Memory Model-aware Testing

a Unified Complexity Analysis

Florian Furbach, Roland Meyer, Klaus Schneider, Maximilian Senftleben
TU Kaiserslautern
{furbach,meyer,schneider,senftleben}@cs.uni-kl.de

***Abstract*—To improve performance, multiprocessor systems implement weak memory consistency models — and a number of models have been developed over the past years. Weak memory models, however, lead to unforeseen program behavior, and there is a current need for memory model-aware program analysis techniques. The problem is that every memory model calls for new verification algorithms.**

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 the memory model. 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 approach from related results is a unified analysis. Instead of considering one memory model after the other, we build upon work of Steinke and Nutt. They showed that the existing memory models form a natural hierarchy where one model is called weaker than another one if it includes the latter's behavior [33].

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. We apply them to establish NP-lower bounds for the stronger memory models. (ii) For the weaker models, we present polynomial-time testing algorithms. Here, the common idea is determinization. (iii) Finally, we give a single SAT encoding of the testing problem that works for all memory models in the Steinke-Nutt hierarchy. It shows membership in NP. Our results are general enough to classify future weak memory models and ensure that SAT solvers are adequate tools for their analysis.

I. Introduction

A. Weak Memory Models

For multiprocessors, communication over shared memory soon becomes a performance bottleneck. To improve processor utilization, architectures implement various optimizations like a distributed shared memory or write buffers that are organized per memory cell or per processor. 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* [33], [2], [23], [28] 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.

In general, weak memory models allow for serializations of the memory operations that are impossible on a sequentially consistent (SC) memory [27]. Under SC, operations take effect instantaneously or, phrased differently, the sequences of memory operations are interleaved. SC matches the developer's intuition about the program behavior. As a result, algorithms that have been developed with SC in mind can have undesirable effects when run on a weak memory. In particular, mutual exclusion algorithms and other programs with data races behave incorrectly if the program order is relaxed only slightly. Therefore, considerable effort has been devoted to developing verification methods for concurrent programs that run under weak memory models (see below for a discussion).

B. The Testing Problem

A core problem of these 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 [20], is defined as follows. Given a sequence of read/write operations for each concurrent process, check whether there is an interleaving of these operations that satisfies the constraints of the weak memory model under study. We use the term *testing algorithms* to refer to algorithms that solves 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 the programs 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. When a may analysis finds a potential counterexample to a correctness statement, checking whether the counterexample is genuine or spurious amounts to solving a testing problem. If the counterexample is spurious, the failing test suggests a refinement of the may analysis, which leads to a CEGAR-like verification loop [18]. As an under-approximation, the testing problem occurs directly when debugging concurrent programs. A further application are synchronization inference algorithms [30], [8], [1], [12]. Their task is to determine the placement of synchronization primitives like fences, barriers, and syncs within a program. A final application is the estimation of best and worst case executing times. Here, testing algorithms can rule out infeasible paths to improve the analysis.

Despite its many applications, there are only few works on the algorithmics and complexity of the testing problem. Gibbons and Korach [20] studied the testing problem under SC and linearizability. They showed that in both cases the problem is **NP**-complete, whereas fixed-parameter variants can be solved in polynomial time. Cantin, Lipasti, and Smith [17] extended these results. They state **NP**-completeness of the testing problem under SPARC's memory models (TSO, PSO,

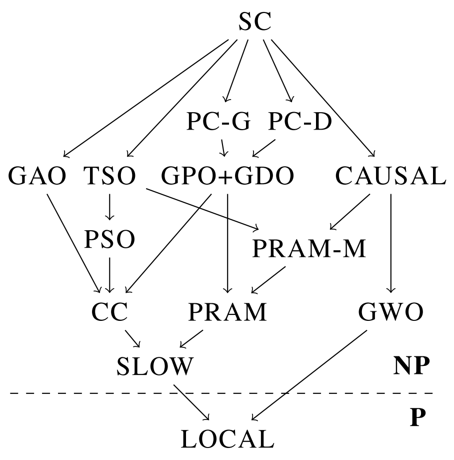

Fig. 1: Hierarchy of weak memory models (see [33]).

RMO) [34], processor consistency PC, release consistency, and a model of the PowerPC architecture. Common to both approaches is that they are tailored towards few specific memory models. The testing problem, however, is important for virtually all memory models. Moreover, since future architectures are likely to bring new memory models, general techniques are desirable 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* according to their weakness as done in [32], [33], [5]. A memory model M_w is thereby called *weaker than* another model M_s, denoted by $M_s \leq M_w$ and indicated by a path in Figure 1, if every execution allowed under M_s is also valid under M_w. Memory models are usually defined by axioms [31], in an operational way, or via local views. Steinke and Nutt [33] have shown that most weak memory models can be obtained as a combination of four basic models called GAO, GWO, GDO, and GPO, all of which can be formulated in a view-based manner. To be precise, this applies to SC, Pipelined RAM (PRAM) [29], CAUSAL consistency [24], cache consistency (CC) [21], two variants of processor consistency (PC-G, PC-D) [21], [3], SLOW consistency [24], and LOCAL consistency [22]. As a consequence of this characterization via basic models, these memory models form the hierarchy depicted in Figure 1. It shows that SC is the strongest and LOCAL consistency is the weakest model.

C. 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 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 provides a fine understanding of 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. Since new memory models are likely to come up, our general concepts will make it easy to adapt testing algorithms and prove their optimality. We elaborate on our main contributions.

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* $\mathrm{M}_S \leq \mathrm{M} \leq \mathrm{M}_W$. The concept of range reductions shows that hierarchies of architectures are not only useful from a semantic point of view (showing the relationship between models), but also helpful from an algorithmic point of view. Once established, they allow us to propagate hardness results between memory models. We present two range reductions. The first covers the range from SC to SLOW, and the second the range from SC to GWO. Since SLOW and GWO are the weakest models above LOCAL in the Steinke-Nutt hierachy of Figure 1, the two reductions are sufficient to conclude all hardness results.

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. That LOCAL admits a polynomial-time testing algorithm came as a surprise to us. A common belief is 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 1. 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 stress that it heavily relies on the view-based formulation of memory models [33]. Phrased differently, the real contribution here is to observe that the Steinke-Nutt framework is good for SAT encodings.

Contribution 4: Influence of parameters: In actual 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 small processes where the length of the read/write sequences is limited. Finally, the number of synchronization variables is usually small. For this reason, we consider fixed-parameter variants of the testing problem where one of these three parameters is bounded by a constant 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 theoretical curiosity on how a memory model influences the complexity of system analysis. Initially, we debated on what results to expect 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 [25] to non-primitive recursive for TSO, PSO, and an approximation of POWER [9], [10]. 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 [6]. 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**.

II. Related Work

We already discussed the related work on the testing problem [20], [17], but would like to add a remark on [17]. 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 [20]). This argument is not convincing if the purpose of testing is synchronization inference where programs come free from synchronization primitives.

Our contributions are precise complexity results for the testing problem. For weak memory models, precise 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 and PSO [9], as well as for an approximation of POWER [10]. Robustness requires the absence of causality cycles, and has been shown to be decidable in polynomial space for TSO [13], and for partitioned global address spaces [16]. The testing problem has a lower complexity since it handles single sequences of operations rather than sets. Our multi-parameter complexity analysis is related to [19]. 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 [14], [15]. Atig et al. extended the idea of bounded context switching to TSO [11]. Recently, Alglave et al. developed memory model-aware BMC algorithms [7]. The approach is remarkable in that it applies to various models, and indeed inspired our SAT encoding given in Section VI. 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 [26].

Finally, we discuss our choice to base this work on the Steinke-Nutt hierarchy rather than the recent framework of Alglave [4]. The reason is that the Steinke-Nutt hierarchy covers more models, which led to the idea of range reductions. Moreover, the view-based formulation was close to formal languages so that we were able to extract polynomial-time algorithms from it.

III. Tests and Memory Models

We first give the definition of tests and memory models following [33]. Then, we turn to the testing problem subject to this paper. For illustration purposes, we consider the mutex algorithm in Figure 2. This is a simplified version of Dekker's algorithm that does not use a token variable.

```
procedure P                    procedure Q
    x = 1;                         y = 1;
    while y=1 do                   while x=1 do
        x = 0;                         y = 0;
        Sleep(time);                   Sleep(time);
        x = 1;                         y = 1;
    Critical Section;              Critical Section;
    x = 0;                         y = 0;
```

Fig. 2: Example procedures implementing Dekker's mutex.

The algorithm consists of procedures P and Q that are executed by two processes. Both processes claim the resource by setting their corresponding variable to 1. If the resource is claimed by the partner, then a process releases the resource and waits some time until it claims the resource again. If the resource is not claimed by the partner, then the process enters its critical section and releases the resource afterwards. Intuitively, the protocol guarantees mutual exclusion. We will study a test for this program which shows that the guarantee depends on the memory model.

A. 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 operations already tell the value that is read.

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 $\perp$. 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 $\mathcal{N} := \{0, 1, \dots\}$ 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 $idx(op) = i$ to access the command, the variable, the value, the process identifier, and the issue index. Given a set of operations $\mathcal{T} \subseteq \mathcal{OP}$, 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 an operation with $cmd(op) = r$. To establish the initial value $\perp$

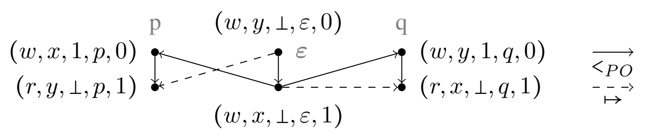

Fig. 3: The execution of test $\mathcal{T}$ = DEKKER

for every variable, we introduce write operations $(w,x,\bot,\varepsilon,i)$ that belong to the initialization process ε.

Definition 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 $|(w,x,\bot,\varepsilon,*)_{\mathcal{T}}| > 0$.*

The following is a test for the program given in Figure 2:

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

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 their partner.

For notational convenience, and like in the example above, we will define 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 assume the initial process ε only writes $\bot$ to each variable that is used. If we fix an ordering on the variables, this fully defines the initial process we can omit it as well. With these conventions, the above example reads

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

B. Memory Model-aware Semantics of Tests

The semantics of tests is defined in terms of executions, which are relations on the operations. Intuitively, these relations determine the write that a read receives its value from.

Definition 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 its 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. *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) [27]. Sequential consistency ensures that every process observed all operations in the order they were issued. It is the intuitive model which is assumed by most programmers. Using Definition 3, an execution is valid under SC if there is a serial view on all operations that respects the program order. The program order is defined as usual: 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:

$$\begin{aligned} op_1 \ <_p \ op_2 \quad \text{if} \quad & (proc(op_1) = proc(op_2) = p \\ & \quad \text{and } idx(op_1) < idx(op_2)) \\ & \text{or } (proc(op_1) = \varepsilon \text{ and } proc(op_2) = p). \end{aligned}$$

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 4. *An execution $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ is* valid under sequential consistency *if $\exists <_{sv} : <_{sv}$ is*

$$SerialView(\mapsto, \mathcal{T}, <_{PO}).$$

In the test for Dekker's protocol, there is only one execution, depicted in Figure 3, where the reads observe the writes of the initial process ε. Intuitively, at least one initial write is overwritten before the corresponding read can be executed. The formalism captures this as follows. Assume there was a strict total order $<_{sv}$ that satisfies the requirements in Definition 3 and 4. Since the serial view respects the program order $<_{PO}$, one of the reads is maximal in $<_{sv}$, say $(r,y,\bot,p,1)$. For the same reason, $(w,y,1,q,0)$ is larger than $(w,y,\bot,\varepsilon,0)$ in $<_{sv}$:

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

Since $(w,y,\bot,\varepsilon,0) \mapsto (r,y,\bot,p,1)$, we obtain a contradiction to Definition 3(ii). The argumentation is similar if the read on x is maximal, and indeed there is no strict total order that satisfies the requirements. This behavior changes if we examine the test under a weaker model.

Hutto and Ahamad [24] developed the SLOW model to solve the exclusion and dictionary problems with minimal consistency maintenance. The definition is as follows. A read returns a previously written value, and successive reads from the same variable may not return writes issued earlier (by the process that issued the source write) than the read one. Furthermore, local writes must be visible immediately. In the Steinke-Nutt framework, this requires a serial view for every process and every variable. It contains all write operations on this variable and all operations of this process on the variable, and respects the program order. We have [33, Theorem 3.7]:

Definition 5. *An execution* $\mapsto\ \subseteq \mathcal{T} \times \mathcal{T}$ *is* valid under SLOW consistency *if* $\forall p \in \mathcal{ID}\ \forall x \in \mathcal{V}\ \exists <_{sv} : <_{sv}$ *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 3 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:

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

C. Memory Model-aware Testing

We consider the testing problem $\textsc{Test}(\mathrm{M})$ for each of the memory models M shown in Figure 1. The formal definition of $\textsc{Test}(\mathrm{M})$ 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 is *successful under* M if there is a valid execution, otherwise it *fails under* M. In the example, test DEKKER fails under SC but succeeds under SLOW.

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

IV. 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, 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.

A. Range Reductions

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

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

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 $\mathrm{M}_S \leq \mathrm{M} \leq \mathrm{M}_W$.

$$\begin{tikzcd}
 & & \mathrm{M}_S \arrow[dl] \arrow[d] \arrow[dr] & \\
\textsc{Prob} \arrow[urr, "\text{Definition 6(ii)}"] & \mathrm{M}' \arrow[r, dash, dotted] \arrow[dr] & \mathrm{M} \arrow[r, dash, dotted] \arrow[d] & \mathrm{M}'' \arrow[dl] \\
 & & \mathrm{M}_W \arrow[ull, "\text{Definition 6(i)}"] &
\end{tikzcd}$$

Fig. 4: Illustration of $\mathrm{M}_S \leq \mathrm{M}_W$-range reductions of $\textsc{Prob}$ to the testing problem. Directed edges correspond to implications.

Definition 6. *A function f from instances of* $\textsc{Prob}$ *to tests is an* $\mathrm{M}_S \leq \mathrm{M}_W$-range reduction of $\textsc{Prob}$ 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.*

If function f is polynomial-time computable and $\textsc{Prob}$ is **NP**-hard, we derive **NP**-hardness of the testing problem.

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

Proof: We argue that $f : \textsc{Prob} \to \textsc{Test}(\mathrm{M})$ is a reduction of $\textsc{Prob}$ to $\textsc{Test}(\mathrm{M})$, which means Equivalence (1) holds. Since f is assumed to be polynomial-time computable, **NP**-hardness follows. For the implication from left to right, note that $\psi(x)$ implies $f(x) = \mathcal{T}$ succeeds under M_S by Definition 6.(ii). Since $\mathrm{M}_S \leq \mathrm{M}$, we conclude that $\mathcal{T}$ remains successful under M. For the reverse direction, assume $\mathcal{T}$ succeeds under M. Then the test remains successful under the weaker model M_W. With Definition 6.(i), we can therefore conclude $\psi(x)$ holds. ■

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 V, we will show that $\textsc{Test}(\mathrm{LOCAL})$ is indeed in **P**.

When developing range reductions, the challenge is to guarantee the implication in Definition 6(i): the hard problem follows from a successful test under the weak memory model. To derive this 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. The difficulty with weak executions is to ensure a consistent view of operations over multiple processes. To achieve this, we study the relaxations of the weak memory model (wrt. the strong model) and design a test that is insensitive to these relaxations.

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

We give an SC ≤ SLOW-range reduction of SAT to the testing problem. Consider a SAT instance of the form $\varphi = \bigwedge_{i\in I} cl_i$ with $cl_i = \bigvee_{j\in J_i} lit_j$. 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 SLOW, then the formula is satisfiable. The challenge is to satisfy this second requirement: conclude satisfiability of the SAT instance despite the weak executions of SLOW. The trick is to observe that SLOW preserves the order of operations on the same variable, and then only use one variable ξ 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 is

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

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:

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

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 the definition of t, we again use the iterated concatenation and assume the same total order of clauses as above:

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

The test process reads, one by one, each propositional variable x from ξ. Since the test only has one variable, this read of x from ξ deletes all previous writes of cl to ξ. The intuition is the following. If the matching write was from p_x, then the read of x from ξ discards the assignment of x to true. To satisfy the following reads (r,ξ,cl), we therefore have to use the writes (w,ξ,cl) from the second process n_x. This corresponds to x being false and thus satisfying cl.

Theorem 1. *The above function f is an* SC ≤ 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 ≤ M ≤ SLOW. *As the reduction only uses one variable, even* TEST_V(M) *is* **NP***-hard for all memory models* SC ≤ M ≤ SLOW.

Proof: We claim that the test $\mathcal{T}$ is indeed successful under SC if φ is satisfiable. To see this, note that a satisfying assignment to 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 to φ, note that the test only has one variable. Therefore, there are no reorderings of operations. 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 operations to the same variable cannot be reordered — which is the case in most memory models. The range reduction relies, however, on an arbitrary number of processes. As we will show, TEST(SLOW) becomes polynomial if we fix this parameter.

C. Fixed Length 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 I and [33]). It requires the following consistency: if a process observes an order between two writes (due to a read), then all processes agree on the order. To render this formally, we introduce the *write-read-write order* (WO), where $w_1 <_{WO} w_2$ if

$$\exists r \in \mathcal{T}: \ w_1 \mapsto r <_{PO} w_2$$

Definition 7. *An execution* $\mapsto\, \subseteq \mathcal{T}\times\mathcal{T}$ *is* valid under GWO consistency *if* $\forall p \in \mathcal{ID}\ \exists <_p : <_p$ *is*

$$SerialView(\mapsto, (*,*,*,p,*)_{\mathcal{T}} \cup (w,*,*,*,*)_{\mathcal{T}}, <_p \cup <_{WO})$$

We give an SC ≤ GWO-range reduction of SAT to the testing problem. Let $\varphi = \bigwedge_{1\le k\le K} cl_k$ with $cl_k = \bigvee_{1\le j\le J_k} lit_{k,j}$, and let N be the number of variables in 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. For each variable x_i in the SAT instance φ there are two corresponding variables $x_i, y_i \in \mathcal{V}$, and for each clause cl_k there is a variable $c_k \in \mathcal{V}$. There are auxiliary variables $h_0 \ldots h_K$. The data domain $\mathcal{D}$ is $\{0,1,2\}$. 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

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

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

$$\begin{aligned} n_i &:= (w,x_i,0).(r,x_i,0).(w,y_i,1) \\ p_i &:= (w,x_i,1).(r,x_i,1).(w,y_i,1) \\ q_i &:= (r,y_i,1).(w,x_i,2). \end{aligned}$$

The idea is that at least one of the writes to x_i by p_i or n_i is overwritten with value 2 by process q_i. Since the write-read-write order is preserved under GWO, all processes which read 2 from x_i before reading another value from it observe the same last write.

There is a process $l_{k,j}$ for each literal $lit_{k,j}$. It checks whether the literal is satisfied by the guessed assignment. The process waits for the corresponding variable to be overwritten by 2 and afterwards reads the satisfying value. To be precise, if it is a negative literal of variable x then it reads 0 from x, otherwise it reads 1 from x. After the reads, the process writes 1 to variable c_k that corresponds to the literal's clause:

$$l_{k,j} \;:=\; (r, v(k,j), 2).(r, v(k,j), b(k,j)).(w, c_k, 1).$$

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 auxiliary variable h_{k-1} of the previous test process, tries to read the clause variable c_k, and then writes its own auxiliary variable h_k:

$$t_k \;:=\; (r, h_{k-1}, 1).(r, c_k, 1).(w, h_k, 1).$$

Process $h := (w, h_0, 1)$ just writes the first of the auxiliary variables. 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 variable h_K is set to 1, and then writes 0 and 1 to x_i:

$$r_i \;:=\; (r, h_K, 1).(w, x_i, 0).(w, x_i, 1).$$

We claim that test $\mathcal{T}$ is successful under SC if φ is satisfiable. Consider a satisfying assignment Φ of φ. Starting with the first variable x_i, 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 finish. We repeat this for all variables and complete by executing process h. Now the processes t_k can execute one after the other. To see this, note that we assume Φ to be satisfying. This means for each clause cl_k there is a literal process that wrote c_k. To conclude, we only have to guarantee that the remaining literal processes terminate. Starting with the first 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 of 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, each process t_k reads value 1 from the clause variable c_k. 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 executes before $(w, x_i, 2)$.

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

Theorem 2. *Function f is an* SC ≤ 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 ≤ M ≤ GWO. *As the reduction requires a process length of at most three, even* TEST$_L$(M) *is* **NP***-hard for all memory models* SC ≤ M ≤ GWO.

V. Some Testing Problems are in **P**

We show that for very weak memory models the general testing problem or restricted variants 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(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 therefore we only have to compute the serial views. We now develop concepts that make this argument work.

A. Read Partitioning and Constructive Serial Views

The catch in the above 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 $SerialView(\mapsto, \mathcal{O}_1, <_1)$ up 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. 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(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 2. *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 $1 \le i \le k$.*

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.

B. TEST(LOCAL) *is in* **P**

LOCAL consistency was defined as the weakest constraint that every shared memory system should satisfy [22]. 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 [33, Theorem 3.8]:

Definition 8. *An execution* $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$ *is* valid under LOCAL consistency *if* $\forall_{p \in \mathcal{T}}\ \exists <_{sv} : <_{sv}$ *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 $\tilde{op}_p = op_1 \dots op_n$. The writes of the other processes are given as an unordered set.

Algorithm 1 Compute Constructive Serial View for LOCAL Consistency

```
Input: Process $p$ with $\tilde{op}_p = op_1 \dots op_n$ and set of writes $W$
    of all processes $q \neq p$.
Output: Constructive serial view $s$, initially empty, $s := \varepsilon$.
 1: $last[x] := \bot$ for all $x \in \mathcal{V}$
 2: for $i = 1 \to n$ 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 $last[x] \neq v$ then
 8:         if $\exists w \in W : var(w) = x$ and $val(w) = v$ then
 9:             $W := W \setminus \{w\}$;
10:             $last[x] := v$;
11:             $s := s.w.op_i$
12:         else
13:             return not LOCAL consistent
14: return $s$ with remaining writes of $W$ inserted at the end
```

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 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 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**.

C. TEST$_P$(SLOW) *is in* **P**

Although general testing is **NP**-hard for SLOW, 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 only exponential in the number of processes.

By Definition 5, we have to find constructive serial views for each process $p \in \mathcal{ID}$ and every variable $x \in \mathcal{V}$. The corresponding subset of operations $\mathcal{O} \subseteq \mathcal{T}$ contains all writes to x in the test, and moreover all reads from x in process p. The serial view has to respect the program order. This means the operations in $\mathcal{O}$ are given as sequences $\tilde{op}_q = op_{q,1} \dots op_{q,n_q}$ for every process q. The task is to find an interleaving of these sequences so that every read $op_{p,i} = r$ obtains the desired value. Wlog., we can assume that no two reads (r,x,v) follow each other in $\tilde{op}_p$, and that no write (w,x,v) of p gives the value to a subsequent read (r,x,v).

We devise a non-deterministic algorithm that reads the sequences of operations for all processes. The algorithm alternates between operations from process p and operations from other processes. It first consumes a sequence of operations from p up to the next read (r,x,v). By the above assumptions, the last value that p writes to x is different from v. Therefore, the algorithm non-deterministically chooses a process $q \neq p$ that contains a write (w,x,v), and consumes the remaining sequence $\tilde{op}_q$ up to and including the first such write. Now (r,x,v) is enabled and the algorithm again consumes the operations of p up to the next read. When $\tilde{op}_p$ has been processed, the algorithm accepts the remaining operations.

Lemma 3. *Given sequences* $\tilde{op}_q$ *for all processes* q *in test* $\mathcal{T}$ *as input, the algorithm has an accepting run iff there is a constructive serial view* $<_{csv}$ *of* $(*,x,*,p,*)_{\mathcal{T}} \cup (w,x,*,*,*)_{\mathcal{T}}$ *that respects the program order.*

Proof: A constructive serial view is given by the order in which the algorithm consumes the operations of the input. It remains to prove completeness. Given some constructive serial view suitable for SLOW, we modify it to a sequence of operations resulting from an accepting run of the algorithm. The key idea is as follows. Whenever a read receives its value from (w,x,v,q,j), we select the earliest write (w,x,v,q,i) with $i < j$ in q that gives value v. ■

To solve the testing problem in polynomial time, it remains to determinize the non-deterministic algorithm. To this end, we introduce, for every process q, a pointer referencing the first operation in $\tilde{op}_q$ 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.

To determinize the algorithm, we store sets of pointer configurations. Like in the powerset construction for finite automata, the current set contains all pointer configurations that the non-deterministic algorithm could have reached after processing the input so far. With the set of possible pointer configurations, the algorithm no longer has to guess the process $q \neq p$ whose write (w,x,v) serves a read (r,x,v) of p. Instead, we compute all successor pointer configurations. To determine the successor set takes time $|\mathcal{O}|^{2|\mathcal{ID}|}$. Indeed, we check for every pointer configuration in the current set whether it can reach another pointer configuration by moving the pointer of a single process. Since we have at most $|\mathcal{O}|$ many reads in process p, the overall running time of the algorithm is $|\mathcal{O}|^{2|\mathcal{ID}|+1}$. With $|\mathcal{ID}|$ fixed, the algorithm is polynomial.

Theorem 4. *The algorithm is deterministic and solves* $\text{TEST}_P(\text{SLOW})$ *in polynomial time.*

VI. 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 Boolean satisfiability. 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 [33] 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.

A. Building Blocks of a Uniform Reduction

We define two propositional formulas in conjunctive normal form (CNF): $\text{EXE}(\mathcal{T})$ 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 $\text{EXE}(\mathcal{T})$ is the following Conjunction (2). It encodes the fact that every read has a write that gives its value (left) and no read has two sources (right):

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

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* $\mapsto \subseteq \mathcal{T} \times \mathcal{T}$.

Satisfiability of the second formula $\text{SV}(\mathcal{T}, \mathcal{O}, <)$ 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 $\text{EXE}(\mathcal{T})$, formula $\text{SV}(\mathcal{T}, \mathcal{O}, <)$ 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 this relation 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:

$$\bigwedge_{\substack{op_1, op_2, op_3 \in \mathcal{O} \\ op_1 \neq op_3 \\ op_1 \neq op_2 \neq op_3}} (sv_{op_1,op_2} \oplus sv_{op_2,op_1})$$

$$\wedge \quad (sv_{op_1,op_2} \wedge sv_{op_2,op_3} \rightarrow sv_{op_1,op_3}). \tag{3}$$

The first requirement in Definition 3 is 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 $w <_{sv} r$ (left) so that no write to the variable is placed in between the two (right):

$$\bigwedge_{\substack{w, r \in \mathcal{O} \\ var(w)=var(r) \\ val(w)=val(r)}} (\neg ex_{w,r} \vee sv_{w,r})$$

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

Formula $\text{SV}(\mathcal{T}, \mathcal{O}, <)$ is the conjunction of the Formulas (3) to (5). To state the relationship between $\mathit{SerialView}(\mapsto, \mathcal{O}, <)$ in Definition 3 and $\text{SV}(\mathcal{T}, \mathcal{O}, <)$, 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 5. $\text{SV}(\mathcal{T}, \mathcal{O}, <)$ *is in CNF and cubic in its input. There is a strict total order* $<_{sv}$ *that is* $\mathit{SerialView}(\mapsto, \mathcal{O}, <)$ *iff* $\text{SV}(\mathcal{T}, \mathcal{O}, <)$ *has a satisfying assignment that respects* $\mapsto$.

B. 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 $\text{TEST}(\text{SLOW})$ to SAT. SLOW consistency serves as a representative example. The other models in the hierarchy only differ in the serial views they require.

Finding an execution amounts to finding an assignment that satisfies $\text{EXE}(\mathcal{T})$. To ensure the required serial views exist, we instantiate formula $\text{SV}(\bullet, \bullet, \bullet)$ with appropriate parameters:

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

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

Theorem 5. $\text{TEST}(\text{SLOW})$ *is in* **NP**.

In this way, show that the testing problem is in **NP** for all memory models defined via serial views. These are all models in Figure 1 except TSO and PSO. Their testing problems have been shown to belong to **NP** in [17].

VII. CONCLUSIONS

We determined the complexity of the testing problem for most known weak memory models. Figure 5 shows a summary of our results that cover all models in the Steinke-Nutt hierarchy of Figure 1. To derive these results, we developed three general concepts. (1) With range reductions, we proposed a general proof technique for lower bounds that hold for a range of memory models. This way, we learned about the importance

Mem. Model	Complexity Class of TEST(M)			
	TEST(M)	TEST_P(M)	TEST_L(M)	TEST_V(M)
SC	**NPC** (2)	**NPC** (7)	**NPC** (6)	**NPC** (2)
TSO	**NPC** (2)	**NPC** (7)	**NPC** (6)	**NPC** (2)
PSO	**NPC** (2)	$\mathbf{NPC}_7$	**NPC** (6)	**NPC** (2)
PC-G	**NPC** (2)		**NPC** (6)	**NPC** (2)
PC-D	**NPC** (2)		**NPC** (6)	**NPC** (2)
GAO	**NPC** (2)		**NPC** (6)	**NPC** (2)
GPO+GDO	**NPC** (2)		**NPC** (6)	**NPC** (2)
Causal	**NPC** (2)		**NPC** (3)	**NPC** (2)
PRAM-M	**NPC** (2)			**NPC** (2)
GWO	**NPC** (3)		$\mathbf{NPC}_3$	
CC	**NPC** (2)	$\mathbf{P}_5$	$\mathbf{NPC}_6$	**NPC** (2)
PRAM	**NPC** (2)		$\mathbf{P}_4$	**NPC** (2)
SLOW	**NPC** (2)	**P** (5)	**P** (4)	$\mathbf{NPC}_2$
LOCAL	$\mathbf{P}_1$	**P** (1)	**P** (1)	**P** (1)

Fig. 5: Time complexity of the testing problem under the memory models in the Steinke-Nutt hierarchy of Figure 1. 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**.

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 5 with only seven proofs (four of which are presented in this paper).

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 5.

REFERENCES

[1] Abdulla, P., Atig, M., Chen, Y., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS. LNCS, vol. 7214, pp. 204–219. Springer (2012)

[2] Adve, S., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)

[3] Ahamad, M., Bazzi, R., John, R., Kohli, P., Neiger, G.: The power of processor consistency. In: SPAA. pp. 251–260. ACM (1993)

[4] Alglave, J.: A Shared Memory Poetics. Ph.D. thesis, University Paris 7 (2010)

[5] Alglave, J.: A formal hierarchy of weak memory models. FMSD 41(2), 178–210 (2012)

[6] Alglave, J.: Weakness is a virtue (2013), $(EC)^2$ Workshop

[7] Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV. LNCS, vol. 8044, pp. 141–157. Springer (2013)

[8] Alglave, J., Maranget, L.: Stability in weak memory models. In: CAV. LNCS, vol. 6806, pp. 50–66. Springer (2011)

[9] Atig, M., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL. pp. 7–18. ACM (2010)

[10] Atig, M., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models. In: ESOP. LNCS, vol. 7211, pp. 26–46. Springer (2012)

[11] Atig, M., Bouajjani, A., Parlato, G.: Getting rid of store buffers in TSO analysis. In: CAV. LNCS, vol. 6806, pp. 99–115. Springer (2011)

[12] Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP. LNCS, vol. 7792, pp. 533–553. Springer (2013)

[13] Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: ICALP. LNCS, vol. 6756, pp. 428–440. Springer (2011)

[14] Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. LNCS, vol. 5123, pp. 107–120. Springer (2008)

[15] Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: TACAS. LNCS, vol. 6605, pp. 11–25. Springer (2011)

[16] Calin, G., Derevenetc, E., Majumdar, R., Meyer, R.: A theory of partitioned global address spaces. In: FSTTCS. LIPIcs, vol. 24, pp. 127–139. Schloss Dagstuhl (2013)

[17] Cantin, J., Lipasti, M., Smith, J.: The complexity of verifying memory coherence and consistency. IEEE Transactions on Parallel and Distributed Systems 16(7), 663–671 (2005)

[18] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. LNCS, vol. 1855, pp. 154–169. Springer (2000)

[19] Esparza, J., Ganty, P.: Complexity of pattern based verification for multithreaded programs. In: POPL. pp. 499–510. ACM (2011)

[20] Gibbons, P., Korach, E.: Testing shared memories. SIAM Journal on Computing 26(4), 1208–1244 (1997)

[21] Goodman, J.: Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin-Madison (1991)

[22] Heddaya, A., Sinha, H.: Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Boston University (1992)

[23] Hennessy, J., Patterson, D.: Computer Architecture: A quantitative Approach. Morgan Kaufmann, 3 edn. (2003)

[24] Hutto, P., Ahamad, M.: Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In: ICDCS. pp. 302–309. IEEE (1990)

[25] Kozen, D.: Lower bounds for natural proof systems. In: FOCS. pp. 254–266. IEEE (1977)

[26] Kuperstein, M., Vechev, M., Yahav, E.: Partial coherence abstractions for relaxed memory models. In: PLDI. pp. 187–198. ACM (2011)

[27] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers 28(9), 690–691 (1979)

[28] Lawrence, R.: A survey of cache coherence mechanisms in shared memory multiprocessors (1998)

[29] Lipton, R., Sandberg, J.: PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University (1988)

[30] Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI. pp. 429–440. ACM (2012)

[31] Loewenstein, P., Chaudhry, S., Cypher, R., Manovit, C.: Multiprocessor memory model verification (2006), Automated Formal Methods Workshop (AFM)

[32] Mosberger, D.: Memory consistency models. ACM SIGOPS: Operating Systems Review 27(1), 18–26 (1993)

[33] Steinke, R., Nutt, G.: A unified theory of shared memory consistency. JACM 51(5), 800–849 (2004)

[34] The SPARC Architecture Manual-Version 9. Prentice-Hall (1994)