{\forall\psi} := t''_1 \parallel \cdots \parallel t''_k$ with $t''_i := t_i;\ r \leftarrow y;$ **if**$(r = 1)$ **then** t'_i.

We give some technical results and show Π_1^P-completeness of portability for common memory models.

Lemma 19. *P_ψ is trace and state portable from every common memory model to another common memory model.*

```
$t_1$
$r_{c,0} \leftarrow 0;\ r_{c,1} \leftarrow 1;\ r_{c,2} \leftarrow 2;$
$x_1 \coloneqq r_{c,0}; \ldots; x_m \coloneqq r_{c,0}$ ;                    // Writes $w_{1,0} \ldots w_{m,0}$
$r_1 \leftarrow x_1; \ldots; r_m \leftarrow x_m$ ;                          // Reads $r_1 \ldots r_m$
if $\psi(r_1, \ldots, r_m)$ then                                  // If $A$ satisfies $\psi$,
    $y \coloneqq r_{c,2}$ ;                                                   // return 2.
else
    $y \coloneqq r_{c,1}$ ;                                  // If it doesn't, return 1.
end
```

```
$t_2$
$r_{c,1} \leftarrow 1;$
$x_1 \coloneqq r_{c,1};\ \ldots x_m \coloneqq r_{c,1}$ ;                    // Writes $w_{1,1} \ldots w_{m,1}$
```

Proof. Since trace portability implies state portability (see Lemma 7), it is sufficient to show trace portability. According to Property (v), any common memory model is portable to SC where an execution corresponds to an interleaving of the two thread executions. First, the threads create some variable assignment A which is read by t_1. Then, t_1 checks whether the assignment satisfies ψ. If it does, y is set to 2, otherwise y is set to 1.

We show that any consistent execution of some common memory model is consistent with SC by examining possible executions.

- If $w_{i,1} \xrightarrow{co} w_{i,0}$ and $w_{i,0} \xrightarrow{rf} r_i$, then this corresponds to an interleaving where $w_{i,1}$ occurs first, then t_1 writes $w_{i,0}$ and reads 0 (r_i).

- If $w_{i,0} \xrightarrow{co} w_{i,1}$ and $w_{i,0} \xrightarrow{rf} r_i$, then $w_{i,0}$ and r_i in t_1 occur first and then $w_{i,1}$.

- If $w_{i,0} \xrightarrow{co} w_{i,1}$ and $w_{i,1} \xrightarrow{rf} r_i$, then t_1 writes $w_{i,0}$, t_2 overwrites this with $w_{i,1}$ and afterward t_1 reads 1 with r_i.

- If $w_{i,1} \xrightarrow{co} w_{i,0}$ and $w_{i,1} \xrightarrow{rf} r_i$, then the derived relation $fr \coloneqq rf^{-1}; co$ satisfies $r_i \xrightarrow{fr} w_{i,0}$. Since $w_{i,0}$ and r_i are related by po and access the same location there is a cycle $w_{i,0} \xrightarrow{po \cap loc} r_i \xrightarrow{fr} w_{i,0}$. This is a violation of uniproc, the situation can not occur in a common memory model.

So we can construct a corresponding interleaving for any execution of a common memory model. It follows that every execution of a common memory model is consistent with SC and according to property (iv) consistent with any common memory model. $\square$

We use the following technical lemmas to show hardness. We call the relations $po, rf, co, addr, data, ctrl, rmw$ and fr basic. Given a common memory model, we define the *violating cycles* as follows: For an assertion $acyclic(r)$, any cycle of r is violating. For an assertion $irreflexive(r)$, any cycle of the form $e \overset{r}{\to} e$ is violating.

The following lemma shows that an execution is not consistent if it contains a violating cycle or violates Constraint $\textcircled{18}$.

Lemma 20. *Let $\mathcal{M}$ be common. An execution X is consistent with $\mathcal{M}$ iff X contains no violating cycle of $\mathcal{M}$ and satisfies Constraint $\textcircled{18}$.*

This follows directly from the definiton of violating cycles.

We say a relation r satisfies the path condition if $e_1 \overset{r}{\to} e_2$ implies $e_1 \overset{b}{\longrightarrow}{}^* e_2$ with b denoting a union of all base relations.

Lemma 21. *Any relation r of a common memory model satisfies the path condition.*

Proof. Note that the recursively defined relations of a common memory model can be obtained with a Kleene iteration. We use a structural induction over the Kleene iteration.

Induction Basis: Any named relation is initially the empty relation and trivially satisfies the path condition. Any base relation also satisfies the path condition.

Induction Step: Assume all named relations satisfy the path condition. A named relation is updated according to its defining equation using the current assignments of the named relations and basic relations. To simplify this proof, we can assume that any equation only contains one operator or a relation in *base*. Any basic relation trivially satisfies the condition. Let r_1 and r_2 satisfy the path condition. We examine the operations that can be applied to them according to properties (i) and (ii) of common memory models. We see that $r_1 \cup r_2, r_1 \cap r_2, r_1 \cap sloc, r_1 \cap sthd, r_1 \cap \langle set \rangle \times \langle set \rangle, r_1 \setminus r_2$ all satisfy the path condition. The relation $r_1; r_2(e_1, e_2)$ requires an event e_3 with $r_1(e_1, e_3)$ and $r_2(e_3, e_2)$ and since r_1 and r_2 satisfy the path condition there is a path from e_1 to e_3 and from e_3 to e_2 and thus $r_1; r_2$ also satisfies the path condition. Similarly, r_1^+ adds a relation only where there already is a path and satisfies the path condition. Relation r_1^* consists of r^+ and the identity relation, which also satisfies the path condition (there is always a path of length 0 from some e_1 to e_1). A named relation still satisfies the path condition after a Kleene iteration step.

□

Note that according to Lemma 21, a violating cycle implies a cycle of basic relations (called a basic path) that contains all events of the violating cycle. We can argue about consistency of an execution by examining the paths of basic relations.

Lemma 22. *Given a relation r of a common memory model and events e_1, e_2 of an execution, whether $r(e_1, e_2)$ holds is fully determined by the basic paths from e_1 to e_2.*

Proof. We use a structural induction over the Kleene iteration.

Induction Basis: Any named relation is initially the empty relation and trivially satisfies the condition. Any basic relation trivially satisfies the condition.

Induction Step: Assume all current assignments of relations are determined by the basic paths between related events. Two events e_1 and e_2 are related by some relation r iff the basic paths from e_1 to e_2 satisfy some property. A named relation is updated according to its defining equation using the current assignments of the named relations and basic relations. Let r_1, r_2 be either named or basic relations. Events e_1, e_2 are related by $r_1 \cup r_2$ ($r_1 \cap r_2$) if the basic paths from e_1 to e_2 satisfy the property of r_1 or r_2 (resp. r_1 and r_2). Similar, events are related by $r_1 \setminus r_2$ if the basic paths satisfy the property of r_1 but not r_2. For $r_1 \cap \mathit{sloc}, r_1 \cap \mathit{sthd}, r_1 \cap \langle \mathit{set} \rangle \times \langle \mathit{set} \rangle$ two events e_1, e_2 are related if the property of r_1 is satisfied and e_1 and e_2 satisfy an additional condition. The condition for the events e_1, e_2 can be expressed as a additional condition for paths from e_1 to e_2.

The relation $r_1; r_2$ relates e_1 to e_2 if there is an e_3 such that the basic paths from e_1 to e_3 ensure $r_1(e_1, e_3)$ and the basic paths from e_3 to e_2 ensure $r_1(e_3, e_2)$. It follows that $r_1; r_2(e_1, e_3)$ depends on the basic paths from e_1 to e_2 over some e_3. The relations r_1^* and r_1^+ are derived using previously examined operators over relations and thus they satisfy the condition.

A named relation still satisfies the path condition after a Kleene iteration step. □

Proof of Theorem 12 (1). We show that $P_{\forall\psi}$ is not portable iff ψ has an unsatisfying assignment.

($\Rightarrow$): We assume an execution X exists that is consistent with $\mathcal{M}_T$ but not with $\mathcal{M}_S$. We assume towards contradiction that no event of P_{np} is executed. Since the execution is consistent with $\mathcal{M}_T$ and thus satisfies uniproc it follows that the read $r \leftarrow y$ in t_1'' has to read from the write in t_1 (in P_ψ). It occurs after t_1 in the program order. The read has incoming basic relations from events in P_ψ but no outgoing relations to some event in P_ψ. Any read $r \leftarrow y$ in another thread can either read from the write in P_ψ (it has an incoming *rf* relation from P_ψ) or it reads the initial value which results in an outgoing from-read relation to P_ψ. This behavior is also allowed under SC. Since $\mathcal{M}_S$ is weaker than SC (Property *(v)*), the execution is consistent with $\mathcal{M}_S$. This is a contradiction.

It follows that any consistency violation requires events from P_{np} to be executed. Since an event of P_{np} can only be executed if $y = 1$ was read in the thread (and thus written by t_1), the if-condition in Line 8 was not satisfied. It follows that there is an assignment that does not satisfy ψ.

($\Leftarrow$): We now assume that there is an assignment that doesn't satisfy ψ. There is an execution Z of P_ψ consistent with SC that executes the write $y := r_{c,1}$.

We extend this execution of P_ψ to an execution X of $P_{\forall\psi}$: We ensure that all reads $r \leftarrow y$ read from $y := r_{c,1}$ and thus P_{np} is executed. Let Y be an execution of P_{np} that is consistent with $\mathcal{M}_T$ but not $\mathcal{M}_S$. This results in an execution $X \supset Y \cup Z$ of $P_{\forall\psi}$ that contains the executions of P_ψ and P_{np}, the read from relation for the reads of y and the required program order additions. Since P_{np} has no registers or locations in common with the rest of the program and occurs last in *po*, no basic relation of X leaves Y. It follows that X contains the same basic paths between events of Y as Y. Thus any violating cycle for $\mathcal{M}_S$ or violation of an emptiness constraint (see Property *(iii)*) in Y is also in X. It follows that X is not consistent with $\mathcal{M}_S$. We show that X is still consistent with $\mathcal{M}_T$. We assume towards contradiction that it is not consistent with $\mathcal{M}_T$: As before, it holds that a violating cycle must contain an event from P_{np} and thus from Y. Since Y is never left, any basic cycle of X with events in Y must be contained entirely in Y. Since only Y may contain edges of *rmw*, any violation of Constraint $\textcircled{18}$ must be contained entirely in Y as well. It follows from (Lemma 21) that a violating cycle or violation of an emptiness constraint for $\mathcal{M}_T$ must be entirely in Y. This is a contradiction to the execution of P_{np} being consistent with $\mathcal{M}_T$.

Since $P_{\forall\psi}$ is a polynomial-time reduction, trace portability is Π_1^P-hard. □

B.3 Complexity of State Portability

We introduce Lemma 23 and Theorem 23 in order to show that state portability is both in Π_2^P and is Π_2^P-hard. It follows, that state portability is Π_2^P-complete for common memory models and thus Theorem 12 (2) is correct.

Lemma 23. *State portability is in* Π_2 *for all memory models.*

Proof. We encode the state portability property in a closed formula (i.e. all variables are quantified) of the form $\forall\exists\psi$. We have already shown how to encode consistency of an execution X with a memory model $\mathcal{M}$ as a formula $(X \in \mathit{cons}_{\mathcal{M}}(P))$ and how to encode assertions on a computed state $\mathit{state}(X)$ in Section 4.5. With this, we can construct the following formula:

$$\forall X\ \exists Y :\ (X \in \mathit{cons}_{\mathcal{M}_T}(P)) \Rightarrow$$
$$(Y \in \mathit{cons}_{\mathcal{M}_S}(P) \wedge \mathit{state}(X) = \mathit{state}(Y)).$$

This is equivalent to state portability (see Definition 12). The state portability problem from $\mathcal{M}_S$ to $\mathcal{M}_T$ can be expressed as a closed quantified formula of the form $\forall\exists\psi$ and thus state portability is in Π_2^P. □

We now introduce the program P_ψ and examine its behavior. We will then use P_ψ in order to prove Π_2^P-hardness.

Let $\psi(x_1 \dots x_m)$ be a be a Boolean formula over variables $x_1 \dots x_m$. The concurrent program $P_\psi := t_1 \parallel t_2$ with two threads t_1 and t_2 is defined below. The program is similar to the program in the previous section. It contains additional synchronization in order to ensure that the formula assignment and the computed state match. The program either computes a satisfying assignment of ψ ($y = 1$), an unsatisfying assignment ($y = 0$) or it ends with an error ($y = 2$).

We use the value 0 to encode the Boolean value false. To avoid confusion, we assume that the variables are initialized with some other unused value, e.g. 3. This does not interfere with the validity of the proofs since our program only assigns constants and thus 0 and 3 are interchangeable.

We will see that it is sufficient to examine the program under SC (the strongest common memory model) where an execution is an interleaving of the two thread executions. The threads first create some variable assignment A; thread t_1 assigns 0 to the variables and t_2 assigns 1. The assignment A is determined by the interleaving of those writes. If the write $x_i := 0$ of t_1 is followed by $x_i := 1$ of t_2 ($w_{i,0} \overset{co}{\rightarrow} w_{i,1}$), then x_i is set to 1. Then t_1 ensures that t_2 has executed all its writes (so that the assignment doesn't change anymore) by using the synchronization variables $x'_1 \dots x'_m$ to check that t_1 and t_2 reads the same assignment. If that is not the case, then some writes of t_2 have not occurred yet and y is set to 2 in Line 6. If all the writes from t_2 have occurred, t_1 checks whether A satisfies ψ (y is set to 1) or not (y is set to 0).

```
t_1
1  $r_{c,0} \leftarrow 0$; $r_{c,1} \leftarrow 1$; $r_{c,2} \leftarrow 2$;
2  $x_1 := r_{c,0}$; $\dots x_m := r_{c,0}$ ;                      // $w_{1,0} \dots w_{m,0}$
3  $r'_1 \leftarrow x'_1; \dots r'_m \leftarrow x'_m$ ;            // $r'_1 \dots r'_m$
4  $r_1 \leftarrow x_1; \dots r_m \leftarrow x_m$ ;                // $r_1 \dots r_m$
5  if $\neg(r_1 = r'_1 \wedge \dots \wedge r_m = r'_m)$ then      // If $A$ is incomplete
6      $y := r_{c,2}$ ;                                         // exit with error.
7  else
8      if $\psi(r_1 \dots r_m)$ then                            // If $A$ satisfies $\psi$,
9          $y := r_{c,1}$ ;                                     // return 1.
10     else
11         $y := r_{c,0}$ ;                          // If it does not, return 0.
12     end
13 end
```

```
t_2
1  $r_{c,1} \leftarrow 1$; $r_{c,3} \leftarrow 3$;
2  $x_1 := r_{c,1}$; $\dots x_m := r_{c,1}$ ;                      // $w_{1,1} \dots w_{m,1}$
3  $r_1 \leftarrow x_1; \dots r_m \leftarrow x_m$ ;                // $\bar{r}_1 \dots \bar{r}_m$
4  $x'_1 := r_1$; $\dots x'_m := r_m$ ;                          // $\bar{w}_1 \dots \bar{w}_m$
5  $x'_1 := r_{c,3}$; $\dots x'_m := r_{c,3}$
```

To simplify our study we will define a state only over its locations, not the registers. This does not change the complexity of the state portability problem:

We could simply add instructions to our input programs that write all registers to locations in the end. We can use our simpler notion of states without registers on the input program with the added instructions to solve the original state portability problem with registers.

An assignment A of a set of Boolean variables V is a function $A \subset V \times \{0,1\}$ that assigns either 0 or 1 to a variable.

Definition 15. *Given an Assignment A of $x_1 \dots x_n$, locations $y_1 \dots y_m$ and a number $a \in \mathbb{N}$, let $\sigma[A; y_1 \dots y_n \leftarrow a]$ denote the state with*

- *$\sigma[A; y_1 \dots y_n \leftarrow a](x_i) = A(x_i)$ for $i \leq n$ and*
- *$\sigma[A; y_1 \dots y_n \leftarrow a](y_j) = a$ for $j \leq m$,*
- *$\sigma[A; y_1 \dots y_n \leftarrow a](z) = v$ for z any other location and v the initial value (we use 3).*

We lift the definition accordingly to

$$\sigma[A; y_1 \dots y_n \leftarrow a; z_1 \dots z_l \leftarrow b].$$

The program P_ψ can compute some assignment A with $y = 1$ or $y = 0$ depending on whether A satisfies ψ.

The following lemmas show that P_ψ behaves similar for all common memory models.

Lemma 24. *Let $\mathcal{M}$ ensure uniproc and A be an assignment of ψ. It holds $A \models \psi$ (resp. $A \not\models \psi$) only if*

$$\exists X \in cons_{\mathcal{M}}(P_\psi) : state(X) = \sigma[A; y \leftarrow 1]$$

$$(resp.\ \ state(X) = \sigma[A; y \leftarrow 0]).$$

Proof. We show that any execution consistent with $\mathcal{M}$ and $y = 1$ (or $y = 0$) computes a desired state for some assignment A. Let X be an execution that satisfies uniproc and computes some state σ with $\sigma(y) = 1$ ($\sigma(y) = 0$ is analogue). Since $y := 1$ is executed in t_1, it follows that ψ is satisfied by the values of $x_1 \dots x_m$ read by $r_1 \dots r_m$ in Line 4 and also $r'_1, \dots, r'_m$ in Line 3 read the same values. We call this assignment A. Since the reads in Line 5 of t_2 occur after the writes $\bar{w}_1 \dots \bar{w}_m$, it follows that they occur last in co according to uniproc. The execution computes 3 for $x'_1 \dots x'_m$.

It remains to show that the writes accessed by $r_1 \dots r_m$ are indeed computed by X, meaning they are ordered last in co. Towards contradiction, we assume this is not the case. Then, there is a write $w_{i,1}$ or $w_{i,0}$ that is accessed by a read but its value is not computed (it is not last in co).

Case 1: Assume that this write is $w_{i,1}$. The write is accessed by a read ($w_{i,1} \overset{rf}{\to} r_i$) and it is not last in the coherence order ($w_{i,1} \overset{co}{\to} w_{i,0}$). According to $fr := rf^{-1}; co$, it holds $r_i \overset{fr}{\to} w_{i,0}$. Since $w_{i,0}$ occurs before r_i in t_1 and they access the same location, they are related w.r.t *po* and *loc* and thus $w_{i,0} \xrightarrow{po \cap loc} r_i \overset{fr}{\to} w_{i,0}$. This cycle is a contradiction to X satisfying uniproc which is Property (iv) of common memory models.

Case 2: Assume that there is a write $w_{i,0}$ that is read ($w_{i,0} \overset{rf}{\to} r_i$) and its value is not computed ($w_{i,0} \overset{co}{\to} w_{i,1}$). It follows that r_i reads the value 0. Since $y = 1$ is computed, the condition in Line 8 is not satisfied and since $val(r_i) = val(r'_i)$, we know that r'_i also reads 0. This means r'_i reads not the initial value 3 so it must read from $\bar{w}_i$. For the write $\bar{w}_i$ that r'_i reads from ($\bar{w}_i \overset{rf}{\to} r'_i$) follows that $val(\bar{w}_i) = val(r'_i) = 0$. According to the data-flow, $\bar{w}_i$ writes the value that was obtained by the previous read $\bar{r}_i$ which must be 0 ($val(\bar{r}_i) = val(\bar{w}_i) = 0$). Since $\bar{r}$ reads 0 it must read from the only write of 0 ($w_{i,0} \overset{rf}{\to} \bar{r}_i$). From $w_{i,0} \overset{rf}{\to} \bar{r}_i$ and $w_{i,0} \overset{co}{\to} w_{i,1}$ follows $\bar{r}_i \overset{fr}{\to} w_{i,1}$. This leads to the cycle $w_{i,1} \xrightarrow{po \cap loc} \bar{r}_i \overset{fr}{\to} w_{i,1}$, which is a contradiction to X satisfying uniproc.

So the writes accessed by $r_1 \dots r_m$ are ordered last by *co* and thus the assignment that ψ is checked against is computed by the execution. It follows that $\sigma[A; y \leftarrow 1]$ is computed. □

Lemma 25. *Let $\mathcal{M}$ be a common memory model and A be an assignment of ψ. It holds $A \vDash \psi$ (resp. $A \nvDash \psi$) iff*

$$\exists X \in cons_{\mathcal{M}}(P_\psi) : state(X) = \sigma[A; y \leftarrow 1]$$

$$(resp.\ state(X) = \sigma[A; y \leftarrow 0]).$$

Proof. Given some assignment A, we can easily construct an execution consistent with SC where the writes to $x_1 \dots x_n$ are interleaved according to the assignment ($w_{i,0} \overset{co}{\to} w_{i,1}$ if x_i is satisfied). Then t_2 reads those values and writes them to $x'_1 \dots x'_n$. Now t_1 reads $x'_1 \dots x'_n$ and the if-condition in Line 5 is not satisfied, we go in the else-branch. So t_1 sets y to 0 or 1 depending on whether $A \vDash \psi$ and t_2 sets $x'_1 \dots x'_n$ back to the initial value 3. It follows that for every assignment A, there is an execution X consistent with SC that computes $\sigma[A; y \leftarrow 0]$ or $\sigma[A; y \leftarrow 1]$ depending on whether A satisfies ψ. Since any execution consistent with SC is also consistent with all common memory models, we can compute the desired state for any assignment of ψ. The other direction follows directly from Lemma 24. □

In order to show Π_2^P-hardness, we reduce validity of a closed formula $\forall x_1 \dots x_n \exists y_1 \dots y_m : \psi$ to state portability. The idea is to construct a program that uses P_ψ in order to check if some assignment satisfies ψ and then overwrite $y_1 \dots y_m$ with 1 so that the assignment of $y_1 \dots y_m$ is not given by the computed state. If ψ was not satisfied, the non-portable component P_{np}

is executed. If the execution of P_{np} is consistent with $\mathcal{M}_T$ but not $\mathcal{M}_S$, then it pretends that the formula was satisfied by setting y to 1. This means, that under $\mathcal{M}_T$, any assignment of $x_1 \dots x_n$ with $y = 1$ can be computed. It follows that the program is portable if any assignment of $x_1 \dots x_n$ with $y = 1$ can be computed under $\mathcal{M}_S$ as well. Under $\mathcal{M}_S$ however, pretending is not possible. Here P_ψ can only be set to 1 by P_ψ. So under $\mathcal{M}_S$, an assignment of $x_1 \dots x_n$ and $y = 1$ can only be computed if there is some assignment of $y_1 \dots y_m$ so that ψ is satisfied. The program is portable if $\forall x_1 \dots x_n \exists y_1 \dots y_m : \psi$ holds.

We want a non portable program that always computes the initial state except under $\mathcal{M}_T$, where it can set a location z to 1. We use a program $P_{np} = t'_1 \parallel \cdots \parallel t'_k$ with the following properties: Any execution consistent with $\mathcal{M}_S$ computes the initial value 3 for all its locations and contains no write that sets z to 1. The executions consistent with $\mathcal{M}_T$ compute either $z = 1$ or $z = 3$ and 3 for all other locations. The program contains only one write to z which is in t'_1.

We assume the state portability problem from $\mathcal{M}_S$ to $\mathcal{M}_T$ is not trivial and a program exists that has an execution consistent with $\mathcal{M}_T$ such that no execution consistent with $\mathcal{M}_S$ computes the same state σ. We can assume that the program only assigns constant values and has no write on z.

Similarly to the synchronization of P_ψ, we can add a mechanism at the end of all threads that does the following: all threads check if they read σ; if so, they communicate that to t_1 which sets z accordingly and then all threads set all other locations back to 3. It follows that a program P_{np} with the required properties exists.

Given a formula $\forall x_1 \dots x_n \exists y_1 \dots y_m : \psi$, we use $P_\psi = t_1 \parallel t_2$ and $P_{np} = t'_1 \parallel \cdots \parallel t'_k$ to construct a program $P_s := t^s_1 \parallel \cdots \parallel t^s_k$. We define the threads below.

```
$t^s_1$
$t_1$ ;                                          // Try some assignment.
$r_y \leftarrow y$;
if $r_y = 0$ then                                // If $\psi$ was not satisfied,
    $t'_1$ ;                                     // execute $P_{np}$.
    $r_z \leftarrow z$;
    if $r_z = 1$ then                            // If not $\mathcal{M}_S$-consistent,
        $z := r_{c,3}$ ;                         // pretend it is $\mathcal{M}_S$-consistent
        $y := r_{c,1}$ ;                         // and pretend $\psi$ was satisfied.
    end
end
$y_1 := r_{c,1}; \cdots y_m := r_{c,1}$ ;        // Overwrite $y_1..y_m$ assignment.
```

Let t_i be an empty sequence for $i \leq 3 \leq k$ (t_1 and t_2 are from P_ψ). The threads are defined for $2 \leq i \leq k$ as

$$t^s_i := t_i;\ r_y \leftarrow y;\ \textbf{if}\ (r_y = 0)\ \textbf{then}\ t'_i.$$

In general terms, P_s does the following: First, it executes P_ψ. If the state computed by P_ψ did not satisfy ψ ($y = 0$), then it executes P_{np}, which is not portable. If the execution of P_{np} is consistent with $\mathcal{M}_T$ but not $\mathcal{M}_S$ ($z = 1$), then P_s pretends that the formula was satisfied by setting y to 1. Afterward, $y_1 \dots y_m$ are set to 1, so that their former assignment checked in P_ψ is no longer given by the computed state.

Lemma 26. *For every assignment A of $x_1 \dots x_n$ holds*

$$\exists X \in cons_{\mathcal{M}_T}(P_s) : \ state(X) = \sigma[A; y, y_1 \dots y_m \leftarrow 1].$$

Proof. According to Lemma 25, the following holds: For every assignment A of $x_1 \dots x_n$ and A' of $y_1 \dots y_m$, there is an execution X of P_ψ consistent with $\mathcal{M}_T$, such that either $state(X) = \sigma[A \cup A'; y \leftarrow 1]$ or $state(X) = \sigma[A \cup A'; y \leftarrow 0]$. We examine both cases:

Case 1: If $state(X) = \sigma[A \cup A'; y \leftarrow 1]$, then $A \cup A' \vDash \psi$ and according to Lemma 25 there is an execution of P_ψ consistent with SC that computes $\sigma[A \cup A'; y \leftarrow 1]$. We can easily extend this to an execution X' (represented by an interleaving) of P_s that is consistent with SC. After P_ψ is executed, we read 1 with reads $r_y \leftarrow y$ of all threads. So the subsequent if conditions are not satisfied. Then we execute the writes in Line 11. So $y_1 := r_{c,1} \dots y_m := r_{c,1}$; are the only writes executed outside of P_ψ. These writes are ordered after the assignments of 0 to $y_1 \dots y_m$ in *co* and thus $state(X')(y_i) = 1$ for $i \leq m$. Since there are no further executed writes outside of P_ψ, the computed state otherwise coincides with $\sigma[A \cup A'; y \leftarrow 1]$ computed by X. The extension of X computes $\sigma[A; y, y_1 \dots y_m \leftarrow 1]$.

Case 2: If $state(X) = \sigma[A \cup A'; y \leftarrow 0]$, then write $y := r_{c,0}$ is executed in t_1. We construct an execution $X' \supset X$ of P_s in the following way: We ensure all reads $r_y \leftarrow y$ of threads t_i^s with $i \leq k$ read from $y := r_{c,0}$ and thus all threads t_i' are executed. This means P_{np} is executed. According to the definition of P_{np}, there is an execution Y of P_{np} consistent with $\mathcal{M}_T$ such that $state(Y)(z) = 1$ is written by some write w_z in t_1' and Y computes the initial value for all other locations of P_{np}. We enforce $X' \supseteq Y$ and ensure the read $r_z \leftarrow z$ reads from the write in P_{np} ($z = 1$) and thus the following if condition is satisfied and the writes $z := 3$ and $y := 1$ are executed. We order them last in *co*. It follows that X' computes $\sigma[A; y, y_1 \dots y_m \leftarrow 1]$

We show that X' is still consistent with $\mathcal{M}_T$. We partition the events of X' into sets E_1, E_2 and E_3 and show that there is no violating cycle of $\mathcal{M}_T$ inside or between these sets.

Set E_1 consists of events in X and the subsequent reads $r_y \leftarrow y$ of all threads. Set E_2 consists of the events of Y. Set E_3 contains the events $r_z \leftarrow z$ in t_1 and the following writes $z := r_{c,3}$, $y := r_{c,1}$ and $y_1 := r_{c,1}; \cdots y_m := r_{c,1}$.

The following two conditions hold: (i) E_2 is only left by basic relations leading to E_3. This is the case since E_1 precedes E_2 in the program order and has no common registers. (ii) There are no basic relations leading from E_3 to some other set: The events in E_3 are ordered last in the program order and the writes are ordered last in the coherence order. The read $r_z \leftarrow z$ has no outgoing fr relation, since there is only one write to z.

From (i) and (ii) follows that X' contains no basic cycle that contains events from more than one of the sets. According to the path property (Lemma 21) exists no violating cycle for $\mathcal{M}_T$ in X' that contains events from more than one of the sets. So any violating cycle must be contained entirely in one of the sets.

From (i) and (ii) follows (iii): If two events e_1, e_2 are in the same set, then there is no basic path from e_1 to e_2 that leaves the set.

We consider E_1: There are read from relations from $y := r_{c,0}$ in t_1 to all reads $r_y \leftarrow y$ in t_i^s with $i \leq k$. However, there are no outgoing basic relations from any of those reads to other events in E_1. It follows from (iii) that there is no basic path from a read $y := r_{c,0}$ to another event in E_1 and according to the path property, there is no relation from a read $y := r_{c,0}$ to some other element in E_1. It follows that a read $r_y \leftarrow y$ cannot occur in a violating cycle. From (iii) follows that the basic paths in X' between events of E_1 (and thus the violating cycles for $\mathcal{M}_T$) are the same as in X. Since X is consistent with $\mathcal{M}_T$, it has no violating cycle for $\mathcal{M}_T$ and thus X' has no violating cycle for $\mathcal{M}_T$ in E_1.

We consider E_2: From (iii) follows that the basic paths in X' between events in E_2 are the same as in Y. According to Lemma 22, X' has a violating cycle for $\mathcal{M}_T$ in E_2 iff Y has a violating cycle for $\mathcal{M}_T$. The execution Y is defined to be $\mathcal{M}_T$-consistent so there is no violating cycle in E_2.

We consider E_3: The set contains no basic cycle and no basic relation leaves E_3 according to (ii). It follows that there is no basic cycle that contains events of E_3. According to the path property, there is no violating cycle in E_3.

We examine possible violations of Constraint (18). The relation *rmw* can only occur between some pair of events e_1, e_2 in E_2. A violation additionally requires a path $e_1 \xrightarrow{fre} e_3 \xrightarrow{coe} e_2$. Here, e_3 cannot be contained in E_2 since Y is consistent with $\mathcal{M}_T$ and it is not contained in another set according to Condition (iii). It follows that X' is consistent with $\mathcal{M}_T$.

□

Lemma 27. *There is an execution of P_s consistent with $\mathcal{M}_S$ that computes $\sigma[A; y, y_1 \ldots y_m \leftarrow 1]$ with some assignment A of $x_1 \ldots x_n$ iff there is an assignment A' of $y_1 \ldots y_m$ such that $A \cup A' \models \psi$.*

Proof. If a program P' is contained in a program P, we can restrict an execution X of P to an execution of P'. We simply remove all events and relation on events that are not in P'. We denote X restricted to P' as $X[P']$.

($\Rightarrow$): Let X be an execution of P_s consistent with $\mathcal{M}_S$ that computes $state(X)(y) = 1$. We assume towards contradiction that X contains a write that sets z to 1. According to the definition of P_{np} the execution $X[P_{np}]$ contains a violating cycle or a violation of Constraint (18) for $\mathcal{M}_S$. As with the proof of Lemma 26, we can show that no basic path in X between events of $X[P_{np}]$ leaves $X[P_{np}]$. It follows that the basic paths between events in $X[P_{np}]$ are the same in X and $X[P_{np}]$ and thus X also contains the violation of $\mathcal{M}_S$. This is a contradiction to X being consistent with $\mathcal{M}_S$.

It follows that X cannot read $z = 1$ in Line 5 and the write in Line 8 is not executed. So $y := r_{c,1}$ must have been executed in P_ψ for X to compute $y = 1$.

The execution X satisfies uniproc since $\mathcal{M}_S$ is common. It has no basic cycle that violates uniproc. It follows that $X[P_\psi]$ has no such basic cycle either and thus satisfies uniproc. It follows from Lemma 24, that $X[P_\psi]$ computes a satisfying assignment of $x_1 \dots y_m$. Since nothing is written to $x_1 \dots x_n$ outside of P_ψ, X computes the same values for $x_1 \dots x_n$ as $X[P_\psi]$. A write $y_i \leftarrow r_{c,0}$ in t_1 is related to write $y_i \leftarrow r_{c,1}$ in Line 11 of t_1^s in $po \cap loc$ and according to uniproc also in co. This implies that X computes 1 for $y_1 \dots y_m$. For the synchronization variables of P_ψ, uniproc ensures that X computes the initial values.

It follows, that if there is an execution consistent with $\mathcal{M}_S$ that computes $\sigma[A; y, y_1 \dots y_m \leftarrow 1]$ for some assignment A of $x_1 \dots x_n$, then there is an assignment A' of $y_1, ..y_m$ such that $A \cup A' \vDash \psi$.

($\Leftarrow$): According to Lemma 25, for each assignment A of $x_1 \dots x_n$ and A' of $y_1 \dots y_m$ where $A \cup A'$ satisfies ψ, there is an execution X of P_ψ consistent with SC such that $state(X) = \sigma[A \cup A'; y \leftarrow 1]$. This SC execution of P_ψ can be represented as an interleaving of the local executions. To this interleaving, we append the subsequent reads $r_y \leftarrow y$. They read the value 1 and the next if conditions are not satisfied. Then we append the remaining writes in Line 11. The resulting interleaving represents an execution X' of P_s consistent with SC.

The only writes not in P_ψ that are executed by X' are in Line 11 (they are last in the program order and thus also last in $po \cap sloc$) and according to uniproc, they must be ordered after any write $y_i := 0$ in co. It follows $state(X') = \sigma[A; y, y_1 \dots y_m \leftarrow 1]$. □

Theorem 23. *State-based portability is Π_2-hard for common memory models.*

Proof. Given common memory models $\mathcal{M}_S$ and $\mathcal{M}_T$, we argue that P_s is a reduction of validity of a formula $\forall x_1 \dots x_n \exists y_1 \dots y_m : \psi(x_1 \dots y_m)$ to state portability from $\mathcal{M}_S$ to $\mathcal{M}_T$.

Note that any execution of P_s satisfying uniproc computes 1 for $y_1 \dots y_m$ and the initial values for the synchronization variables. We show that the following properties (i)-(iii) hold:

(i) Any state with $y = 2$ that is computable with an execution of P_s consistent with $\mathcal{M}_T$ can be computed by an execution of P_s consistent with $\mathcal{M}_S$. It is easy to see that any state with $y = 2$ that can be computed by an execution satisfying uniproc, can be computed by an execution consistent with SC and thus also $\mathcal{M}_S$.

(ii) Any state with $y = 0$ that is computed by an execution X of P_s consistent with $\mathcal{M}_T$ can be computed by an execution consistent with $\mathcal{M}_S$. No event of P_{np} is executed, the inital value is computed for locations of P_{np}. According to uniproc, any such state has the form $\sigma[A; y \leftarrow 0; y_1 \dots y_m \leftarrow 1]$ for some assignment A of $x_1 \dots x_n$. It is easy to see that A and $y = 0$ is computed by $X[P_\psi]$ and $X[P_\psi]$ satisfies *uniproc.* According to Lemma 24, there is an assignment A' of $y_1 \dots y_m$ such that $A \cup A' \nvDash \psi$. It follows from Lemma 25 that there is an execution of P_ψ consistent with SC that computes $\sigma[A \cup A'; y \leftarrow 0]$. This can be easily extended to an execution of P_s consistent with SC that computes $\sigma[A; y \leftarrow 0; y_1 \dots y_m \leftarrow 1]$. Since $\mathcal{M}_S$ is common, $\sigma[A; y \leftarrow 0; y_1 \dots y_m \leftarrow 1]$ is also computable under $\mathcal{M}_S$.

(iii) Any state with $y = 1$ computed by an execution X consistent with $\mathcal{M}_T$ has the form $\sigma[A; y, y_1 \dots y_m \leftarrow 1]$ for some assignment A of $x_1 \dots x_n$. This holds since $X[P_{np}]$ is also consistent with $\mathcal{M}_T$ (the basic paths between its events are the same as in X) and according to the definition of P_{np}, it computes the initial value for all its locations except maybe 1 for z. If that is the case, then the write $z \leftarrow r_{c,3}$ in t_1^s is executed and according to uniproc ordered later in *co*.

From (i)-(iii) follows that P_s is state portable from $\mathcal{M}_S$ to $\mathcal{M}_T$ iff any $\mathcal{M}_T$-computable state $\sigma[A; y, y_1 \dots y_m \leftarrow 1]$ can be computed under $\mathcal{M}_S$. Lemma 26 implies that P_s is state portable from $\mathcal{M}_S$ to $\mathcal{M}_T$ iff any state $\sigma[A; y, y_1 \dots y_m \leftarrow 1]$ can be computed under $\mathcal{M}_S$. According to Lemma 27, this is the case iff for every assignment A of $x_1 \dots x_n$ there is an assignment A' of $y_1 \dots y_m$ with $A \cup A' \vDash \psi$. This is the case iff $\forall x_1 \dots x_n \exists y_1 \dots y_m : \psi(x_1 \dots y_m)$ is valid. □

B.4 Trace Portability Experiments

Benchmark	SC-TSO	SC-PSO	SC-Power	TSO-PSO	TSO-Power	PSO-Power	RMO-Alpha	RMO-Power	Alpha-RMO	Alpha-Power
Bakery	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
Bakery x86	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Bakery Power	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓
Burns	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓
Burns x86	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
Burns Power	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓
Dekker	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
Dekker x86	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Dekker Power	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓
Lamport	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Lamport x86	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗
Lamport Power	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Parker	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
Parker x86	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓
Parker Power	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
Peterson	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
Peterson x86	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Peterson Power	✓	✗	✓	✗	✓	✓	✓	✓	✗	✓
Szymanski	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓
Szymanski x86	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
Szymanski Power	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓

Table B.1: Bounded trace portability analysis of mutual exclusion algorithms: trace portable (✓), non trace portable (✗)

This section presents the complete set of experiments for trace portability of mutual exclusion algorithms. Besides the trace portability analysis between

Benchmark	TSO ⊆ SC					
	SAT	HERD7	PORTHOS	Δ	IT	S.U.
PARKER	✔	**0.15**	0.70	3	1	0.21
DEKKER	✔	T/O	**12.31**	-	1	>**292.44**
PETERSON	✔	9.96	**1.31**	12	1	**7.60**
BURNS	✔	610.65	**2.00**	53	1	**305.32**
BAKERY	✔	T/O	**10.78**	-	2	>**333.95**
LAMPORT	✔	T/O	**10.64**	-	3	>**338.34**
SZYMANSKI	✔	T/O	**101.32**	-	1	>**35.53**

Benchmark	POWER ⊆ TSO					
	SAT	HERD7	PORTHOS	Δ	IT	S.U.
PARKER	✔	**0.15**	2.46	3	2	0.06
DEKKER	✘	T/O	**108.89**	-	0	>**33.06**
PETERSON	✘	9.94	**6.33**	0	0	**1.57**
BURNS	✘	578.55	**6.12**	18	1	**94.53**
BAKERY	✘	T/O	**836.44**	-	43	>**4.30**
LAMPORT	-	T/O	T/O	-	-	-
SZYMANSKI	✘	T/O	**940.75**	-	0	>**3.82**

Benchmark	ARM ⊆ TSO					
	SAT	HERD7	PORTHOS	Δ	IT	S.U.
PARKER	✔	**0.15**	1.90	3	1	0.07
DEKKER	✘	T/O	**134.43**	-	0	>**26.77**
PETERSON	✘	10.28	**6.51**	0	0	**1.57**
BURNS	✘	546.90	**7.89**	18	1	**69.31**
BAKERY	-	T/O	T/O	-	-	-
LAMPORT	-	T/O	T/O	-	-	-
SZYMANSKI	✘	T/O	**850.44**	-	0	>**4.23**

Table B.2: State inclusion of mutual exclusion algorithms.

SC, TSO and Power shown on Table 5.1, we report on several combinations of memory models. The results of the portability analysis are shown in Table B.1.

B.5 Experiments without Relation Analysis

We are performing experiments to compare a previous version of PORTHOS without relation analysis against HERD7. The speed up is significantly lower than the one in Table 5.2

We are checking the inclusions TSO ⊆ SC, POWER ⊆ TSO, and ARM ⊆ TSO. Inclusion in the other direction (necessary for equivalence) holds by the definition of the memory models. E.g., every state reachable under TSO is also reachable under the weaker models POWER and ARM.

The results are given in Table B.2. The SAT column reports whether a counterexample to inclusion was found (✔) or not (✘). When HERD7 returns a result, we report on the number of delta executions (Δ). This corresponds to an upper bound on the maximal number of iterations PORTHOS might perform. As it can be seen from Table B.2, in general this number is several orders of magnitude smaller than the total number of executions. The cases reporting zero iterations correspond to the set of executions coinciding for both memory models. For most of the cases, PORTHOS is at least one order of magnitude faster than HERD7. For TSO, the speed-up (S.U. column) can be up-to two orders of magnitude.

Appendix C

Future Work in Memory Model Synthesis

Differentiating between Models We assume that the input programs Pos and Neg are well chosen to highlight different behavioral aspects of the memory model we wish to synthesize and that the equivalence class they induce contains useful approximations of the model.

Here, we might want to know whether a chosen input (Pos, Neg) is sufficient to define a semantically unique CAT memory model. If this is not the case, we want to know where ambiguity is introduced This leads us to the following problem: Is there a model $\mathcal{M}' \in [\mathcal{M}]$ such that $L(\mathcal{M}') \neq L(\mathcal{M})$? This problem is addressed in [39].

A possible approach for a semi-decider is as follows. After the model synthesis has returned a model $\mathcal{M}$, we continue the synthesis until a syntactically different model $\mathcal{M}'$ is returned. Then, we need to decide whether $L(\mathcal{M}') = L(\mathcal{M})$ holds. An acylicity constraint on a derived relation induces a language over base relations that form the labels on cycles. For instance, $acyclic(po;co)$ forbids all cycles labeled $(po.co)^+|(co.(po.co)^*.po)$.

We can compare these languages in order to find a cycle that is only possible under one model. It remains to decide whether there exists an annotated program that only succeeds under one model because it forms such a cycle.

Equivalence Checking Further methods of eliminating equivalent relations could be applied to the synthesis. This can be achieved by comparing regular expressions. In some cases, applying a certain operator has no effect. For example $(po|rf)\&po$ is equivalent to po. Applying union or intersection to relations r_1 and r_2 only produces a relation that is not equivalent to r_1 or r_2 if they are both incomparable with respect to set inclusion. This property can be used as a basis for an fast method to check relations for equivalence. A fast method to indicate if two relations are comparable could be worth the trade-off in efficiency.

Targeted Synthesis In the expansion by enumeration, we list every relation in a non-targeted way. Instead, we can modify the expansion phase in order to produce "better" relations. We attempt to use checks that succeed for some execution in $X(\textsc{Neg})$ or fail for some P_+ in order to move towards relations without that behavior. We can identify circles of base relations in an execution and given such a cycle, we may attempt to construct relations that forbid it. This approach differs from the CEGIS method insofar that it is done without using an SMT solver and thus potentially faster. The relations are constructed such that they forbid a certain cycle.

If an execution $X \in X(\textsc{Neg})$ succeeds for a relation r, then X contains no cycle of r. The synthesized model however should contain a relation that has a cycle in X. Any constraint that violates cycles in common memory models implies a cycle of base relations (see Lemma 21). It follows that we can assume X contains a cycle of base relations described by a word $b_1...b_n$ over base relation names. We can now construct a relation r' that forbids X and contains r ($r' \supseteq r$). We know that a check still fails for r' when it fails for r (Theorem 14). Every relation r can be interpreted as a regular expression over base (and named) relations and describes a language $L(r)$. We require $r \subseteq r'$ and that there is an $i \leq n$ such that $b_{i+1} \dots b_n.b_1 \dots b_i \in L((r')^+)$. There are many relations that satisfy such a criteria. We would need to aim for a balance between closeness to r and simplicity.

The other direction is analogue. If a program $P_+ \in \textsc{Pos}$ fails for a relation r, then there is an execution X that reaches a state that satisfies S and that contains a cycle of r. We can find such an execution and the corresponding cycle c' with a cyclicity query. The synthesized model however should contain a relation that describes no cycle in X.

Any cycle that violates constraints in common mcms implies a cycle of base relations (see Lemma 21). It follows that X contains a cycle of base relations that contains the events of c' and is described by a word $c = b_1...b_n$. We now need to construct a relation r' such that $r' \subset r$ and for every such cycle c and all $i \leq n$ holds $b_{i+1} \dots b_n.b_1 \dots b_i \notin L((r')^+)$. We find such a relation by using the property that every edge in c' implies a path of base relations between the same events. For every edge $e \xrightarrow{r} e' \in c'$ there is a path $e \xrightarrow{b_1} \dots \xrightarrow{b_m} e' \in c$ such that $b_1...b_m \in L(r)$. We pick one such word $b_1...b_m$ and ensure that $b_1...b_m \notin L(r')$.

Appendix D

Petri Net Invariant Synthesis

D.1 Proof of Theorem 15

Theorem 15. *A half space (k, c) is t-inductive iff*

$$\nexists x \in \mathbb{N}^n : c \leq k \cdot x + k \cdot t^- < c - k \cdot t^\Delta$$

Proof. We use the following property:

$$\exists m \in \mathbb{N}^n : m \geq t^- \Leftrightarrow \exists x \in \mathbb{N}^S : m = x + t^-. \tag{D.1}$$

According to the definition of t-inductivity, an inequality is not t-inductive iff there is a marking m that satisfies the following condition:

$$\begin{aligned} &\exists m \in \mathbb{N}^n : k \cdot m \geq c \wedge m \geq t^- \wedge k \cdot m + k \cdot t^\Delta < c \\ \overset{(D.1)}{\Leftrightarrow}\ &\exists x \in \mathbb{N}^n : k \cdot x + k \cdot t^- \geq c \wedge k \cdot x + k \cdot t^- + k \cdot t^\Delta < c \\ \Leftrightarrow\ &\exists x \in \mathbb{N}^n : k \cdot x + k \cdot t^- \geq c \wedge k \cdot x + k \cdot t^- < c - k \cdot t^\Delta \\ &\exists x \in \mathbb{N}^n : c \leq k \cdot x + k \cdot t^- < c - k \cdot t^\Delta \end{aligned}$$

□

D.2 Proof of Lemma 15

First, we require the following lemma which examines the non-trivial case:

Lemma 28. *Let k be an non-mixed vector with $k \cdot t^\Delta < 0$ and $|k(i)| > -k \cdot t^\Delta$ for all $k(i) \neq 0$. Then there is a $c \in \mathbb{Z}$ so that $k \cdot m \geq c$ is t-inductive.*

Proof. We examine both cases for the non-mixed vector k: $k \geq 0$ and $k \leq 0$.

$k \geq 0$: First, we assume $k \geq 0$ and set $c := k \cdot t^- + 1$. From $k \cdot t^\Delta < 0$ and $|k(i)| > -k \cdot t^\Delta$ follows $k(i) > 1$ for all $k(i) \neq 0$. We check if the condition in Theorem 15 is satisfied for any $x \in \mathbb{N}^n$:

$$c \leq k \cdot x + k \cdot t^- < c - k \cdot t^\Delta.$$

If x is the vector 0^n then it follows $c = k \cdot t^- + 1 \nleq k \cdot x + k \cdot t^- = k \cdot t^-$. Let x contain an entry $x(i) > 0$. If $k(i) = 0$ for all such entries then the case is analogue to $x = 0^n$. If there is an $i \leq n$ with $x(i) > 0$ and $k(i) > 0$ (and thus $k(i) \geq -k \cdot t^\Delta + 1$), then it holds

$$k \cdot x + k \cdot t^- \geq k(i) + k \cdot t^- \geq -k \cdot t^\Delta + 1 + k \cdot t^- = c - k \cdot t^\Delta.$$

$k \leq 0$: We define $c := k \cdot t^- + k \cdot t^\Delta$. If x is the vector 0^n, then it follows $k \cdot x + k \cdot t^- = k \cdot t^- \nless kt^- = c - k \cdot t^\Delta$. Let x contain an entry $x(i) > 0$. If $k(i) = 0$ for all such entries then the case is analogue to $x = 0^n$. If there is an $i \leq n$ with $x(i) > 0$ and $k(i) < 0$ (and thus $k(i) < k \cdot t^\Delta$), then it holds

$$k \cdot x + k \cdot t^- \leq k(i) + k \cdot t^- < k \cdot t^\Delta + k \cdot t^- = c.$$

This means the condition is not satisfied in either case and the inequality is t-inductive according to Theorem 15. □

We recall the Theorem 15 from Section 7.5:

Lemma 15. *There is a $c \in \mathbb{Z}$ such that (k, c) is a separating t-inductive inequality iff k is a solution of ϕ_t.*

Proof. Now, we examine the formula ϕ. Let (k, c) be separating, meaning $m_0 \in \mathrm{Sol}(k, c)$ and $m_0 \notin \mathrm{Sol}(k, c)$, and t-inductive. Since it is separating, condition (0) holds and c is in the interval $(k \cdot m_f, k \cdot m_0]$. If it is trivial, it follows that one of the conditions (1)-(3) holds. If it is non-trivial, conditions (4) and (5) hold. It follows that k is a satisfying assignment of ϕ_t.

Let k be a solution of ϕ_t. We show that there is a value c such that (k, c) is t-inductive:

- If k satisfies (1), it is t-inductive for any c and since (0) holds, we can choose a c such that it is separating.
- If k satisfies (2) then we set $c = k \cdot m_0$ and thus (k, c) is separating and antitone.
- Condition (3) is analogue, here we set $c = k \cdot m_f + 1$ and thus (k, c) is separating and monotone.
- If k satisfies Conditions (4) and (5), than the property holds according to the following Lemma.

□

D.3 Proofs for Algorithm 7

We recall Theorem 19 from Section 7.6

Theorem 19. *A half space (k, c) is t-inductive iff*

$$\nexists k_1..k_l \in K^* : c \leq k \cdot t^- + \sum_{i=1}^{l} k_i < c - k \cdot t^\Delta$$

Proof. We use the following property: For any sum $\sum_{i=1}^{l} k_i$ over K with $x(j)$ the number of occurrences of value k_j in the sum, it holds

$$\sum_{i=1}^{l} k_i = \sum_{j=1}^{n} k(j) \cdot x(j) = k \cdot x. \tag{D.2}$$

Obviously, we can construct a sum with value $k \cdot x$ for any vector x. It follows:

$$\exists x \in \mathbb{N}^n : k \cdot x = z \Leftrightarrow \exists k_1 \dots k_l \in K^* : \sum_{i=1}^{l} k_i = z \tag{D.3}$$

An inequality is not t-inductive iff there is a marking x that violates Theorem 15:

$$\exists x \in \mathbb{N}^n : c \leq k \cdot x + k \cdot t^- < c - k \cdot t^\Delta$$
$$\overset{(D.3)}{\Leftrightarrow} \exists k_1 \dots k_l \in K^* : c \leq \sum_{i=1}^{l} k_i + k \cdot t^- < c - k \cdot t^\Delta$$

□

Lemma 29. *The algorithm is correct.*

Proof. Assume the algorithm returns "Not inductive". It holds $c \leq \mathit{current} < c - k \cdot t^\Delta$ and *current* was derived by adding elements of K to the starting value $k \cdot t^-$. It follows that there is a sequence $k_1 \dots k_l \in K^*$ and $c \leq k \cdot t^- + \sum_{i=1}^{l} k_i < c - k \cdot t^\Delta$. This violates the condition of Theorem 19 and thus the inequality is not t-inductive. □

Lemma 30. *The algorithm is complete.*

Proof. We show that the algorithm identifies any half space that is not t-inductive. We know that k is not mixed. We now examine the case $k \geq 0$. Let $k_1 \dots k_l$ be a sequence that satisfies the condition of Theorem 19. W.l.o.g., we can assume that $k_1 \dots k_l$ is the shortest sequence that reaches the target area. We examine its prefixes $k_1 \dots k_i$ with $i < l$. It follows that $k \cdot t^- + \sum_{j=1}^{i} k_j < c$ holds for all $i < l$ and $k \cdot t^- + \sum_{j=1}^{l} k_j < c - k \cdot t^\Delta$.

We now apply a induction over the sequence to show that the algorithm processes $k \cdot t^- + \sum_{j=1}^{l} k_j$ or returns "Not inductive" before that:

Induction basis: The algorithm processes $k \cdot t^-$ (Line 1 and 2).

Induction hypothesis: The algorithm processes $current = k \cdot t^- + \sum_{j=1}^{i} k_j$.

Induction step: We know $current + k_{i+1} = k \cdot t^- + \sum_{j=1}^{i+1} k_j < c - k \cdot t^\Delta$ holds and thus the condition in Line 9 is satisfied. It is added to the queue (Line 12) and it is either processed later or the algorithm returns "Not inductive" before that. The argument is analogue for $k \leq 0$. Here, the condition in Line 10 is satisfied.

It follows that unless "Not inductive" is returned earlier, $k \cdot t^- + \sum_{j=1}^{l} k_j$ is processed. It meets the condition in Line 5 and the algorithm returns "Not inductive". □

Theorem 24. *The run-time of Algorithm 7 is polynomial in the input values.*

Proof. We show that the algorithm's runtime is polynomial in the input values. The processing time of one value is linear in $|K|$. Either the lowest processed value is the starting value $k \cdot t^-$ and the highest is some value at most $c - k \cdot t^\Delta$ (garanteed by the condition in Line 9) or the highest processed value is the starting value $k \cdot t^-$ and the lowest is at least c (garanteed by the condition in Line 10). It follows that the algorithm only processes values in the polynomial sized segment

$$[min(k \cdot t^-, c), max(k \cdot t^-, c - k \cdot t^\Delta)].$$

Since every processing step reaches a new unprocessed value in the segment, the number of processing steps are limited by the seqment size

$$l_s := |max(k \cdot t^-, c - k \cdot t^\Delta) - min(k \cdot t^-, c)|.$$

□

We continue by analyzing the space-complexity of the problem. We study the complexity class **L** which denotes problems that can be solved deterministically using an amount of memory space that is logarithmic in the size of the input. The class **NL** describes problems that can be solved non-deterministically in logarithmic space. The problems that can be solved non-deterministically using space that is linear in the input size are in **CSL**. For any class of problems **C**, we denote the class of their complements as **co-C**.

Theorem 25. *A half space is not t-inductive if and only if there is a vector x that violates Theorem 15 and $x(j) \leq l_s$ for all $j \leq n$*

Proof. Since there are l_s many possible values, and the algorithm processes a value only once, the algorithm constructs a sum of length at most l_s iff it is not t-inductive. It follows from (Equation D.2) that if (Theorem 15) is violated, than it is violated by some vector x with $|x| \leq l_s$. □

It follows from Theorem 25 that deciding inductivity is in co-NP. Non-deterministically choosing some vector x with values at most l_s and checking if it violates Theorem 15 takes polynomial time.

We assume the dimension n of k (which is the number of places in the Petri net) is a fixed parameter and introduce a new algorithm that solves t-inductivity in logarithmic space. It simply iterates all possible vectors x that satisfy $x(j) \leq l_s$ for all $j \leq n$ and checks whether they satisfy the inequality. The successor function *succ* handles the vector $x \leq l_s$ like a number with n digits to the basis l_s and works like a standard successor. It starts at the first value and if it is less than l_s it adds one and terminates, if the current value is l_s, it sets it to 0 and handles the next one.

```
Inductivity-LogSpace
x = 0^n;
repeat
    if $c \leq k \cdot x + k \cdot t^- < c - k \cdot t^\Delta$ then
        return Not inductive
    end
    $x = succ_{l_s}(x)$;
until $x = l_s^n$;
return Inductive
```

The value of l_s is linear in every input variable and thus it can be stored in logarithmic space. Any vector x with $x(j) \leq l_s$ for all $j \leq n$ can also be stored in logspace.

Theorem 26. *Deciding inductivity of a half space is in **L** for unary encoded input and fixed dimension of k.*

A nondeterministic version of the inductivity algorithm has to store only the current value and the number of executed steps and it executes at most l_s steps.

Theorem 27. *Deciding inductivity of a half space is in **co-NL** for unary encoded input.*

For binary encoded input, it follows from Theorem 27:

Theorem 28. *Deciding inductivity of a half space is in **co-CSL** for binary encoded input.*

For an instance of the inductivity problem given by an an inequality and a transition we introduce the parametrized instance with the greatest total value k_{max} of k as the parameter.

Theorem 29. *The parametrized inductivity problem is fixed parameter tractable.*

Proof. For any vector $x \in \mathbb{N}^n$ with $x_i \geq k_1$ it holds

$$k \cdot x = k \cdot (x_1 + k_i, \ldots, x_{i-1}, x_i - k_1, x_{i+1}, \ldots, x_n)^T$$

We iterate this argument and it follows that if there is a vector that satisfies the condition of Theorem 15 then it is also satisfied by a vector x with $x_2, \ldots x_n \leq k_1$. We assume $k \geq 0$ and $k \cdot t^- < c$.

$$k \cdot t^- + k \cdot x = k \cdot t^- + k_1 \cdot x_1 + \sum_{i=2}^{n} k_i \cdot x_i \geq c$$

$$\Rightarrow k_1 \cdot x_1 \geq c - k \cdot t^- - \sum_{i=2}^{n} k_i \cdot x_i \geq c - k \cdot t^- - k_1 \cdot \sum_{i=2}^{n} k_i$$

$$\Rightarrow x_1 \geq \lceil \frac{c - k \cdot t^- - k_1 \cdot \sum_{i=2}^{n} k_i}{k_1} \rceil$$

Instead of imposing a lower bound on x_1 we introduce x' with $x_1 = x'_1 - \lceil \frac{c-k\cdot t^- - k_1 \cdot \sum_{i=2}^{n} k_i}{k_1} \rceil$ and $x'_i = x_i$ for $i > 1$:

$$k \cdot x = k \cdot x' + k_1 \cdot \lceil \frac{c - k \cdot t^- - k_1 \cdot \sum_{i=2}^{n} k_i}{k_1} \rceil$$

It follows that an inequality $k \cdot m \geq c$ is t-inductive iff the following inequality is t-inductive: $k \cdot m \geq c - k_1 \cdot \lceil \frac{c-k\cdot t^- - k_1 \cdot \sum_{i=2}^{n} k_i}{k_1} \rceil$. Note that the size of the new inequality is only bounded by k_{max} and not c:

$$c - k_1 \cdot \lceil \frac{c - k \cdot t^- - k_1 \cdot \sum_{i=2}^{n} k_i}{k_1} \rceil \leq k \cdot t^- + k_1 \cdot \sum_{i=2}^{n} k_i$$

The construction for $k \leq 0$ and $k \cdot t^- > c$ is analogue. □

D.3.1 Proofs for Generating c

The main contribution of this subsection is the proof of Theorem 22. This requires the following two technical lemmas.

Lemma 31.

$$|\gcd k| \leq |k \cdot t^{\Delta}|$$

Proof. It holds $k \cdot t^{\Delta} = t^{\Delta}(1) \cdot k(1) + \ldots + t^{\Delta}(n) \cdot k(n) = z \cdot \gcd k$ for some $z \in \mathbb{Z}$. It follows $|k \cdot t^{\Delta}| \geq |\gcd k|$. □

Lemma 32. *Let $k \cdot m \geq c$ be a non-trivial t-inductive inequality and let $y \in \mathbb{N}$ denote the Frobenius number of $\frac{k(1)}{gcd(k)}, \ldots, \frac{k(n)}{gcd(k)}$.*

a) If $k \geq 0$, it holds $k \cdot t^- + k \cdot t^{\Delta} < c \leq gcd(k) \cdot y + k \cdot t^-$.

b) If $k \leq 0$, it holds $gcd(k) \cdot y + k \cdot t^- \leq c < k \cdot t^-$.

Proof. We prove the lower and upper bounds for both cases.

a) The lower bound follows immediately from Lemma 13. For the upper bound, we assume $c > gcd(k) \cdot y + k \cdot t^-$ and thus $\frac{c-k\cdot t^-}{gcd(k)} > y$. Since y is the Frobenius number, there is a vector $b \in \mathbb{N}^n$ such that $\lfloor \frac{c-kt^-}{gcd(k)} \rfloor = \frac{k^T}{gcd(k)} \cdot b$. This means

$$c - k \cdot t^- \leq k \cdot b < c - k \cdot t^- + gcd(k).$$

According to Lemma 13, it holds $k \cdot t^{\Delta} < 0$. We apply Lemma 31 and get $-k \cdot t^{\Delta} \geq gcd(k)$. This means that $c \leq k \cdot b + kt^{-} < c - k \cdot t^{\Delta}$ holds and thus the vector b satisfies Theorem 15. This is a contradiction to inductivity.

b) The upper bound follows immediately from Lemma 13. For the lower bound, we assume $c < gcd(k) \cdot y + k \cdot t^{-}$ and thus $\frac{c-kt^{-}}{gcd(k)} > y$. The remainder is analogue to a).

□

We recall Theorem 22 from Section 7.6.1:

Theorem 22. *Let $k \cdot m \geq c$ be a non-trivial t-inductive half space and let k_{max}, k_{min} denote entries of k with maximal and minimal absolute values.*

1. *If $k \geq 0$, then it holds that $c < k_{max} \cdot k_{min} + k \cdot t^{-}$.*
2. *If $k \leq 0$, then it holds that $c \geq -k_{max} \cdot k_{min} + k \cdot t^{-}$.*

Proof. Let y denote the Frobenius number of $\frac{k(1)}{\mathrm{gcd}(k)}, \dots, \frac{k(n)}{\mathrm{gcd}(k)}$. By Lemma 14 and Lemma 31, we know that $|k(i)| > -k \cdot t^{\Delta} \geq |\mathrm{gcd}(k)|$. From this we obtain that $\frac{k(i)}{\mathrm{gcd}(k)} = \frac{|k(i)|}{|\mathrm{gcd}(k)|} \geq 2$. Now we apply Theorem 21 and get: $y \leq (\frac{k_{max}}{\mathrm{gcd}(k)} - 1)(\frac{k_{min}}{\mathrm{gcd}(k)} - 1)$.
Now assume that $k \geq 0$. We give an estimation for $\mathrm{gcd}(k) \cdot y$:

$$\begin{aligned}
\mathrm{gcd}(k) \cdot y &\leq \mathrm{gcd}(k) \cdot (\frac{k_{max}}{\mathrm{gcd}(k)} - 1)(\frac{k_{min}}{\mathrm{gcd}(k)} - 1) \\
&\leq \mathrm{gcd}(k)^2 \cdot (\frac{k_{max}}{\mathrm{gcd}(k)} - 1)(\frac{k_{min}}{\mathrm{gcd}(k)} - 1) \\
&\leq (k_{max} - \mathrm{gcd}(k))(k_{min} - \mathrm{gcd}(k)) \\
&\leq k_{max} \cdot k_{min}.
\end{aligned}$$

We now combine this with the bound proven in Lemma 32 and derive the criterion of Theorem 22

$$c < k_{max} \cdot k_{min} + k \cdot t^{-}.$$

If $k \leq 0$, we derive a similar bound. Note that it holds $\mathrm{gcd}(k) < 0$). We now derive a lower bound of $\mathrm{gcd}(k) \cdot y$:

$$\begin{aligned}
\mathrm{gcd}(k) \cdot y &\geq \mathrm{gcd}(k) \cdot (\frac{k_{max}}{\mathrm{gcd}(k)} - 1) \cdot (\frac{k_{min}}{\mathrm{gcd}(k)} - 1) \\
&\geq -\mathrm{gcd}(k)^2 \cdot (\frac{k_{max}}{\mathrm{gcd}(k)} - 1) \cdot (\frac{k_{min}}{\mathrm{gcd}(k)} - 1) \\
&\geq -(k_{max} - \mathrm{gcd}(k)) \cdot (k_{min} - \mathrm{gcd}(k)) \\
&\geq -k_{max} \cdot k_{min}.
\end{aligned}$$

Like above, we apply this to the bound on c from Lemma 32 and get

$$c \geq -k_{max} \cdot k_{min} + k \cdot t^{-}$$

□

D.4 Alternative Proof for Theorem 17

For an inequality not to be inductive, there must be a marking that satisfies the inequality from which t can be fired. We can describe the set of such markings for emptiness using the following lemma:

Lemma 33.
$$m_t \cap m(k \cdot m \geq c) = \varnothing \Leftrightarrow k \cdot t^- < c \wedge k \leq 0$$

The set is not empty if it contains either t^-, which means $k \cdot t^- \geq c$ holds, or it contains some greater marking m. There must be some positive value of k for the greater m to satisfy the inequality. Otherwise $k \cdot m \leq k \cdot t^-$ would hold. The formal proof is omitted since it is trivial. In the following, we will assume that the set is not empty.

We say a vector is *mixed* if it contains both positive and negative entries. An inequality $k \cdot m \geq c$ is mixed if k is mixed.

We recall Theorem 17 from Section 7.4:

Theorem 17. *If (k, c) is t-inductive but not oriented towards t, then either $k \geq 0$ or $k \leq 0$ holds.*

Proof. Let (k, c) is t-indcutive and not oriented towards t We assume towards contradiction that there are i, j such that $k(i) > 0$ and $k(j) < 0$. We denote the i-th unit vector with e_i. We construct a counterexample marking m such that the following holds:

$$k \cdot m = -k \cdot t^{\Delta} \tag{D.4}$$

Construct a marking m satisfying $k \cdot m = -k \cdot t^{\Delta}$.

```
1  $m := -t^{\Delta} \in \mathbb{Z}$ for $l : m(l) < 0$ do
2      if $k(l) > 0$ then
3          $m := m + |m(l)| \cdot (k(l) \cdot e_j - k(j) \cdot e_l)$
4      end
5      if $k(l) < 0$ then
6          $m := m + |m(l)| \cdot (k(i) \cdot e_l - k(l) \cdot e_i)$
7      end
8      if $k(l) = 0$ then
9          $m := m + |m(l)| \cdot e_l$
10     end
11 end
```

Note that in each iteration, one negative value $m(l)$ becomes positive (or zero) and no value becomes negative. It follows that the algorithm terminates with a marking $m \geq 0$.

After the initial assignment in line 1, Equation D.4 holds. We show that it is actually an invariant. Assume in an iteration $k(l) > 0$ holds, then the update in line 4 is executed:

$$k \cdot m_{new} = k \cdot m_{old} + k \cdot |m_{old,l}| \cdot (k(l) \cdot e_j - k(j) \cdot e_l)$$

$$= k \cdot m_{old} + |m_{old,l}| \cdot (k(l) \cdot k(j) - k(j) \cdot k(l)) = k \cdot m_{old}$$

The case $k(l) < 0$ is analogue and in case $k(l) = 0$, it holds:

$$k \cdot m_{new} = k \cdot m_{old} + k \cdot |m_{old,l}| \cdot e_l = k \cdot m_{old} + |m_{old,l}| \cdot k(l) = k \cdot m_{old}$$

We assume $k \cdot t^- \leq c$. The are $d, e \in \mathbb{N}$ with $k \cdot t^- + d \cdot (-k \cdot t^\Delta) = c + e$ and $e < -k \cdot t^\Delta$. It follows from Equation D.4, that $k \cdot t^- + d \cdot k \cdot m = c + e$ and thus $c \leq k \cdot t^- + k \cdot (d \cdot m) < c - k \cdot t^\Delta$ The marking $d \cdot m$ satisfies the condition of Theorem 15 and thus it is not inductive.

We assume $k \cdot t^- > c$. Since the inequality is relevant, it holds $k \cdot t^\Delta < 0$ and thus there are $d, e \in \mathbb{N}$ with $k \cdot t^- + d \cdot (-k \cdot t^\Delta) = c + e$ and $e < -k \cdot t^\Delta$. If we modify the algorithm such that m is initialized as $m := t^\Delta$ then it is easy to see that the algorithm produces a marking m that satisfies

$$k \cdot m = k \cdot t^\Delta \tag{D.5}$$

We apply this equation and derive $k \cdot t^- + d \cdot k \cdot m = c + e$ and thus $c \leq k \cdot t^- + k \cdot (d \cdot m) < c - k \cdot t^\Delta$. The marking $d \cdot m \in \mathbb{N}$ satisfies the condition of Theorem 15 and thus it is not inductive.

It follows that the inequality is not inductive. □

Appendix E

Curriculum Vitae

Technische Universität Braunschweig
Institut für Theoretische Informatik
Mühlenpfordtstr. 23, 38106 Braunschweig

Personal Information

Name	Florian Wayan Aaron Furbach
Born	1985 in Germany
Citizenship	Germany

Education

12/2012 – to date	**Computer Science PhD**, *Technische Universität Kaiserslautern*, Kaiserslautern, Germany. Pursuing a PhD in Concurrency Theory under supervision of Jun.-Prof. Roland Meyer in the Concurrency Theory group. Title: Verification with Memory Models as Input
04/2009 – 12/2012	**Computer Science MSc**, *Technische Universität Kaiserslautern*, Kaiserslautern, Germany. Grade:1,5 Main Focus: Theoretical Computer Science Minor: Mathematics
04/2005 – 03/2009	**Computer Science BSc**, *Technische Universität Kaiserslautern*, Kaiserslautern, Germany. Grade: 2.3 Main Focus: Theoretical Computer Science Minor: Mathematics
1995 – 2004	**High School Diploma**, *Hilda-Gymnasium*, Koblenz, Germany.

Master Thesis

title	*Automata-theoretic Control for Total Store Ordering Architectures*
supervisors	Roland Meyer, Klaus Madlener

Research Interests

Parallel programming, control synthesis, weak memory models, automata theory, game theory, program verification, Petri nets