A

Memory-Model-Aware Testing - a Unified Complexity Analysis

Florian Furbach, TU Kaiserslautern
Roland Meyer, TU Kaiserslautern
Klaus Schneider, TU Kaiserslautern
Maximilian Senftleben, TU Kaiserslautern

To improve the performance of the memory system, multiprocessors implement weak memory consistency models. Weak memory models admit different views of the processes on their load and store instructions and thus allow for computations that are not sequentially consistent. Program analyses have to take into account the memory model of the targeted hardware. This is challenging due to the fact that numerous memory models have been developed, and every memory model requires its own analysis.

In this article, we study a prominent approach to program analysis: testing. The testing problem takes as input sequences of operations, one for each process in the concurrent program. The task is to check whether these sequences can be interleaved to an execution of the entire program that respects the constraints of a memory model under consideration. 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 processes, the length of their executions, and the number of shared variables.

What differentiates our contribution from related results is a uniform approach that avoids to consider each memory model on its own. We build upon work of Steinke and Nutt. They showed that the existing memory models form a hierarchy where one model is called weaker than another one if it includes the latter's behavior. Using the Steinke-Nutt hierarchy, we develop three general concepts that allow us to quickly determine the complexity of a testing problem. (i) We generalize the technique of problem reductions from complexity theory. So-called range reductions propagate hardness results between memory models, and we apply them to establish **NP** lower bounds for the stronger memory models. (ii) For the weaker models, we present polynomial-time testing algorithms that are inspired by determinization algorithms for automata. (iii) Finally, we describe a single SAT encoding of the testing problem that works for all memory models in the Steinke-Nutt hierarchy to prove their membership in **NP**. Our results are general enough to carry over to future weak memory models. Moreover, they show that SAT solvers are adequate tools for testing.

1. INTRODUCTION

1.1. Weak Memory Models

For multiprocessors, communication over shared memory is a performance bottleneck. To improve processor utilization, architectures implement various optimizations like a distributed shared memory or write buffers. However, with these optimizations, different processes may observe the writes of other processes at different points in time. This results in different local views of the processes on the shared memory. *Weak memory models* [Adve and Gharachorloo 1996; Hennessy and Patterson 2003; Lawrence 1998; Steinke and Nutt 2004] have been developed as an interface to the programmer that abstracts from architectural details. They specify, without reference to the processor, the local views that are possible in a concurrent execution.

This work was supported by the DFG project *R2M2: Robustness against Relaxed Memory Models*.

DOI:http://dx.doi.org/10.1145/0000000.0000000

```
Procedure p                    Procedure q
1 x = 1;                       1 y = 1;
2 while y=1 do                 2 while x=1 do
3 |   x = 0;                   3 |   y = 0;
4 |   Sleep(time);             4 |   Sleep(time);
5 |   x = 1;                   5 |   y = 1;
6 end                          6 end
7 Critical Section;            7 Critical Section;
8 x = 0;                       8 y = 0;
```

Fig. 1. Example procedures implementing Dekker's mutex.

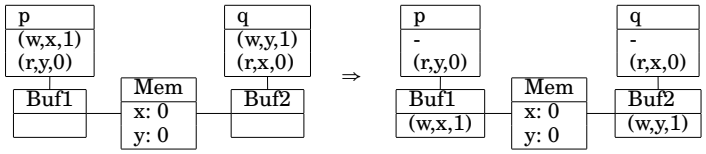

Fig. 2. Illustration of weakly consistent behavior under PSO and TSO.

In general, weak memory models define serializations of the memory operations that are impossible on a sequentially consistent (SC) memory [Lamport 1979]. Under SC, operations are immediately seen by all processes or, phrased differently, the sequences of memory operations are interleaved. SC matches the developer's intuition about program behavior. As a result, 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.

To illustrate the problems caused by weak consistency, Figure 1 shows a simplified version of Dekker's mutual exclusion algorithm (it does not use a token variable). Both processes claim a resource by setting their variable to 1. If the resource is claimed by the partner, a process releases the resource and waits for some time until it claims the resource again. If the resource is not claimed by the partner, the process enters its critical section and releases the resource afterwards.

Intuitively, the protocol guarantees mutual exclusion. However, if the algorithm is executed on an architecture where the processes buffer their writes, like TSO or PSO, it may behave incorrectly (Figure 2). Both p and q issue their write operations and put them into their buffers. No write operation was passed to the main memory yet. Since each process can only access its own write buffer, both will still see the initial value of the main memory for the variable of the partner process. This means both read 0 and enter the critical section. To detect bugs like this, considerable effort has been made to develop verification methods for concurrent programs that run under weak memory models (see the discussion below).

1.2. The Testing Problem

A core problem behind such 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 [Gibbons and Korach 1997], is defined as follows. Given a sequence of read/write operations for each concurrent process, check whether there is an interleaving of the operations that satisfies the constraints of the weak memory model at hand. 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.

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 leads to a counterexample to a correctness statement. To check whether the counterexample corresponds to an actual execution, we extract for each process the sequence of operations 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 [Clarke et al. 2000]. As an under-approximation, testing is used during debugging. We check reachability of an undesirable state for each process and then solve the testing problem on the collected sequences of operations. A further application are synchronization inference algorithms [Abdulla et al. 2012; Alglave and Maranget 2011; Bouajjani et al. 2013; Liu et al. 2012; Alglave et al. 2014]. 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.

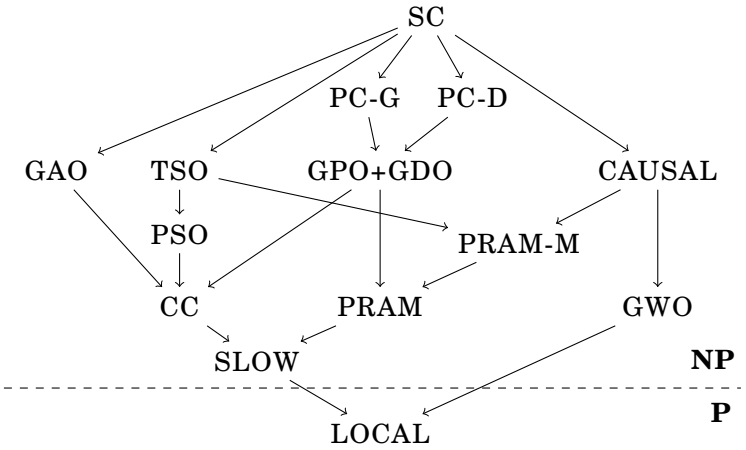

Fig. 3. Hierarchy of weak memory models [Steinke and Nutt 2004].

Despite its many applications, there are only a few works on the algorithmics and complexity of the testing problem. Gibbons and Korach [Gibbons and Korach 1997] 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 [Cantin et al. 2005] extended these results. They state **NP**-completeness of the testing problem under SPARC's memory models (TSO, PSO, RMO) [WeGe94 1994], processor consistency PC, release consistency, and a model of the PowerPC architecture. Conflict serializability was studied in [Farzan and Madhusudan 2009], with and without synchronization. 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 therefore introduce general techniques that address the testing problem under different memory models in a *uniform* way.

To develop a uniform approach to memory-model-aware testing, it is important to classify the weak memory models as done in [Alglave 2012; Mosberger 1993; Steinke and Nutt 2004]. A memory model M_w is called *weaker than* another memory model M_s,

denoted by $M_s \preceq M_w$ and indicated by a path in Figure 3, if every execution allowed under M_s is also valid under M_w. Memory models are usually defined by axioms [Loewenstein et al. 2006], in an operational way, or via local views. Steinke and Nutt [Steinke and Nutt 2004] have shown that most 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) [Lipton and Sandberg 1988], CAUSAL consistency [Hutto and Ahamad 1990], cache consistency (CC) [Goodman 1991], two variants of processor consistency (PC-G, PC-D) [Goodman 1991; Ahamad et al. 1993], SLOW consistency [Hutto and Ahamad 1990], and LOCAL consistency [Heddaya and Sinha 1992]. As a consequence of this characterization via basic models, the memory models form the hierarchy depicted in Figure 3. The hierarchy shows that SC is the strongest and LOCAL consistency is the weakest model.

1.3. Contributions

We present *algorithms and complexity results for the testing problem under the weak memory models in the Steinke-Nutt hierarchy*. As shown in Figure 3, 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*, which we consider the actual main contributions of this paper. For new memory models that are likely to come up, our general concepts will make it easy to devise optimal testing algorithms.

Contribution 1: Most testing problems are NP-hard. 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* M *with* $\mathrm{M}_S \preceq \mathrm{M} \preceq \mathrm{M}_W$. 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, 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 give additional results for fixed parameter testing problems.

Contribution 2: Some testing problems are in P. 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 a common 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*. Processes are given as sequences of operations. 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 powerset construction for finite automata.

Contribution 3: The remaining testing problems are in NP. We show that the testing problem is in **NP** for all models in the Steinke-Nutt hierarchy of Figure 3. In the framework of Steinke and Nutt, the testing problem under a weak memory model

is defined as the ability to serialize certain partial orders. We define a SAT encoding that takes a partial order as a parameter. It computes a propositional formula that is satisfiable if and only if the partial order admits a serialization. Contribution 1 shows 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 [Steinke and Nutt 2004]. Phrased differently, the real contribution is to observe that the Steinke-Nutt framework is well-suited for SAT.

Contribution 4: Influence of parameters. 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 processes. Second, protocols often consist of short processes which means the length of the read/write sequences is limited. Finally, the number of synchronization variables is usually small. For these reasons, we consider variants of the testing problem where one of the three parameters is bounded for all inputs. 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 [Kozen 1977] to non-primitive recursive for TSO, PSO, and an approximation of POWER [Atig et al. 2010; Atig et al. 2012]. 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 [Alglave 2013]. 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**.

This paper is an extended version of [Furbach et al. 2014]. It differs from the preliminary paper in the following aspects. We have improved the SC $\leq$ GWO-range reduction so as to use a fixed number of variables. We additionally list the SC $\leq$ PSO-range reduction that shows **NP**-hardness from SC to PSO if the number of processes is fixed. Finally, we add the algorithm that solves the testing problem under CC. We prove its correctness and give a sketch of completeness.

The paper is organized as follows. After a discussion of related work, we define the formal set-up in Section 3. We present **NP**-hard testing problems and their reductions in Section 4. Then we develop polynomial-time testing algorithms in Section 5. We show that all considered testing problems are in **NP** in Section 6. We summarize the results and conclude the paper in Section 7.

2. RELATED WORK

We already discussed the related work on the testing problem, but would like to add a remark on [Cantin et al. 2005]. 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 [Gibbons and Korach 1997]). This argument does not apply if programs come free from synchronization primitives. This is the case in the present paper, 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 [Atig et al. 2010] and for an approximation of POWER [Atig et al. 2012]. Robustness requires the absence of causality cycles, and has been shown to be decidable in polynomial space for TSO [Bouajjani et al. 2011], partitioned global address spaces [Calin et al. 2013], and for POWER [Derevenetc and Meyer 2014]. The testing problem has a lower complexity as it handles single sequences of operations rather than sets. Our multi-parameter complexity analysis is related to [Esparza and Ganty 2011]. 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). Runtime verification techniques for TSO and PSO have been developed in [Burckhardt and Musuvathi 2008; Burnim et al. 2011]. Atig et al. extended the idea of bounded context switching to TSO [Atig et al. 2011]. Recently, Alglave et al. developed memory-model-aware BMC algorithms [Alglave et al. 2013]. The approach is remarkable in that it applies to various models, and indeed inspired our SAT encoding given in Section 6. It does, however, not lead to complexity results. Instead, the focus was on practical verification algorithms for popular architectures, including Intel's x86, and IBM Power. Vechev et al. developed over-approximate verification techniques that prove programs correct [Atig et al. 2010].

Finally, we discuss our choice to base this work on the Steinke-Nutt hierarchy rather than the recent framework of Alglave [Alglave 2010]. The reason is that the Steinke-Nutt hierarchy covers more models, which led to the idea of range reductions. Moreover, the view-based formulation is close to formal languages so that we were able to extract polynomial-time algorithms from it.

3. TESTS AND MEMORY MODELS

We first give the definition of tests and memory models following [Steinke and Nutt 2004]. Then, we turn to the testing problem. For illustration purposes, we consider the mutex algorithm given in Figure 1. We study a test for this program which shows that the mutex guarantee depends on the memory model.

3.1. Syntax of Tests

A test consists of a finite set of processes that operate on a shared memory. Each process is given as a sequence of read/write operations that, intuitively, correspond to the local view of the process on the shared memory. In this view, both the memory location and the value of a read/write operation are fully determined. In particular does a read operation already provide the value that is read. A test for the example in Figure 1 is

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

To explain the test, we first define the operations involved. Formally, an *operation* op is an element in $\mathcal{OP} := (\mathcal{C} \times \mathcal{V} \times \mathcal{D}) \times (\mathcal{ID} \times \mathcal{N})$. The operation executes a write or read command from $\mathcal{C} := \{w, r\}$ on a variable from the set $\mathcal{V}$, assuming a fixed value from some data domain $\mathcal{D}$ that contains $\bot$. Each operation carries a process identifier from the set $\mathcal{ID}$ which has a distinguished element ε for the initialization process. Moreover, an operation has an issue index, a natural number in $\mathcal{N} := \{0, 1, \ldots\}$ that determines the order in which operations are issued by a process. Given an operation $op = (c, x, v, p, i) \in \mathcal{OP}$, we use $cmd(op) = c$, $var(op) = x$, $val(op) = v$, $proc(op) = p$, and

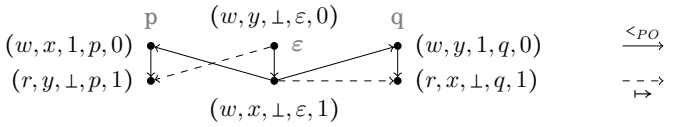

Fig. 4. The execution and program order of test $\mathcal{T}_{Dekker}$.

$idx(op) = i$ to access the command, the variable, the value, the process identifier, and the issue index. Given $\mathcal{T} \subseteq \mathcal{OP}$, we denote a subset of operations that share certain properties using wildcard $*$. For example, the set of operations writing variable x, $\{op \in \mathcal{T} \mid cmd(op) = w \text{ and } var(op) = x\}$, is denoted by $(w, x, *, *, *)_{\mathcal{T}}$. By slight abuse of notation, we use w to denote a write operation and similarly r for a read operation. To establish the initial value $\bot$ for all variables, we introduce write operations $(w, x, \bot, \varepsilon, i)$ that belong to the initialization process ε.

Definition 3.1. A *test* is a finite subset of operations $\mathcal{T} \subseteq \mathcal{OP}$ so that $|(*, *, *, p, i)_{\mathcal{T}}| \leq 1$ for all $p \in \mathcal{ID}$ and $i \in \mathcal{N}$. Moreover, if $|(*, x, *, *, *)_{\mathcal{T}}| > 0$ then $|(w, x, \bot, \varepsilon, *)_{\mathcal{T}}| = 1$.

We now formally define $\mathcal{T}_{Dekker}$, the test for the program in Figure 1 given above:

$$(w, x, 1, p, 0).(r, y, \bot, p, 1) \parallel (w, y, 1, q, 0).(r, x, \bot, q, 1)$$
$$\parallel (w, y, \bot, \varepsilon, 0).(w, x, \bot, \varepsilon, 1).$$

The test consists of two processes with identifiers p and q, and the initial process 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 processes p and q write 1 to their variable but read the initial value 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 processes, and processes as sequences of operations. In this case, we may omit both the process identifier and the issue index. We further assume that the initial process ε only writes $\bot$ to each used variable. If we fix an ordering on the variables, this fully defines the initial process and we can omit it as well. With these conventions, we arrive at the previous description of $\mathcal{T}_{Dekker} : (w, x, 1).(r, y, \bot) \parallel (w, y, 1).(r, x, \bot)$.

3.2. Memory-Model-Aware Semantics of Tests

The semantics of tests is defined in terms of executions, relations on the operations determining the write that a read receives its value from.

Definition 3.2. An *execution of* $\mathcal{T} \subseteq \mathcal{OP}$ is a relation $\mapsto \subseteq (w, *, *, *, *)_{\mathcal{T}} \times (r, *, *, *, *)_{\mathcal{T}}$ so that for every read $r \in \mathcal{T}$ there is precisely one write $w \in \mathcal{T}$ with $w \mapsto r$. Moreover, $w \mapsto r$ implies $var(w) = var(r)$ and $val(w) = val(r)$.

Memory consistency models restrict the set of executions to so-called valid ones. In the Steinke-Nutt framework, valid executions are defined in terms of serial views. Roughly, a serial view of an execution is the total order in which the operations become visible to a process. 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 process may, however, not see all the operations of other processes. To model this, the definition of serial views takes a subset of operations as a parameter. Given an execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$, we call a subset $\mathcal{O} \subseteq \mathcal{T}$ *source-closed* if for all $r \in \mathcal{O}$ and $w \in \mathcal{T}$ with $w \mapsto r$, we have $w \in \mathcal{O}$.

Definition 3.3. Consider an execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$, a source-closed set $\mathcal{O} \subseteq \mathcal{T}$, and a strict partial order $< \subseteq \mathcal{O} \times \mathcal{O}$. A strict total order $<_{sv} \subseteq \mathcal{O} \times \mathcal{O}$ is a *serial view of $\mathcal{O}$ in $\mapsto$ that respects* $<$ if it satisfies the following:

(i) It refines $<$, which means $< \subseteq <_{sv}$.
(ii) For all pairs $w \mapsto r$ with $w, r \in \mathcal{O}$ we have $w <_{sv} r$. Moreover, there is no $w' \in \mathcal{O}$ so that $w <_{sv} w' <_{sv} r$ and $var(w) = var(w')$.

We also write $<_{sv}$ *is* $SerialView(\mapsto, \mathcal{O}, <)$.

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) [Lamport 1979]. SC ensures that every process observes all operations in the order in which they were issued. It is the classic memory model which is standard in textbooks and intuitively assumed by most programmers. Using Definition 3.3, an execution is valid under SC if there is a serial view of all operations that respects the program order. The program order is thereby defined as the order of operations within the same process. Let $p \in \mathcal{ID}$. The *process order* $<_p \subseteq \mathcal{T} \times \mathcal{T}$ orders the commands of p according to their issue index, and after all initial writes:

$$op_1 <_p op_2 \quad \text{if} \quad [proc(op_1) = proc(op_2) = p \text{ and } idx(op_1) < idx(op_2)] \\ \text{or } [proc(op_1) = \varepsilon \text{ and } proc(op_2) = p].$$

The *program order* $<_{PO} \subseteq \mathcal{T} \times \mathcal{T}$ is the union of all process orders, $<_{PO} := \bigcup_{p \in \mathcal{ID}} <_p$.

Definition 3.4. An execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ is *valid under sequential consistency* if

$$\exists <_{sv} : <_{sv} \text{ is } SerialView(\mapsto, \mathcal{T}, <_{PO}).$$

The example test $\mathcal{T}_{Dekker}$ admits only one execution, depicted by dashed arcs in Figure 4, where the reads observe the writes of the initial process ε. In any serial view, 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 3.3 and 3.4. Since the serial view respects the program order $<_{PO}$, the maximal operation 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 operation:

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

Since $(w, y, \bot, \varepsilon, 0) \mapsto (r, y, \bot, p, 1)$, we obtain a contradiction to Definition 3.3(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. This behavior changes if we examine the test under a weaker model.

Hutto and Ahamad [Hutto and Ahamad 1990] developed the SLOW model to solve the exclusion and dictionary problems with minimal consistency maintenance. The model requires that the processes 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 by a serial view for every process and every variable. The view contains all write operations on this variable and all operations of this process on the variable, and respects the program order. We have [Steinke and Nutt 2004, Theorem 3.7]:

Definition 3.5. An execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ is *valid under* SLOW *consistency* if

$$\forall p \in \mathcal{ID}\ \forall x \in \mathcal{V}\ \exists <_{sv} : \\ <_{sv} \text{ is } SerialView(\mapsto, (*, x, *, p, *)_{\mathcal{T}} \cup (w, x, *, *, *)_{\mathcal{T}}, <_{PO}).$$

Since writes on different variables can be observed out of order, the execution in Figure 4 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:

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

3.3. Memory-Model-Aware Testing

We consider the testing problem TEST(M) for every memory model M shown in Figure 3. The definition is as follows.

> **Problem:** Given a test $\mathcal{T} \subseteq \mathcal{OP}$, is there an execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ that is valid under M?

We say a test *succeeds under* M if there is a valid execution, otherwise it *fails under* M. In the example, $\mathcal{T}_{Dekker}$ fails under SC but succeeds under SLOW.

We also consider restricted variants of the testing problem that admit more efficient algorithms: $\text{TEST}_P(\text{M})$ assumes a fixed number of processes in input tests, $\text{TEST}_L(\text{M})$ fixes the length of processes (number of operations), and $\text{TEST}_V(\text{M})$ studies the problem for a fixed number of variables.

4. MOST TESTING PROBLEMS ARE NP-HARD

We derive basic hardness results that guide our search for testing algorithms in the next sections. Interestingly, the main finding 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.

4.1. 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, TEST(M) contains all tests $\mathcal{T}$ and the predicate asks for an execution of $\mathcal{T}$ that is valid under M. A reduction $f : \text{PROB} \to \text{TEST(M)}$ is a function that maps instances of PROB to tests so that

$$\psi(x) \text{ holds} \quad \text{iff} \quad \text{test } f(x) = \mathcal{T} \text{ succeeds under M.} \tag{1}$$

Our goal is to conclude such an equivalence not only for a single memory model, but for a range of models M that are weaker than a given model M_S and stronger than another model M_W. To this end, we reformulate Equivalence (1) in such a way that it comprises all models $\text{M}_S \preceq \text{M} \preceq \text{M}_W$.

Definition 4.1. A function f from instances of PROB to tests is an $\text{M}_S \preceq \text{M}_W$*-range reduction of* PROB *to the testing problem* if the following implications hold.

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

Figure 5 illustrates the definition. If function f is polynomial-time computable and PROB is **NP**-hard, we derive **NP**-hardness of the testing problem.

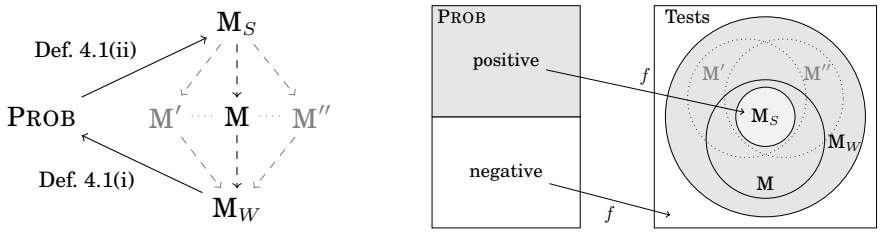

Fig. 5. Illustration of an $M_S \preceq M_W$-range reduction of PROB to the testing problem. The left-hand side shows the implications in Def. 4.1 as solid directed edges. The dashed edges represent the weaker-than relation among memory models. The Venn diagram on the right-hand side illustrates the validity of tests obtained from a range reduction. Positve instances of PROB are mapped to tests that succeed under M_S. Negative instances of PROB are mapped to tests that fail under M_W.

LEMMA 4.2. *Let* PROB *be* **NP***-hard and let* f *be a polynomial-time computable* $M_S \preceq M_W$*-range reduction of* PROB *to the testing problem. Then* TEST(M) *is* **NP***-hard for all memory models* $M_S \preceq M \preceq M_W$.

PROOF. We show Equivalence (1) for any M with $M_S \preceq M \preceq M_W$. Assume $\psi(x)$ holds. With Definition 4.1(ii), test $f(x) = \mathcal{T}$ succeeds under M_S. Since $M_S \preceq M$, the test remains successful under M. For the reverse direction, assume $\mathcal{T}$ succeeds under M. Then the test remains successful under M_W. With Definition 4.1(i), $\psi(x)$ holds. Since f is polynomial-time computable, **NP**-hardness follows. □

In the remainder of the section, we show that almost all testing problems are **NP**-hard. Using Lemma 4.2, 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 5, 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 4.1(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 operations over multiple processes. 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.

4.2. Fixed Variable Testing is NP-hard from SC to SLOW

We present an SC $\preceq$ SLOW-range reduction of SAT to the testing problem. Consider a SAT instance of the form $\varphi = \bigwedge_{i \in I} cl_i$ with $cl_i = \bigvee_{j \in J} lit_{i,j}$. We translate it 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 operations on the same variable, and then introduce only one variable ξ used by the processes in the reduction. The data domain $\mathcal{D}$ of ξ is defined as follows. For each variable x in the SAT instance φ there is a corresponding value $x \in \mathcal{D}$, and for each clause cl of the SAT instance there is a value $cl \in \mathcal{D}$. The test consists of a process t, and for each variable x in the propositional formula φ, there are two further processes

p_x and n_x:

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

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 processes for each variable x in φ. The first process p_x writes to ξ, one by one, the clauses cl that contain x positively. Afterwards, the process writes x to ξ to indicate that the variable has been handled. The second process n_x is similar, but writes the clauses that contain x negatively. The intuition is as follows. If process p_x is executed, variable x is set to true, and consequently all clauses that contain x positively will be satisfied. Likewise, if process n_x is executed, then variable x is made false, and all clauses that have negative occurrences of x hold:

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

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

The processes p_x and n_x are augmented by a test process t. For its definition, we again use the iterated concatenation and assume the same order of clauses as above:

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

Process 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 processes has terminated when process t proceeds with reading clause values from its variable ξ. Say n_x has terminated. The remaining process p_x effectively assigns true to variable x. When process t continues to read all clauses in its second part, it thereby checks that these clauses are satisfied by the corresponding assignment.

THEOREM 4.3. *The above function f is an* SC $\preceq$ SLOW*-range reduction of* SAT *to the testing problem that is polynomial-time computable. Hence,* TEST(M) *is* **NP***-hard for all memory models* SC $\preceq$ M $\preceq$ SLOW. *As the reduction only uses one variable, even* TEST_V(M) *is* **NP***-hard for all memory models* SC $\preceq$ M $\preceq$ SLOW.

PROOF. 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). The full process 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, process 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 valid execution means we satisfy all clauses, we obtain a satisfying assignment for φ. □

The theorem shows that testing is **NP**-hard if the operations of one process to the same variable cannot be reordered — which is the case in most memory models. The reduction relies, however, on an unbounded number of processes. As we will show, TEST(SLOW) becomes polynomial if we fix this parameter.

4.3. Fixed Length and Fixed Variable Testing is NP-hard 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 [Steinke and Nutt 2004]). It requires the following consistency: if a process observes an order between two writes (because it reads from the first write and later performs the second), then all processes agree on this order. Formally, we introduce the *write-read-write order* (WO):

$$w_1 <_{WO} w_2 \text{ if } \exists r \in \mathcal{T} : \; w_1 \mapsto r \; \wedge \; r <_{PO} w_2.$$

Definition 4.4. An execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ is *valid under* GWO *consistency* if

$$\forall p \in \mathcal{ID} \; \exists <_{sv} : <_{sv} \text{ is } \mathit{SerialView}\left(\mapsto, (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}, <_p \cup <_{WO}\right).$$

We construct an SC $\leq$ GWO-range reduction of SAT to the testing problem. The idea is to add reads that enforce a global order $<_{WO}$ among critical write operations. Let $\varphi = \bigwedge_{1\leq k\leq K} cl_k$ with $cl_k = \bigvee_{1\leq j\leq J_k} lit_{k,j}$, and literals $lit_{k,j}$ be either positive or negative instances of variables x_i with $0 \leq i < N$. 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 x for the assignment, a variable c for checking clauses, and auxiliary variables h and y. The values of the variables x and y can be pairs. This allows us to only use a fixed number of variables. For each literal $lit_{k,j}$, let $v(k,j)$ determine its variable. Moreover, we set $b(k,j) := 1$ for a positive literal and $b(k,j) := 0$ for a negative literal. The encoding is

$$\mathcal{T} := \prod_{1\leq i\leq N} (n_i \parallel p_i \parallel q_i \parallel r_i) \parallel \prod_{\substack{1\leq k\leq K \\ 1\leq j\leq J_k}} l_{k,j} \parallel h \parallel \prod_{1\leq k\leq K} t_k.$$

Test $\mathcal{T}$ defines four processes per variable x_i in φ. The first two processes n_i and p_i write to x, respectively the value $(i,0)$ or $(i,1)$, read that value again, and then write $(i,1)$ to y. The third process q_i reads value $(i,1)$ from y and writes value $(i,2)$ to x:

$$\begin{aligned} n_i &:= (w,x,\tbinom{i}{0}).(r,x,\tbinom{i}{0}).(w,y,\tbinom{i}{1}) \\ p_i &:= (w,x,\tbinom{i}{1}).(r,x,\tbinom{i}{1}).(w,y,\tbinom{i}{1}) \\ q_i &:= (r,y,\tbinom{i}{1}).(w,x,\tbinom{i}{2}). \end{aligned}$$

The idea is that at least one of the writes of $(i,1)$ or $(i,0)$ to x by p_i or n_i is overwritten with value $(i,2)$ by process q_i. Say it is $(i,0)$. Under the write-read-write order, all processes which read $(i,2)$ from x can only read the remaining write of $(i,1)$ by p_i afterwards. This corresponds to the value 1 being assigned to x.

There is a process $l_{k,j}$ for each literal $lit_{k,j}$. It checks whether the literal is satisfied by the assignment. The process 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_i then it reads $(i,0)$ from x, otherwise it reads $(i,1)$ from x. After the reads, the process writes the literal's clause k to variable c:

$$l_{k,j} := (r,x,\tbinom{v(k,j)}{2}).(r,x,\tbinom{v(k,j)}{b(k,j)}).(w,c,k).$$

For each clause cl_k there is a test process t_k. It ensures that all clauses up to the current one were satisfied by at least one literal. The test process reads the value $k-1$ from variable h of the previous test process, tries to read the clause id k from variable

c, and then writes value k to auxiliary variable h:

$$t_k \;:=\; (r,h,k-1).(r,c,k).(w,h,k).$$

Process $h := (w,h,0)$ just writes the first value to the auxiliary variable h. Process r_i ensures that after all test processes t_i have finished the remaining literal processes $l_{k,j}$ may finish too. It first checks whether the last auxiliary value K was written to h, and then writes $(i,0)$ and $(i,1)$ to x:

$$r_i \;:=\; (r,h,K).(w,x,\binom{i}{0}).(w,x,\binom{i}{1}).$$

We claim that test $\mathcal{T}$ is successful under SC if φ is satisfiable. Consider a satisfying assignment Φ for φ. Starting with $i = 1$, we execute p_i if $\Phi(x_i) = 0$ and n_i if $\Phi(x_i) = 1$. Then, we execute q_i followed by the first read of all literal processes $l_{k,j}$ with this variable: $v(k,j) = x_i$. Afterwards, we execute n_i or p_i, whichever remains. Finally, all literal processes $l_{k,j}$ that correspond to satisfied literals may read the correct value and then halt before their write operation. We repeat this for all variables and proceed with executing process h. Now for each clause cl_k the last operation of the literal processes that correspond to satisfied literals of this clause may execute, followed by processes t_k. 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_k there is a literal process that wrote k to c. To conclude, we only have to guarantee that the remaining literal processes terminate. For each variable x_i we execute r_i up to its first write, followed by all remaining literal processes which correspond to negative literals of x_i. Then r_i executes its last write such that the remaining literal processes which correspond to positive literals of x_i 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 process issued a write after reading another write, then these two writes are ordered for all processes. In an execution, process t_k reads value k from the clause variable c. This clause variable is written by some process $l_{k,j}$ corresponding to a satisfied literal. These literal processes determine the assignment: $\Phi(v(k,j)) := b(k,j)$. To see that Φ is well-defined, consider two literal processes with the same variable x_i that are both read by test processes. Both literal processes receive their value from the same p_i or n_i, as the other process n_i or p_i occurs before $(w,x,(i,2))$.

Note that the length of processes is at most three in $\mathcal{T}$ and that the number of variables is only four. Therefore, the reduction shows **NP**-hardness of the corresponding restricted versions of the testing problem.

THEOREM 4.5. *Function f is an* SC $\leq$ GWO*-range reduction of* SAT *to the testing problem that is polynomial-time computable. Hence,* TEST(M) *is* **NP***-hard for all memory models* SC $\leq$ M $\leq$ GWO*. As the reduction requires a process length of at most three, even* TEST_L(M) *is* **NP***-hard for all memory models* SC $\leq$ M $\leq$ GWO*. As the reduction requires only four variables, even* TEST_V(M) *is* **NP***-hard for all memory models* SC $\leq$ M $\leq$ GWO.

4.4. Fixed Length and Fixed Variable Testing is NP-hard from SC to CC

We now introduce the cache consistency model as defined by [Steinke and Nutt 2004]. Here only the operations on the same variable occur in program order:

Definition 4.6. An execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ is *valid under* CC *consistency* if

$$\forall x \in \mathcal{V}\ \exists <_{sv} : <_{sv} \text{ is } \mathit{SerialView}\left(\mapsto, (*,x,*,*,*)_{\mathcal{T}}, <_{PO}\right).$$

We describe an SC $\preceq$ CC-range reduction of SAT to the testing problem that fixes both, the length of processes and the number of variables. Consider a SAT instance of the form $\varphi = \bigwedge_{1\le i\le I} cl_i$ with $cl_i = \bigvee_{1\le j\le J_i} lit_{i,j}$. 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 the SAT instance φ there are two values $(x,0),(x,1) \in \mathcal{D}$, and for each clause cl_i of the SAT instance there is a value $cl_i \in \mathcal{D}$. Furthermore, there are auxiliary values $h_0 \dots h_I$ in the data domain. For a positive literal $lit_{i,j} = x$, let $n(i,j) := (x,0)$ and $p(i,j) := (x,1)$. For a negative literal $lit_{i,j} = \neg x$, define $n(i,j)$ and $p(i,j)$ vice versa. The test is:

$$\mathcal{T} \quad := \quad \prod_{x\in\varphi}(p_x \parallel n_x) \parallel \prod_{\substack{1\le i\le I\\ 1\le j\le J_i}} k_{i,j} \parallel h \parallel \prod_{1\le i\le I} t_i \parallel \prod_{x\in\varphi} r_x.$$

Program $\mathcal{T}$ defines three processes for each variable x in φ. The first two processes 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:

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

There is a process $k_{i,j}$ in $\mathcal{T}$ for each literal $lit_{i,j}$ which tests whether the literal 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_i signalling that the literal's clause is satisfied:

$$k_{i,j} := (r,\xi,n(i,j)).(r,\xi,p(i,j)).(w,\xi,cl_i).$$

The auxiliary process h just writes the first of the auxiliary values:

$$h := (w,\xi,h_0).$$

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

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

The processes r_x check for each variable x in φ, whether the last test process t_I was successful by reading h_I and then write $(x,0)$, $(x,1)$ and again $(x,0)$. This allows all processes corresponding to unsatisfied literals to finish by reading their expected order of values:

$$r_x := (r,\xi,h_I).(w,\xi,\binom{x}{0}).(w,\xi,\binom{x}{1}).(w,\xi,\binom{x}{0}).$$

THEOREM 4.7. *The above function f is an* SC $\preceq$ CC*-range reduction of* SAT *to testing that is polynomial-time computable. Hence,* TEST(M) *is* **NP***-hard for all memory models* SC $\preceq$ M $\preceq$ CC. *As the reduction only uses one variable and is fixed in the length of processes, even* TEST_V(M) *and* TEST_L(M) *are* **NP***-hard for all memory models* SC $\preceq$ M $\preceq$ CC.

We claim that the test program $\mathcal{T}$ indeed succeeds under SC if φ is satisfiable. Given a satisfying assignment Φ, assume variable x is true in this assignment. Process n_x is executed first, followed by the first reads of all processes $k_{i,j}$ that correspond to a positive literal of variable x. Afterwards p_x is executed, again followed by the second reads of the former $k_{i,j}$. This is done successively for each variable. Then process h executes. For each clause cl_i, a test process t_i reads the prior auxiliary value h_i. Then the processes $k_{i,j}$ that already executed their two reads execute their write of cl_i. This way t_i can read the value and continue writing its auxiliary value and thus enabling the next test process. As Φ is a satisfying assignment, there must exist at least one process for each clause cl_i that can write the value cl_i. This means each test process t_i can read its cl_i. To finish the program, we must allow the processes corresponding to unsatisfied literals to complete. All processes r_x execute their first read. For each variable x, we execute the write of $(x, 0)$ from process r_x. The write enables the first reads of the remaining processes that correspond to positive literals of x. Then r_x writes $(x, 1)$. The write enables the first reads of the remaining processes 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 processes corresponding to negative literals of x can execute their second read. Finally, all literal processes $k_{i,j}$ that have not yet executed their write can do so.

To see that a CC execution of $\mathcal{T}$ gives a satisfying assignment to φ, note that the program only has one variable and thus consistency under CC and SC coincide. All operations 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_i of φ the test process t_i reads cl_i from one of the writes of a literal process $k_{i,j}$. As explained before, this corresponds to the literal being satisfied by the modeled assignment. Assuming $lit_{i,j}$ 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 theorem shows that testing is still **NP**-hard if operations to the same variable cannot be reordered and the length of processes is fixed. The range reduction relies, however, still on an unbounded number of processes. We will show that the problem is polynomial for a fixed number of processes.

4.5. Fixed Process Testing is NP-hard from SC to PSO

PSO consistency is best explained by describing a possible computer architecture as defined in [Atig et al. 2010]. Each process has a set of FIFO buffers, one for each variable. These buffers hold the writes to that variable that have been executed by the process but have not yet been commited to memory. If a process reads from a variable, 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 processes. 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 [Weaver and Germond 1994] using axioms. The definition describes how operations are observed from the viewpoint of the shared memory. This should be contrasted with the view-based approach focussing on the observations made by the processes. Unfortunately, no view-based definition for PSO has been introduced at this time. Therefore, the following reduction relies on the operational model sketched above [Atig et al. 2010].

We give an SC $\preceq$ PSO-range reduction of 3SAT to testing. Consider a 3SAT instance on variable set X of the form $\varphi = \bigwedge_{i \in I} cl_i$ with $cl_i = a_i \vee b_i \vee c_i$. We translate it to a test

$f(\varphi) = \mathcal{T}$ that satisfies the following. If formula φ is satisfiable then $\mathcal{T}$ is valid under SC. Moreover, if the test is valid under PSO, then the formula is satisfiable.

Each variable x in the 3SAT instance φ is also a variable $x \in \mathcal{V}$. For each literal a of the 3SAT instance, let $var(a)$ be the variable occurring in a and let $b(a)$ be 0 if a is negated and 1 otherwise. For every clause i in the formula and every process p that we define in the test, we introduce a synchronization variable $z_{i,p}$. Moreover, every clause 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 processes p_a and p'_a, where p_a writes the value 1 into every variable and p'_a writes 0. The processes 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.

The test then processes each clause $cl_i = a_i \vee b_i \vee c_i$. We introduce $Sync$ sequences to ensure that all processes of $\mathcal{T}_p$ handle the same clause. For each clause literal $l_i \in \{a_i, b_i, c_i\}$, the processes 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 $Flip$:

$$\begin{aligned}
p_a &:= [\bullet_{x\in\mathcal{V}}(w,x,0)].\ [\bullet_{i\in I} Sync_i(p_a).\ (r, var(a_i), b(a_i)).\ Flip(b_i,0)]\\
p'_a &:= [\bullet_{x\in\mathcal{V}}(w,x,1)].\ [\bullet_{i\in I} Sync_i(p'_a).\ (r, var(a_i), b(a_i)).\ Flip(b_i,1)]\\
p_b &:= [\bullet_{x\in\mathcal{V}}(r,x,0)].\ [\bullet_{i\in I} Sync_i(p_b).\ (r, var(b_i), b(b_i)).\ Flip(c_i,0)]\\
p'_b &:= [\bullet_{x\in\mathcal{V}}(r,x,1)].\ [\bullet_{i\in I} Sync_i(p'_b).\ (r, var(b_i), b(b_i)).\ Flip(c_i,1)]\\
p_c &:= [\bullet_{i\in I} Sync_i(p_c).\ (r, var(c_i), b(c_i)).\ Flip(a_i,0)]\\
p'_c &:= [\bullet_{i\in I} Sync_i(p'_c).\ (r, var(c_i), b(c_i)).\ Flip(a_i,1)]\ .
\end{aligned}$$

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

$$\forall l \in \{a,b,c\} : q_l = [\bullet_{i\in I} SyncF(l_i, 0)]$$
$$q'_l = [\bullet_{i\in I} SyncF(l_i, 1)]$$

The $Sync_i(*)$ sequences are constructed such that they can only be processed when all processes in $\mathcal{T}_p$ have reached their $Sync_i(*)$ 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 $Sync$ processes must also have been processed. For a given clause cl_i and process p, we define

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

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

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

The purpose of the $SyncF$ sequence is to make sure that the inverting write has left the buffer before $Flip(l, v)$ 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:

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

THEOREM 4.8. *The above function f is an* SC $\preceq$ PSO*-range reduction of* 3SAT *to testing with a fixed number of processes that is polynomial-time computable. Hence,* $\text{TEST}_P(\text{M})$ *is* **NP***-hard for all memory models* SC $\preceq$ M $\preceq$ PSO.

PROOF. We show the following. 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 processes 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 processes 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 process 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 $\preceq$ PSO-range reduction of 3SAT to testing with a fixed number of processes. □

5. SOME TESTING PROBLEMS ARE 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 $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ 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 3.3(ii), a serial view $<_{sv}$ has to respect a given execution $\mapsto$. This means for every read r occurring in $<_{sv}$ the serial view implicitly gives the write w with $w \mapsto r$: 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.

5.1. Read Partitioning and Constructive Serial Views

The catch in the argumentation is that serial views are defined for subsets of operations $\mathcal{O} \subseteq \mathcal{T}$. 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 $<_{sv}^1$ for requirement $SerialView\,(\mapsto, \mathcal{O}_1, <_1)$ to $<_{sv}^k$ for $SerialView\,(\mapsto, \mathcal{O}_k, <_k)$. The problem is that the partial executions for $\mathcal{O}_1$ to $\mathcal{O}_k$ may be incompatible. Serial view $<_{sv}^1$ may give $w_1 \mapsto r$ while $<_{sv}^2$ yields $w_2 \mapsto r$ 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 operations $\mathcal{O}_j \subseteq \mathcal{T}$ so that $r \in \mathcal{O}_j$. SC, SLOW, and the LOCAL model defined 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 3.3. Consider $\mathcal{O} \subseteq \mathcal{T}$ and a strict partial order $< \subseteq \mathcal{O} \times \mathcal{O}$. A *constructive serial view of $\mathcal{O}$ which respects* $<$ is a strict total order $<_{csv} \subseteq \mathcal{O} \times \mathcal{O}$ that is defined like a serial view but replaces Definition 3.3(ii) by

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

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 M is defined by the serial views $SerialView\,(\mapsto, \mathcal{O}_1, <_1)$ to $SerialView\,(\mapsto, \mathcal{O}_k, <_k)$.

LEMMA 5.1. *Let* M *be read-partitioning and consider a test $\mathcal{T}$. Then $\mathcal{T}$ succeeds under* M *if and only if there are constructive serial views $<_{csv}^i$ 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 $w \mapsto r$ 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.

5.2. TEST(LOCAL) is in P

LOCAL consistency was defined as the weakest constraint that every shared memory system should satisfy [Heddaya and Sinha 1992]. It requires that every process observes all visible operations (all writes and its own reads). Moreover, each process sees its own operations in process order but may see the writes of other processes in an arbitrary order. The Steinke-Nutt formulation is as follows [Steinke and Nutt 2004, Theorem 3.8]:

Definition 5.2. An execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ is *valid under* LOCAL *consistency* if

$$\forall p \in \mathcal{ID}\ \exists <_{sv} : <_{sv} \text{ is } SerialView\,(\mapsto, (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}, <_p)\,.$$

The definition introduces a serial view for each process p. The corresponding subset $\mathcal{O} \subseteq \mathcal{T}$ contains all operations of p as well as all writes in the test. The serial view only has to respect the process order of p. This means the operations of p in $\mathcal{O}$ can be understood as a sequence $\tilde{op}_p = op_1 \ldots op_n$. The writes of the other processes are given as an unordered set.

Algorithm 1 copies $\tilde{op}_p$ to the constructive serial view s, inserting writes from other processes where necessary to satisfy reads. To check whether a write is needed to satisfy a read, we hold the last value that has been written to a variable x in $last[x]$. The

ALGORITHM 1: Compute Constructive Serial View for LOCAL

Input: Process p with $\tilde{op}_p = op_1 \dots op_n$ and set of writes W of all processes $q \neq p$
Result: Constructive serial view s, initially empty, $s := \varepsilon$.

```
1  last[x] := ⊥ for all x ∈ V;
2  for i = 1 → n do
3      if op_i = (w, x, v) then
4          last[x] := v; s := s.op_i;
5      else if op_i = (r, x, v) and last[x] = v then
6          s := s.op_i;
7      else if op_i = (r, x, v) and last[x] ≠ v then
8          if ∃w ∈ W : var(w) = x and val(w) = v then
9              W := W ∖ {w};
10             last[x] := v;
11             s := s.w.op_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 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.

THEOREM 5.3. *Algorithm 1 terminates in polynomial time and returns a constructive serial view iff $\mathcal{O}$ and $<_p$ admit one. Hence,* TEST(LOCAL) *is in* **P**.

5.3. TEST$_P$(CC) is in P

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

By Definition 4.6, we need to find a constructive serial view $<_{csv}$ for every $x \in \mathcal{V}$. The corresponding subset of operations $\mathcal{O}_x := (*, x, *, *, *)_{\mathcal{T}}$ contains all operations on x in the test. The serial view has to respect the program order. This means the operations in $\mathcal{O}_x$ are given as sequences $\tilde{op}_p = op_{p,1} \dots op_{p,n_p}$ for every process p. The task is to find an interleaving of these sequences so that every read obtains the desired value. Wlog., we can assume that no read is preceded (in process order) by a write with the same value. Such a read can always follow the write in the serial view and is thus trivial.

We give a non-deterministic algorithm A that solves the problem. In each processing step, A reads segments of operations from two processes and outputs an interleaving (cf. Figure 6). More precisely, A chooses non-deterministically a process p and consumes it up to the next read $r = (r, x, v)$. By our assumption, the last value that p writes to x is different from v. The algorithm non-deterministically chooses a process $q \neq p$ that contains a write $w = (w, x, v)$ which is not preceded by an unconsumed read. It consumes the sequence of writes up to and including the first such write. Now r is enabled and the algorithm consumes it.

If an input sequence starts with a read whose value is that of the last processed write, the read is consumed immediately. To achieve this, the algorithm remembers the last written value in its state. This way, writes are not processed unless necessary.

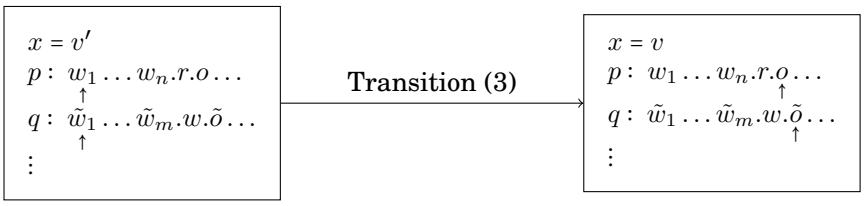

Fig. 6. The algorithm executes the third transition. It chooses the read r on process p and the write w with the same value v on process q. It processes the sequence $w_1 \dots w_n.r$ on p and $\tilde{w}_1 \dots \tilde{w}_m.w$ on q. It returns $w_1 \dots w_n.\tilde{w}_1 \dots \tilde{w}_m.w.r$ and updates the value of x to v.

The algorithm repeats the following:

(1) If the remaining input of a process starts with a read (r, x, v) such that v is the current value of x, then A reads (r, x, v) and also returns it.
(2) If the input sequences contain no more reads, A returns an arbitrary interleaving of the remaining operations. Then the algorithm terminates.
(3) Otherwise, A non-deterministically chooses processes p and q. Let r be the next read of the remaining input of p. The remaining input of q contains a first write w that has the same value v as r and the prefix up to w contains no read. If the remainder of p is $\alpha.r.\beta$ and the remainder of q is $\gamma.w.\delta$ with $\alpha, \gamma \in (w, *, *, *, *)^*$ and $\beta, \delta \in \mathcal{O}^*$, the algorithm returns $\alpha.\gamma.w.r$. The value of the variable is updated to v.

LEMMA 5.4. *For every test $\mathcal{T}$ it holds that A accepts all inputs $\mathcal{O}_x$ restricted to variable x if and only if $\mathcal{T}$ is* CC *consistent.*

PROOF SKETCH. Since for each process the order of operations 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 A accepts all inputs $\mathcal{O}_x$ of a test $\mathcal{T}$, then $\mathcal{T}$ is CC consistent.

It remains to prove completeness. Given some constructive serial view for CC, we can modify it to a sequence of operations 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') or program order. Moreover, writes of the same process are packed together. When a read r receives its value from (w, x, v, q, j) in the constructive serial view, we select the earliest write (w, x, v, q, i) with $i < j$ in q instead, and order the read directly after it. After these modifications we still have a valid serial view. Now the sequence of writes that occurs between two reads r followed by r' can be turned into the following form. Let w be the write accessed by r'. The first writes are those that occur before r' in the program order followed by those that occur before w in the program order and then w. Such a sequence is the result of a processing step of the algorithm. □

To show that the problem is polynomial, it remains to determinize the nondeterministic algorithm. We introduce, for every process p, a pointer referencing the first operation in $\tilde{o}p_p$ that has not yet been processed. There are $|\mathcal{ID}|$ many pointers with at most $|\mathcal{O}|$ positions for each pointer. Hence, there are at most $|\mathcal{O}|^{|\mathcal{ID}|}$ many pointer configurations. The algorithm also stores the last write, which can have at most $|\mathcal{O}|$ many values. This means the algorithm can reach at most $|\mathcal{O}|^{|\mathcal{ID}|+1}$ different 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 processes p and q. Instead, we compute all successor configurations of a given set of configurations. To determine this successor set takes time $|\mathcal{ID}|^2 \times |\mathcal{O}|^{|\mathcal{ID}|+2}$. We check for every configuration in the current set whether it can reach another configuration by moving the pointers of two processes and update the current value. Note that moving the pointers adds another factor of $|\mathcal{O}|$. In every step a read is processed. Since we have at most $|\mathcal{O}|$ many reads, the overall running time of the algorithm is $|\mathcal{ID}|^2 \times |\mathcal{O}|^{|\mathcal{ID}|+3}$. With $|\mathcal{ID}|$ fixed, the resulting deterministic algorithm processes an input $\mathcal{O}_x$ in polynomial time. With Lemma 5.4, we have:

THEOREM 5.5. $\text{TEST}_P(\text{CC})$ *is in* **P**.

According to Definition 3.5, a test succeeds under SLOW if and only if there are constructive serial views for every process p and variable x such that the following holds. The view contains all operations of p on x and all writes to x of other processes. This testing problem amounts to a task that is similar to CC but with the additional restriction that only one process contains reads. The presented algorithm A can be used to solve the testing problem for SLOW [Furbach et al. 2014].

THEOREM 5.6. $\text{TEST}_P(\text{SLOW})$ *is in* **P**.

Similar to this approach is the polynomial algorithm for $\text{TEST}_L(\text{PRAM})$.

THEOREM 5.7. $\text{TEST}_L(\text{PRAM})$ *is in* **P**.

The PRAM model requires a serial view per process. In a recursion, we traverse the reads of this process and attempt to assign the writes they could access from other processes. It is enough to check the earliest matching write from each process. The depth of the recursion is bounded by the length of the test, which we assume to be fixed. From this observation, we obtain a polynomial runtime.

6. TESTING IS IN NP

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 [Steinke and Nutt 2004] 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 the results of Steinke and Nutt, appears much harder.

6.1. Building Blocks of a Uniform Reduction

We define two propositional formulas in conjunctive normal form: $\text{EXE}(\mathcal{T})$ and $\text{SV}(\mathcal{T}, \mathcal{O}, <)$. 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 operations $w, r \in \mathcal{T}$ that use the same variable and access the same value, $var(w) = var(r)$

and $val(w) = val(r)$. Formula $\mathrm{EXE}(\mathcal{T})$ is the following Conjunction (2). It requires that every read has a write providing its value (left) and no read has two sources (right):

$$\bigwedge_{r \in \mathcal{T}} \left[\bigvee_{\substack{w \in \mathcal{T} \\ var(w)=var(r) \\ val(w)=val(r)}} ex_{w,r} \quad \wedge \bigwedge_{\substack{r, w_1, w_2 \in \mathcal{T}, w_1 \neq w_2 \\ var(w_1)=var(w_2)=var(r) \\ val(w_1)=val(w_2)=val(r)}} (\neg ex_{w_1,r} \vee \neg ex_{w_2,r}) \right]. \tag{2}$$

LEMMA 6.1. $\mathrm{EXE}(\mathcal{T})$ *is in CNF and cubic in the size of* $\mathcal{T}$*. Moreover,* $\mathrm{EXE}(\mathcal{T})$ *is satisfiable if and only if there is an execution* $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$*.*

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

Formally, a serial view is a strict total order $<_{sv} \subseteq \mathcal{O} \times \mathcal{O}$. We encode it with variables sv_{op_1,op_2}, one for each pair of operations $op_1, op_2 \in \mathcal{O}$. Intuitively, variable sv_{op_1,op_2} is set to true iff $op_1 <_{sv} op_2$ holds. The following exclusive-or ensures the serial view is total and asymmetric. The implication is transitivity. We use the exclusive-or as a macro for a conjunction and the implication as a macro for a disjunction so that the resulting formula is in conjunctive normal form:

$$\bigwedge_{\substack{op_1, op_2, op_3 \in \mathcal{O} \\ op_1 \neq op_3 \\ op_1 \neq op_2 \neq op_3}} \left[(sv_{op_1,op_2} \oplus sv_{op_2,op_1}) \quad \wedge \quad (sv_{op_1,op_2} \wedge sv_{op_2,op_3} \rightarrow sv_{op_1,op_3}) \right]. \tag{3}$$

Definition 3.3 requires that $<_{sv}$ refines $<$ to a total order:

$$\bigwedge_{\substack{op_1, op_2 \in \mathcal{O} \\ op_1 < op_2}} sv_{op_1,op_2}\ . \tag{4}$$

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

$$\bigwedge_{\substack{w, r \in \mathcal{O} \\ var(w)=var(r) \\ val(w)=val(r)}} \left[(\neg ex_{w,r} \vee sv_{w,r}) \quad \wedge \bigwedge_{\substack{w' \in \mathcal{O} \\ var(w')=var(r)}} (\neg ex_{w,r} \vee \neg sv_{w,w'} \vee \neg sv_{w',r}) \right]. \tag{5}$$

Formula $\mathrm{SV}(\mathcal{T}, \mathcal{O}, <)$ is the conjunction of the Formulas (3) to (5). To state the relationship between $SerialView\ (\mapsto, \mathcal{O}, <)$ in Definition 3.3 and $\mathrm{SV}(\mathcal{T}, \mathcal{O}, <)$, we restrict the satisfying assignments to the propositional variables. An assignment *respects* $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ if $op_1 \mapsto op_2$ holds if and only if ex_{op_1,op_2} is set to true.

LEMMA 6.2. $\mathrm{SV}(\mathcal{T}, \mathcal{O}, <)$ *is in CNF and cubic in its input. There is a strict total order* $<_{sv}$ *that is* $SerialView\ (\mapsto, \mathcal{O}, <)$ *if and only if* $\mathrm{SV}(\mathcal{T}, \mathcal{O}, <)$ *has a satisfying assignment that respects* $\mapsto$*.*

6.2. 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:

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

Test $\mathcal{T}$ succeeds under SLOW iff this formula is satisfiable. Note that the restriction on the admissible assignments in Lemma 6.2 is no longer needed: $\mathrm{EXE}(\mathcal{T})$ ensures that the assignment to the execution variables matches an execution.

THEOREM 6.3. TEST(M) *is in* **NP** *for all models* M *defined via serial views.*

All memory models in Figure 3 except TSO and PSO are defined via serial views. The testing problem for the latter has been shown to belong to **NP** in [Cantin et al. 2005].

7. CONCLUSIONS

We determined the complexity of the testing problem for most known weak memory models. Figure 7 shows a summary of our results that cover all models in the Steinke-Nutt hierarchy of Figure 3. 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. (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 an **NP** upper bound. 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 7 with only four reductions ($\mathbf{NPC}_{4-7}$) and three algorithms ($\mathbf{P}_{1-3}$). The algorithms in Sections 5.2 and 5.3 lead to the results $\mathbf{P}_1$ to $\mathbf{P}_3$. The **NPC** results $\mathbf{NPC}_4$, $\mathbf{NPC}_5$, $\mathbf{NPC}_6$, and $\mathbf{NPC}_7$ stem from the reductions in Sections 4.2, 4.3, 4.4, and 5.5, respectively.

Mem. Model	Complexity Class of TEST(M)			
	TEST(M)	TEST_P(M)	TEST_L(M)	TEST_V(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	$\mathbf{NPC}_4$	$\mathbf{P}_3$	$\mathbf{NPC}_6$	$\mathbf{NPC}_4$
PRAM	$\mathbf{NPC}_4$		$\mathbf{P}_2$	$\mathbf{NPC}_4$
SLOW	$\mathbf{NPC}_4$	$\mathbf{P}_3$	$\mathbf{P}_2$	$\mathbf{NPC}_4$
LOCAL	$\mathbf{P}_1$	$\mathbf{P}_1$	$\mathbf{P}_1$	$\mathbf{P}_1$

Fig. 7. Time complexity of the testing problem under the memory models in the Steinke-Nutt hierarchy of Figure 3. We use TEST_P(M), TEST_L(M), and TEST_V(M) for the restricted problems where the number of processes, their length, and the number of variables are fixed, respectively. **NPC** means **NP**-complete: the problem is **NP**-hard and in **NP**. Each index represents a different algorithm or reduction.

Finding range reductions is challenging. However, they provide insights into the synchronization capabilities of a memory model and guide the search for testing algorithms. Therefore, as future work we plan to fill the missing entries in Figure 7. Furthermore, range reductions could be used to analyse other problems like reachability or robustness.

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. As another line of future work, it would therefore be interesting to study more compact encodings, to SAT or to other **NP**-complete problems. Besides the size of the encoding, also the solver technology is important. To handle serial views, it should be beneficial to reduce to a problem with built-in support for transitive closures.

Acknowledgements

We thank the reviewers for their valuable comments on drafts of this paper that helped us improve the presentation. We thank one reviewer for pointing out a missing factor in the complexity analysis in Section 5.3.

REFERENCES

P.A. Abdulla, M.F. Atig, Y.F. Chen, C. Leonardsson, and A. Rezine. 2012. Counter-Example Guided Fence Insertion under TSO. In *TACAS (LNCS)*, Vol. 7214. Springer, 204–219.

S.V. Adve and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1996), 66–76.

M. Ahamad, R.A. Bazzi, R. John, P. Kohli, and G. Neiger. 1993. The Power of Processor Consistency. In *SPAA*. ACM, 251–260.

J. Alglave. 2010. *A Shared Memory Poetics*. Ph.D. Dissertation. University Paris 7.

J. Alglave. 2012. A Formal Hierarchy of Weak Memory Models. *FMSD* 41, 2 (2012), 178–210.

J. Alglave. 2013. Weakness is a Virtue. (2013). $(EC)^2$ Workshop.

J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. 2014. Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion. In *CAV (LNCS)*, Vol. 8559. Springer, 508–524.

J. Alglave, D. Kroening, and M. Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV (LNCS)*, Vol. 8044. Springer, 141–157.

J. Alglave and L. Maranget. 2011. Stability in Weak Memory Models. In *CAV (LNCS)*, Vol. 6806. Springer, 50–66.

M.F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. 2010. On the Verification Problem for Weak Memory Models. In *POPL*. ACM, 7–18.

M.F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. 2012. What's decidable about Weak Memory Models. In *ESOP (LNCS)*, Vol. 7211. Springer, 26–46.

M.F. Atig, A. Bouajjani, and G. Parlato. 2011. Getting Rid of Store Buffers in TSO Analysis. In *CAV (LNCS)*, Vol. 6806. Springer, 99–115.

A. Bouajjani, E. Derevenetc, and R. Meyer. 2013. Checking and Enforcing Robustness against TSO. In *ESOP (LNCS)*, Vol. 7792. Springer, 533–553.

A. Bouajjani, R. Meyer, and E. Möhlmann. 2011. Deciding Robustness against Total Store Ordering. In *ICALP (LNCS)*, Vol. 6756. Springer, 428–440.

S. Burckhardt and M. Musuvathi. 2008. Effective Program Verification for Relaxed Memory Models. In *CAV (LNCS)*, Vol. 5123. Springer, 107–120.

J. Burnim, K. Sen, and C. Stergiou. 2011. Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models. In *TACAS (LNCS)*, Vol. 6605. Springer, 11–25.

G. Calin, E. Derevenetc, R. Majumdar, and R. Meyer. 2013. A Theory of Partitioned Global Address Spaces. In *FSTTCS (LIPIcs)*, Vol. 24. Schloss Dagstuhl, 127–139.

J.F. Cantin, M.H. Lipasti, and J.E. Smith. 2005. The Complexity of Verifying Memory Coherence and Consistency. *IEEE Transactions on Parallel and Distributed Systems* 16, 7 (2005), 663–671.

E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV (LNCS)*, Vol. 1855. Springer, 154–169.

E. Derevenetc and R. Meyer. 2014. Robustness against Power is PSPACE-complete. In *ICALP (LNCS)*, Vol. 8573. Springer, 158–171.

J. Esparza and P. Ganty. 2011. Complexity of Pattern Based Verification for Multithreaded Programs. In *POPL*. ACM, 499–510.

A. Farzan and P. Madhusudan. 2009. The Complexity of Predicting Atomicity Violations. In *TACAS (LNCS)*, Vol. 5505. Springer, 155–169.

F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. 2014. Memory Model-aware Testing - A Unified Complexity Analysis. In *ACSD*. IEEE, 92–101.

P.B. Gibbons and E. Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244.

J.R. Goodman. 1991. *Cache Consistency and Sequential Consistency*. Technical Report 1006. University of Wisconsin-Madison.

A. Heddaya and H. Sinha. 1992. *Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing*. Technical Report BU-CS-92-004. Boston University.

J.L. Hennessy and D.A. Patterson. 2003. *Computer Architecture: A quantitative Approach* (3 ed.). Morgan Kaufmann.

P.W. Hutto and M. Ahamad. 1990. Slow Memory: Weakening Consistency to Enchance Concurrency in Distributed Shared Memories. In *ICDCS*. IEEE, 302–309.

D. Kozen. 1977. Lower Bounds for Natural Proof Systems. In *FOCS*. IEEE, 254–266.

L. Lamport. 1979. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (1979), 690–691.

R. Lawrence. 1998. A Survey of Cache Coherence Mechanisms in Shared Memory Multiprocessors. (1998).

R.J. Lipton and J.S. Sandberg. 1988. *PRAM: A Scalable Shared Memory*. Technical Report CS-TR-180-88. Princeton University.

F. Liu, N. Nedev, N. Prisadnikov, M.T. Vechev, and E. Yahav. 2012. Dynamic Synthesis for Relaxed Memory Models. In *PLDI*. ACM, 429–440.

P. Loewenstein, S. Chaudhry, R. Cypher, and C. Manovit. 2006. Multiprocessor Memory Model Verification. (2006). Automated Formal Methods Workshop (AFM).

D. Mosberger. 1993. Memory Consistency Models. *ACM SIGOPS: Operating Systems Review* 27, 1 (1993), 18–26.

R.C. Steinke and G.J. Nutt. 2004. A Unified Theory of Shared Memory Consistency. *JACM* 51, 5 (2004), 800–849.

D. Weaver and T. Germond (Eds.). 1994. *The SPARC Architecture Manual - Version 9*. Prentice-Hall.

1994. *The SPARC Architecture Manual - Version 9*. Prentice-Hall.