Optimal Stateless Model Checking for Reads-From Equivalence under Sequential Consistency

PAROSH AZIZ ABDULLA, Uppsala University, Sweden
MOHAMED FAOUZI ATIG, Uppsala University, Sweden
BENGT JONSSON, Uppsala University, Sweden
MAGNUS LÅNG, Uppsala University, Sweden
TUAN PHONG NGO, Uppsala University, Sweden
KONSTANTINOS SAGONAS, Uppsala University, Sweden

We present a new approach for stateless model checking (SMC) of multithreaded programs under Sequential Consistency (SC) semantics. To combat state-space explosion, SMC is often equipped with a partial-order reduction technique, which defines an equivalence on executions, and only needs to explore one execution in each equivalence class. Recently, it has been observed that the commonly used equivalence of Mazurkiewicz traces can be coarsened but still cover all program crashes and assertion violations. However, for this coarser equivalence, which preserves only the reads-from relation from writes to reads, there is no SMC algorithm which is (i) *optimal* in the sense that it explores precisely one execution in each reads-from equivalence class, and (ii) *efficient* in the sense that it spends polynomial effort per class. We present the first SMC algorithm for SC that is both optimal and efficient *in practice*, meaning that it spends polynomial time per equivalence class on all programs that we have tried. This is achieved by a *novel test* that checks whether a given reads-from relation can arise in some execution. We have implemented the algorithm by extending Nidhugg, an SMC tool for C/C++ programs, with a new mode called rfsc. Our experimental results show that Nidhugg/rfsc, although slower than the fastest SMC tools in programs where tools happen to examine the same number of executions, always scales similarly or better than them, and outperforms them by an exponential factor in programs where the reads-from equivalence is coarser than the standard one. We also present two non-trivial use cases where the new equivalence is particularly effective, as well as the significant performance advantage that Nidhugg/rfsc offers compared to state-of-the-art SMC and systematic concurrency testing tools.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Theory of computation** → *Verification by model checking*.

Additional Key Words and Phrases: concurrent programs, sequential consistency, program verification, stateless model checking, dynamic partial order reduction

ACM Reference Format:
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-From Equivalence under Sequential Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 150 (October 2019), 29 pages. https://doi.org/10.1145/3360576

Authors' addresses: Parosh Aziz Abdulla, Uppsala University, Sweden, parosh@it.uu.se; Mohamed Faouzi Atig, Uppsala University, Sweden, mohamed_faouzi.atig@it.uu.se; Bengt Jonsson, Uppsala University, Sweden, bengt@it.uu.se; Magnus Lång, Uppsala University, Sweden, magnus.lang@it.uu.se; Tuan Phong Ngo, Uppsala University, Sweden, tuan-phong.ngo@it.uu.se; Konstantinos Sagonas, Uppsala University, Sweden, kostis@it.uu.se.

2475-1421/2019/10-ART150
https://doi.org/10.1145/3360576

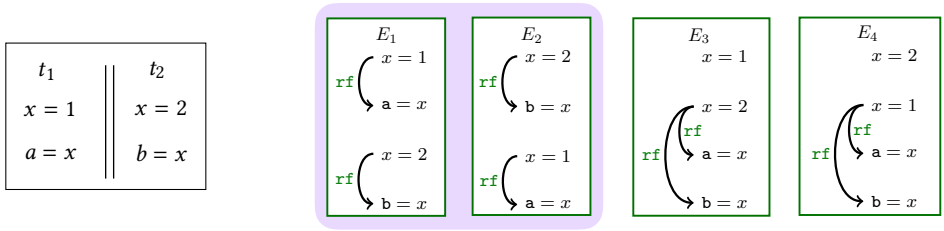

Fig. 1. Simple program: (left) pseudo-code and (right) executions corresponding to four Mazurkiewicz traces. The shaded background indicates that executions E_1 and E_2 are equivalent.

1 INTRODUCTION

Ensuring correctness of concurrent programs is difficult. Under *sequential consistency* (SC), the memory model to which we confine ourselves in this paper, one must consider all the different ways in which actions of threads can be interleaved. A successful technique for finding concurrency bugs (i.e., defects that arise only under some thread schedulings) and for verifying their absence is *stateless model checking* (SMC) [Godefroid 1997]. Given a terminating program, which may be annotated with assertions, SMC systematically explores the set of all thread schedulings that are possible during runs of this program. A special runtime scheduler drives the SMC exploration by making decisions on scheduling whenever such choices may affect the interaction between threads. Given enough time, the exploration covers all possible executions and detects any unexpected program results, program crashes, or assertion violations. The technique is entirely automatic, has no false positives, does not consume excessive memory, and can reproduce the concurrency bugs it detects. SMC has been implemented in many tools (e.g., VeriSoft [Godefroid 2005], Chess [Musuvathi et al. 2008], Concuerror [Christakis et al. 2013], Nidhugg [Abdulla et al. 2015], rInspect [Zhang et al. 2015], CDSChecker [Norris and Demsky 2016], and RCMC [Kokologiannakis et al. 2018]), and successfully applied to realistic concurrent programs (e.g., by Godefroid et al. [1998] and Kokologiannakis and Sagonas [2017]).

SMC faces the problem that the number of possible thread schedulings grows exponentially with the length of program execution, and must therefore be equipped with techniques to reduce the number of explored executions. The most prominent one is *partial order reduction* [Clarke et al. 1999; Godefroid 1996; Peled 1993; Valmari 1991], adapted to SMC as *dynamic partial order reduction* (DPOR) [Abdulla et al. 2014; Flanagan and Godefroid 2005; Sen and Agha 2007]. DPOR is based on the observation that two executions can be regarded as equivalent if they induce the same ordering between conflicting statement executions (called *events*), and that it is therefore sufficient to explore at least one execution in each equivalence class. Such equivalence classes are called *Mazurkiewicz traces* [Mazurkiewicz 1987].

As an illustration, Fig. 1 shows a simple program with two threads, t_1 and t_2, that access a shared variable x. Each thread writes to the variable and reads from it into a local register, a resp. b. We would like to explore the possible executions of this program, e.g., to check whether the program can satisfy $a = 2$ and $b = 1$ upon termination. Under SC [Lamport 1979], the interleaving model that we consider here, executions of this program fall into four equivalence classes (Mazurkiewicz traces), corresponding to the possible ways in which the writes can be ordered relative to the other events. Figure 1 shows one representative execution from each equivalence class as $E_1, \ldots, E_4$, respectively. Any SMC algorithm based on Mazurkiewicz traces (e.g., the DPOR algorithms by Flanagan and Godefroid [2005] or by Abdulla et al. [2017]), must thus explore at least four executions. However, when checking for unexpected program results, assertion violations, or crashes, it is possible to reduce this number even more by weakening the equivalence of Mazurkiewicz traces. Namely, a

closer inspection reveals that the two executions E_1 and E_2 are equivalent in the sense that each thread goes through the same sequences of local states and computes the same results. This is because the read operations get their values from the same write operation in both executions. To make this precise, introduce the relation `rf` ("reads-from") on the events of an execution, which connects a write with any read that reads its value. We can see that if two executions have the same `rf` relation, then each thread performs the same local computation. When annotating the executions in Fig. 1, we see that E_1 and E_2 have the same `rf` relation. Under Mazurkiewicz equivalence, E_1 and E_2 would be distinguished by the ordering of writes, but this is not relevant for the computed results.

The preceding example illustrates that there is a potential for improving the efficiency of SMC algorithms by using an equivalence which preserves only the `rf` relation. We introduce the term `rf` *equivalence* for this equivalence[1] and the term `rf` *trace*, or just *trace* for an equivalence class. In this small example, the improvement is modest. (It reduces the number of explored executions from four to three.) However, in bigger programs, it can be significant, sometimes even exponential, as we will show in Sect. 8. Intuitively, `rf` equivalence is the coarsest equivalence that respects both the local control flow of each thread, and the flow of data between events of different threads.

To exploit the potential efficiency gains offered by the `rf` equivalence fully, we should design a SMC algorithm that is both (i) *optimal* in the sense that it explores precisely one execution in each equivalence class, and (ii) *efficient* in the sense of spending only a small (at most polynomial in the size of the program) effort per equivalence class. Intuitively, such an algorithm would guarantee to explore the possible combinations of control flow and data flow in a non-redundant manner.

Several recent SMC techniques [Aronis et al. 2018; Chalupa et al. 2018; Huang 2015; Norris and Demsky 2016] try to exploit the potential offered by equivalences that are coarser than Mazurkiewicz traces. However, they are far from optimal in the above sense, since in general they explore a significant number of different executions that are `rf` equivalent. The DC-DPOR algorithm of Chalupa et al. [2018] explicitly uses `rf` equivalence as a criterion, but it is neither optimal nor efficient, except for the very restricted case when the communication graph on the set of threads (which connects two threads if they share a variable) is acyclic.

There is a fundamental obstacle which makes it difficult to construct an efficient SMC algorithm for `rf` equivalence. To explain it, we note that a natural way for an SMC algorithm to find new executions is by modifying the `rf` relation of previously explored executions. For instance, in Fig. 1 it could try to generate a new execution from E_4 by letting the third statement ($a = x$) read from the first ($x = 2$) instead of the second. It should then check whether the new `rf` relation is *consistent*, i.e., whether it can arise by reordering the statements to obtain an actual SC execution. In this example, the new `rf` relation is in fact *not* consistent, and so will not induce a new execution. A fundamental obstacle for efficiency is now that the problem of checking consistency of a given `rf` relation under SC is NP-complete in the size of the program (which is based on the length of the program and the number of threads) [Gibbons and Korach 1997, Theorem 4.1]. An SMC algorithm which generates new `rf` relations as modifications of explored ones and checks their consistency will therefore not be efficient in the worst case. In fact, the NP-completeness of consistency makes the construction of efficient SMC algorithms significantly more challenging for SC than for many other memory models; e.g., Release-Acquire (RA) for which consistency can be checked in polynomial time [Abdulla et al. 2018].

In this paper, we present a new SMC algorithm for checking multithreaded programs under SC, which is both (i) optimal and (ii) efficient *in practice*. First, we achieve optimality by a strategy for exploring executions, which by construction prevents generating executions that are equivalent to

[1] Chalupa et al. [2018] use the term *observation equivalence*, which we think is unfortunate since in the context of process algebra it is a well-known term which means something completely different [Hennessy and Milner 1980].

already explored ones. It is in part inspired by the strategy of the SMC algorithm for multithreaded programs under the RA semantics [Abdulla et al. 2018], but is redesigned to minimize the required calls to a consistency decision procedure by maintaining ordering information that makes such calls unnecessary. Second, we achieve "efficiency in practice" by a *novel check* for consistency of a given `rf` relation. The consistency check consists of three phases. The first two of them constitute a sound check for consistency which is polynomial time in the size of the program, but incomplete: the first phase detects inconsistency and the second phase detects consistency. The third phase is a sound and complete check to detect both consistency and inconsistency. More specifically:

(1) The first phase is a *saturation* procedure, which extends a given `rf` relation by orderings that must be respected by any execution with the given `rf` relation. If the resulting extension is cyclic, the saturation procedure returns "inconsistent", otherwise the check continues below.
(2) The second phase attempts to construct an execution that respects the given `rf` relation, which becomes a witness for consistency. The phase makes a single attempt, using as guidance a previously explored execution. If this is successful the check returns "consistent", otherwise it moves to the third phase.
(3) The third phase consists of a *novel decision algorithm*, which decides consistency in time that is polynomial in the length of the program and exponential in the number of threads.

An important finding is that the polynomial-time test represented by the first two phases has so far not returned "unknown" for any `rf` relation on any benchmark or "real" program that we have tried, implying that using this polynomial-time test is sufficient for checking consistency of `rf` relations on "typical" programs. For this reason, we claim that our algorithm is efficient in practice. As a further comment, the fact that the decision algorithm in the third phase runs in polynomial time in the length of the program implies that for the case where the number of threads is bounded, our SMC algorithm is in fact both optimal and efficient (i.e., not only "efficient in practice").

We have implemented our algorithm by extending Nidhugg, an SMC tool for multithreaded C/C++ programs based on the LLVM tool chain, with a new mode called rfsc. Our experimental evaluation (Sect. 8), on a wide range of benchmarks, shows that Nidhugg/rfsc: (i) although slower than the fastest SMC tools when the number of `rf` traces coincides with the number of Mazurkiewicz traces, scales similarly to those tools in these cases (i.e., in these cases, the performance difference is only a small constant factor); (ii) scales better or significantly better than all other SMC tools in cases where the `rf` equivalence is coarser than Mazurkiewicz; and (iii) outperforms each and every other tool/algorithm by an exponential factor in at least one program. Moreover, we show that the `rf` equivalence sometimes provides a significant performance advantage, both in identifying concurrency errors which are known to be very challenging—and often impossible—to detect by other methods, as well as when verifying that they have been fixed properly (Sect. 9).

In summary, the contributions of this paper include:

(1) A *novel SMC algorithm* for concurrent programs under the SC semantics (Sect. 4) that is both optimal for `rf` equivalence and efficient in practice.
(2) An *efficient polynomial-time test* (in the size of the program) for checking consistency of a given `rf` relation (Sect. 5), which in theory is incomplete, but in practice manages to correctly classify all potential `rf` relations arising in all programs we have tried.
(3) A *complete decision algorithm* for checking consistency (Sect. 5.3), which runs in polynomial time in the length of the program and exponential time in the number of threads.
(4) An *implementation* (Sect. 7) of all the above, which is publicly available both as an artifact [Abdulla et al. 2019] and in a possibly more up-to-date version as part of the Nidhugg tool.
(5) A presentation of use cases in which the `rf` equivalence, and our new algorithm, provides a significant performance advantage over other SMC techniques.

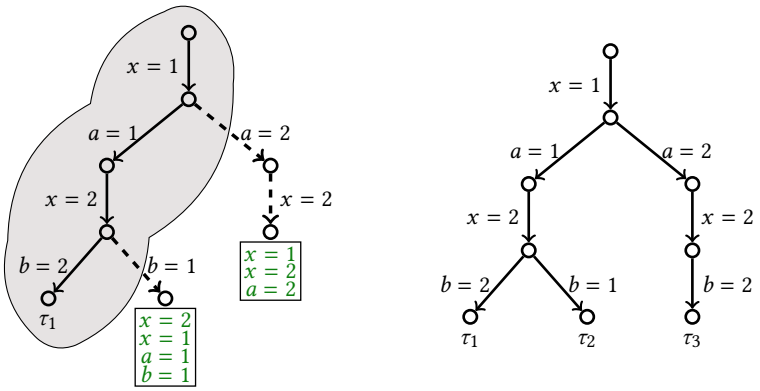

Fig. 2. Exploring the program of Fig. 1: (left) the first trace and two witnesses from it, (right) the tree of traces after the full exploration. We skip the initial write to x in traces here, and use green colour for witnesses.

2 ILLUSTRATING EXAMPLE

Let us illustrate the ideas of our SMC algorithm by its application to the program given in Fig. 1.

The algorithm represents each *execution* as a sequence of *events*, each of which is a particular execution of a program statement. We also regard the initialization of the shared variables (of x in this case) as events. In this program, the initialization event has no impact, so we omit it in the following presentation. Each execution thus consists of four events. For instance, the sequence $x{=}1\ a{=}x\ x{=}2\ b{=}x$ is an execution, shown as E_1 in Fig. 1. Since execution is under SC semantics, in which each read gets the value written by the most recent write to the same variable, this execution assigns the value 1 to a and the value 2 to b.

The goal of our SMC algorithm is to explore executions in such a way that each `rf` equivalence class (called a *trace*) is explored exactly once. Recall from the introduction that a trace is characterized by a sequence of events of each thread, and a relation `rf` ("reads-from") which connects a write with any read that reads its value. A trace is thus naturally represented as a partially ordered set of events, in which the events of each thread are totally ordered, events of different threads are unordered, and in which each read event is labeled by the write event from which it reads. Throughout the paper, we will represent a trace by any of its linearizations. Since such a linearization uniquely defines the represented trace, we will refer to it simply as a *trace*, and mean the (unique) trace that is represented by that sequence. Since in this program, each write event writes a unique value, we can denote a read event, such as $a = x$, which reads from $x = 1$, simply by $a = 1$. Thus, the trace containing E_1 and E_2 can be represented in six different ways, corresponding to its possible linearizations, e.g., as $x{=}1\ a{=}1\ x{=}2\ b{=}2$, or as $x{=}2\ b{=}2\ x{=}1\ a{=}1$, etc. Note that a read can also read from a write which succeeds it in the linearization; thus a possible representation of the trace containing E_3 is $x{=}1\ a{=}2\ x{=}2\ b{=}2$.

We can now view an execution as a trace linearization in which the events are ordered so that the source of each read is the most recent write to the same variable. We say that a trace τ is *consistent* if it has a linearization which is also an execution E. If so, we say that E is a *witness* for τ. A trace is inconsistent if it has no witness. An example of an inconsistent trace is $x{=}1\ x{=}2\ a{=}2\ b{=}1$. We say that a trace is *complete* if it cannot be extended, i.e., each thread reaches a terminal state.

To explore complete consistent traces, our SMC algorithm generates a tree, whose arcs are labeled by events of traces. Each node represents the trace represented by the sequence of events labeling the path from the root to that node. When the algorithm finishes, each complete consistent trace is represented by exactly one leaf. During the generation, the algorithm checks that generated

(complete or incomplete) traces are consistent by finding witnesses, which are then associated with the representing node. We now illustrate, using Fig. 2, how the exploration proceeds for the program in Fig. 1.

First, an arbitrary execution is generated. Let us assume that this is $E_1 := x=1\ a=1\ x=2\ b=2$. This execution also represents a complete trace τ_1, which becomes the leftmost path in the exploration tree in Fig. 2.

For each newly generated complete trace, the algorithm considers each of its read events, starting from the end in the generated linearization. It finds all possible write events in the trace, which can be alternative sources for this read event, while still maintaining consistency. Each such alternative source is entered into a set of so-called *schedules*, which is associated with the node from which the read is performed. Each schedule represents an alternative continuation, which must later be explored by the algorithm. These sets of schedules are analogous to the backtrack sets in the DPOR algorithm of Flanagan and Godefroid [2005] or the wakeup trees in the Optimal DPOR algorithm of Abdulla et al. [2017].

Let us illustrate this for the trace τ_1 in Fig. 2. Starting from the end, first the read $b = x$ is considered. A potential source is $x = 1$, which can be the source of $b = x$ precisely when the trace $x = 1\ a = 1\ x = 2\ b = 1$ is consistent. The algorithm performs a consistency check, which returns "consistent" and finds a witnessing execution (more details will be given in Sect. 5), say, $E_2 := x=2\ x=1\ a=1\ b=1$. (We use green colour for witnesses.) The algorithm then adds the read event $b=1$ together with this witness as a schedule that should be later explored from the node representing $x=1\ a=1\ x=2\ b=1$, denoted by the lower dashed arrow in Fig. 2.

Thereafter, the algorithm considers the read event $a=x$. A possible alternative source is $x=2$. Since this write appears after the corresponding read, the added schedule must include both this write event and the alternative read $a=2$. Before adding this schedule, the algorithm invokes the consistency check for $x=1\ a=2\ x=2$. In a similar way to the previous consistency check, the test succeeds and returns a witness, say $x=1\ x=2\ a=2$. The sequence of alternative write $x=2$ and alternative read $(a=2)$ together with this witness is entered as a possible schedule, denoted by the right-most path in the left tree of Fig. 2.

The search for alternative sources is now finished. Thereafter the algorithm extends each discovered schedule into a new complete consistent trace. In our example, the bottom schedule is already complete, so it becomes the second consistent trace generated by the algorithm, denoted τ_2. For the top one, the witness $x=1\ x=2\ a=2$ is extended to a complete trace by adding the read event $b=2$, resulting in execution $E_3 := x=1\ x=2\ a=2\ b=2$ and trace τ_3. For each new complete trace, the algorithm repeats the same steps that we just showed for τ_1. Here τ_2 does not offer any alternative sources that have not already been detected for τ_1. For τ_3, a possible alternative source for $b=2$ could be $x=1$. However, the consistency check using the polynomial-time test reveals that this is not possible, and so no new schedule is generated.

We can picture the explored traces in a tree, as in Fig. 2. Note that this tree is just a conceptual record of the explored traces, and is not stored in memory. At any point in time, the algorithm only needs to store the trace which is currently being explored, plus the schedules representing alternative reads of that trace. Thus, when the trace τ_3 is being explored, the left half of the tree (below $a=1$) need no longer be kept in memory. We also note that each consistency check in this example has successfully been handled by our polynomial-time test.

3 COMPUTATION MODEL

In this section, we introduce the class of program we consider in this paper, and then define several semantical notions such as transitions, events, executions and traces. We will use these notions later to define and reason about our SMC algorithm.

Programs. We consider a program $\mathbb{P}$ consisting of a finite set $\mathbb{T}$ of *threads* that share a finite set $\mathbb{X}$ of *(shared) variables*, ranging over a domain $\mathbb{V}$ of *values* that includes a special value 0. A thread has a finite set of local registers that store values from $\mathbb{V}$. Each thread runs a deterministic code, built in a standard way from expressions and atomic commands, using standard control flow constructs (sequential composition, selection and bounded loop constructs). Throughout the paper, we use x, y for shared variables, a, b, c for registers, and *expr* for expressions. *Global statements* are either a write $x = expr$ of the value of an expression *expr* to a shared variable or a read $a = x$ from a shared variable to a register *a. Local statements* only access and affect the local state of the thread and include assignments $a = expr$ to registers and conditional control flow constructs. Note that expressions do not contain shared variables, implying that a statement accesses at most one shared variable. In order to simplify the presentation, we do not consider atomic read-modify-write (RMW) operations. This is handled by our implementation (Sect. 7). The local state of a thread $t \in \mathbb{T}$ is defined as usual by its program counter and the contents of its registers.

Events, Traces, and Executions. Executions of a program $\mathbb{P}$ are defined according to the standard interleaving (a.k.a. sequential consistency) model. We formally define this model using the following pattern: after defining events and transitions, we define a run as a sequence of transitions of an individual thread; we define a trace as a composition of the events generated in a run of each thread, in which each read event reads from a write in the trace. An execution is then a linearization of a trace in which each read reads from the most recent write to the same variable.

First we define an event as a particular execution of a statement in an execution. A *write event* of $\mathbb{P}$ is a tuple $e = \langle id, t, \mathrm{W}, x, v\rangle$, where $id \in \mathbb{N}$ is an event identifier, $t \in \mathbb{T}$ is a thread, $x \in \mathbb{X}$ is a variable, and $v \in \mathbb{V}$ is a value. This event corresponds to thread t writing the value v to variable x. The identifier id denotes that t has executed $id{-}1$ events before e in the execution. For each variable $x \in \mathbb{X}$, we assume a special write event $\mathtt{init}_x = \langle -, -, \mathrm{W}, x, 0\rangle$, called the *initializer* event for x. This event is not performed by any of the threads in $\mathbb{T}$, and writes the value 0 to x. We define $\mathbb{E}_{init} := \{\mathtt{init}_x \mid x \in \mathbb{X}\}$ as the set of initializer events. A *read event* of $\mathbb{P}$ is a tuple $e = \langle id, t, \mathrm{R}, x, e'\rangle$, where id, t, and x are as for a write event, and e' is a write event of $\mathbb{P}$ or an initializer event for x. This event corresponds to thread t reading variable x and obtaining the value that was written by e'. That is, e' is of the form $\langle id', t, \mathrm{W}, x, v\rangle$ or $\langle -, -, \mathrm{W}, x, v\rangle$, causing e to read the value v. For a write event e of form $\langle id, t, \mathrm{W}, x, v\rangle$, we define $e.id := id$, $e.th := t$, $e.type := \mathrm{W}$, $e.var := x$, and $e.val := v$. For a read event e of form $\langle id, t, \mathrm{R}, x, e'\rangle$, we define $e.id$, $e.th$, and $e.var$ as for write events, and define $e.type := \mathrm{R}$ and $e.src := e'$.

A *transition* of a thread t is of form $\gamma \overset{e}{\rightsquigarrow} \gamma'$, where γ and γ' are local states of t and e is an event with $e.th = t$, which can be executed from the state γ, such that executing any subsequent local statements of t takes the thread t into local state γ'. A *run* of a thread t is a sequence of transitions of t of form $\gamma_0 \overset{e_1}{\rightsquigarrow} \gamma_1 \overset{e_2}{\rightsquigarrow} \cdots \overset{e_n}{\rightsquigarrow} \gamma_n$, where $e_i.id = i$ for $i = 1, \ldots, n$, i.e., event identifiers are increasing. The run is *complete* if there is no outgoing transition from γ_n. Note that in a run, we allow read events to read any value, not only values from write events in the same run (since they can read from a write event from another thread). The write events and the values of read events will be constrained (as they should be) when runs of the program's threads are combined to form traces and executions, to be defined below.

A *trace* τ of $\mathbb{P}$ is a set of events of $\mathbb{P}$, which consists of $\mathbb{E}_{init}$ and for each thread t the set of events in some run of t, such that for each read event $e \in \tau$, the write or initializer event $e.src$ is also in τ.

Given a trace τ, we let $\leq_\tau$ be the partial order on the events of τ, in which (i) the initializer events are ordered before the other events, (ii) for each thread $t \in \mathbb{T}$, the events of t are totally ordered by their event identifiers, and (iii) events of different threads are unordered, Note that the identifiers of the events of a thread t in τ form a contiguous increasing sequence starting from 1. A

linearization of a trace τ is a sequence obtained by ordering the events of τ in a way which respects $\leq_\tau$. For convenience, we use the term *trace* also for such a linearization, and mean the (unique) represented trace. For two sequences of events, π and π', we let $\pi \equiv \pi'$ denote that they contain the same sets of events. A trace is *complete* if for each thread t, the events of t come from a complete run of t.

Given a trace τ, we define the reads-from relation $\mathtt{rf}$ on its events by e $[\mathtt{rf}]$ e' if e' is a read event with $e'.src = e$, i.e., e' reads its value from e. We will use the term "happens-before" for the relation $[\leq_\tau \cup \mathtt{rf}]^+$, i.e., we will say that e *happens-before* e', or that e' *happens-after* e to denote that $e[\leq_\tau \cup \mathtt{rf}]^+e'$.

An *execution* E of $\mathbb{P}$ is a linearization of a trace such that for each read event e in E, the event $e.src$ is the last preceding write event to the same variable (note that $e.src$ can also be the initializer event for $e.var$). A trace τ may or may not have a linearization which is an execution. Define a trace τ to be *consistent* if it has a linearization E which is an execution. In this case, we say that E is a *consistency witness* (or just *witness*) for τ. Notice that a necessary (but not sufficient) condition for consistency of a trace is that the relation $[\leq_\tau \cup \mathtt{rf}]$, on the events of the trace, is acyclic.

Example 3.1 (Trace and execution). The rightmost trace τ_3 in Fig. 2 is consistent since it has a witness $E_3 := x{=}1\ x{=}2\ a{=}2\ b{=}2$, in which each read event reads from the last preceding write to the same variable. Note that in Fig. 2, the trace is represented by a linearization where the read event $a{=}2$ reads from the write $x = 2$, even though that write appears after the read. □

We say that executions E and E' are $\mathtt{rf}$ *equivalent* $E \equiv E'$, i.e., they contain the same sets of events. In two equivalent executions, each thread runs through the same sequence of local states, implying that they exhibit the same local assertion violations. Note however, that they need not end up with the same values of global variables.

Example 3.2 (Global variables in $\mathtt{rf}$ *equivalent executions).* In Fig. 1, the executions E_1 and E_2 result in different values of x. We have $x = 2$ and $x = 1$ at the end of E_1 and E_2 respectively. □

4 THE EXPLORATION ALGORITHM

In this section, we present our SMC algorithm, named ReadsFrom-SMC, which optimally explores complete consistent $\mathtt{rf}$ traces (*traces* for short) of a given concurrent program. In Sect. 6, we will show that it satisfies the following three properties:
(i) *Soundness*: each complete trace explored by the algorithm is a consistent trace of the program.
(ii) *Completeness*: the algorithm explores all consistent traces of the program.
(iii) *Optimality*: each trace is explored exactly once.
The exploration algorithm uses a novel test for checking whether a given trace is consistent, described in Sect. 5, thereby achieving "efficiency in practice". Recall that we represent each trace by some linearization.

Given a trace τ, we define a *cut* of τ to be a subsequence τ' of τ such that whenever e and e' are events in τ, such that τ' contains e and e' $[\leq_\tau \cup \mathtt{rf}]^*$ e, then τ' also contains e'. That is, a cut is closed under causal dependencies (in the sense of $\leq_\tau \cup \mathtt{rf}$). Note that a cut is also a trace.

For a trace τ and an event $e \in \tau$ let

$pre(\tau, e)$ denote the prefix of τ up to, but not including, e;
$post(\tau, e)$ denote the suffix of τ after, but not including, e;
$predecs(\tau, e)$ denote the minimal cut of τ which contains e, i.e., the set of events (including e) on which e is causally dependent.

As an example, if τ is the execution E_4 in Fig. 1, i.e., $\tau := x{=}2\ x{=}1\ a{=}1\ b{=}1$, then $predecs(\tau, b{=}1)$ is $x{=}2\ x{=}1\ b{=}1$.

Algorithm 1: ReadsFrom-SMC.

```
1  ReadsFrom-SMC(τ, E)
2      extend E to a complete execution E · Ê where each event of Ê is unmarked
3      τ' := τ · Ê
4      for each read event e_R ∈ Ê do schedules(pre(τ', e_R)) := ∅
5      for each e_R, e_W ∈ τ' : e_W.var = e_R.var and e_W ≠ e_R.src and
6                               (e_R ∈ Ê or e_W ∈ Ê) and unmarked(e_R) and MayRead(τ', e_R, e_W) do
7          τ'' := pre(τ', e_R)
8          π := predecs(τ', e_W) ∩ post(τ', e_R)
9          σ := e_R[src := e_W] · mark(π)
10         E'' := GetWitness(τ'' · σ, E · Ê)
11         if E'' ≠ ⟨⟩ and ¬∃⟨σ', −⟩ ∈ schedules(τ'') : σ' ≡ σ then add ⟨σ, E''⟩ to schedules(τ'')
12     for each read event e_R ∈ Ê starting from the end do
13         τ'' := pre(τ', e_R)
14         for each ⟨σ, E''⟩ ∈ schedules(τ'') do ReadsFrom-SMC(τ'' · σ, E'')
15         erase schedules(τ'')
```

The pseudocode of ReadsFrom-SMC is given as Algorithm 1. The algorithm consists in a call ReadsFrom-SMC$(\langle\rangle, \langle\rangle)$ to the recursive procedure ReadsFrom-SMC(τ, E), where τ is a trace and E is an execution. A precondition for ReadsFrom-SMC(τ, E) is that τ is consistent and E is a witness for τ. If the precondition is satisfied, a call to ReadsFrom-SMC(τ, E) will explore all complete consistent traces that extend τ. More precisely, for each complete consistent trace τ' that extends τ (i.e., τ is a cut of τ'), a witness $E' = E \cdot \hat{E}$ of τ' will be generated at line 2 of some subcall of ReadsFrom-SMC(τ, E) in such a way that each trace (i.e., `rf` equivalence class) is explored exactly once.

An important element in ReadsFrom-SMC is to analyze an explored trace to see whether another trace can be formed by changing the source of one of its read events. In order to avoid redundant such analyses, the representation of events in traces that arise in the algorithm is extended with an additional field whose value is either $\top$ or $\bot$. In the first case, we say that the event is *marked*; in the second, that it is *unmarked*. Whenever a new read event is generated to explore a new trace, it is unmarked, denoting that the algorithm should look for alternative sources for the read. When an unmarked event is included in a schedule (at line 9), it becomes marked to denote that it is merely a copy of an unmarked event, and that searching for alternative sources is redundant. For a sequence π of events, let $mark(\pi)$ be the sequence π but with each event marked. We adapt the equivalence $\equiv$ on sequences of events, by letting it ignore whether events are marked, so that, e.g., $mark(\pi) \equiv \pi$.

When analyzing a trace τ', then for each prefix τ'' of τ' such that $\tau'' = pre(\tau', e_R)$ for some unmarked read event e_R in τ', the algorithm maintains a set $schedules(\tau'')$ that records discovered alternative sources for read events. Each element of $schedules(\tau'')$ is a pair $\langle\sigma, E\rangle$, where σ is a sequence of events, the first of which is obtained from e_R by changing its source to another event in τ'' or in σ, and E is a witness for $\tau'' \cdot \sigma$. Each sequence σ is a way to extend τ'' to a trace which is incompatible with τ', since the source of e_R has been changed. Each such σ will subsequently be explored as a continuation of τ'' in a call to ReadsFrom-SMC which happens in the recursive exploration phase; see below. Thus, the sets $schedules(\tau'')$ play a role analogous to that of the backtrack sets in the DPOR algorithm of Flanagan and Godefroid [2005] or the wakeup trees in the

Optimal DPOR algorithm of Abdulla et al. [2017]. Each set $schedules(\tau'')$ is maintained across calls to ReadsFrom-SMC as long as τ'' is a prefix of the currently explored trace.

A recursive call to ReadsFrom-SMC consists of three phases: the exploration phase (lines 2–3), the new-source-detection phase (lines 4–11), and the recursive-exploration phase (lines 12–15).

In the exploration phase, ReadsFrom-SMC(τ, E) extends E to an arbitrary complete execution $E \cdot \hat{E}$ with all events of $\hat{E}$ unmarked. Correctness properties can here be checked on $E \cdot \hat{E}$. Thereafter, τ is extended correspondingly to $\tau' = \tau \cdot \hat{E}$. Note that τ' is consistent, since it has a witness $E \cdot \hat{E}$.

In the new-source-detection phase, the algorithm searches for all possible ways to change the source $e_R.src$ of a read event e_R in τ' to a write event e_W in τ' which is different from $e_R.src$. First, the sets $schedules(\tau'')$ for traces τ'' that are generated for the first time are initialized (at line 4). Thereafter, all possible alternative sources for any unmarked read event e_R of τ' are found. It is sufficient to consider alternative sources e_W that satisfy the conditions of line 6: (i) either e_R or e_W is in $\hat{E}$ (otherwise their combination has been explored in some previous call), (ii) e_R is unmarked (otherwise alternative sources for e_R are investigated elsewhere), and (iii) e_W passes the test MayRead(τ', e_R, e_W), which returns true if $e_W \in \tau'$ is such that $e_R[\leq_\tau \cup \texttt{rf}]^+ e_W$ does not hold (i.e., e_W does not happen-after e_R), and such that there is no write event e'_W with $e_W[\leq_\tau]^+ e'_W[\leq_\tau \cup \texttt{rf}]^+ e_R$ (i.e., e'_W is a later write to the same variable, by the same thread as e_W or with $e_W \in \mathbb{E}_{init}$, which anyway happens-before e_R). It is not difficult to see that these conditions are necessary—but in general not sufficient—for any alternative source of e_R. The test MayRead can be viewed as a low-cost filter which rejects combinations of reads and writes that are trivially inconsistent, in order to reduce the number of calls to GetWitness.

If e_R and e_W pass all these necessary checks, the algorithm constructs a new trace. It first lets τ'' be the events in τ' which precede e_R, then lets π be the sequence of events that succeed e_R in τ' and happen-before e_W (at line 8; note that the intersection is well-defined since both arguments are subsequences of τ'). The algorithm then constructs σ by appending π to $e_R[src := e_W]$ (the read event obtained from e_R by changing its source to e_W), and marking its events. It thereafter checks whether the trace $\tau'' \cdot \sigma$ is consistent by asking GetWitness$(\tau'' \cdot \sigma, E \cdot \hat{E})$ to generate a witness for it (to be described in Sect. 5). If GetWitness finds such witness E'' (signalled by returning a non-empty sequence) the algorithm has discovered that σ is a way to extend τ'' to a consistent trace. Therefore, the pair $\langle \sigma, E'' \rangle$ is inserted into $schedules(\tau'')$ (unless a sequence σ' equivalent to σ, i.e., with the same set of events, is already there), so that a call to ReadsFrom-SMC$(\tau'' \cdot \sigma, E'')$ will be performed later.

Finally, in the recursive-exploration phase, the algorithm considers all read events e_R in $\hat{E}$, in the reverse order of their appearance in $\hat{E}$. For each pair $\langle \sigma, E'' \rangle$ in $schedules(pre(\tau', e_R))$, it performs a new call ReadsFrom-SMC$(pre(\tau', e_R) \cdot \sigma, E'')$ to explore the complete consistent traces that extend $pre(\tau', e_R) \cdot \sigma$. It is important to consider the read events in $\hat{E}$ in reverse order of appearance, i.e., to explore traces in a depth-first manner, since calls that explore alternative sources of later read events can lead to the addition of new schedules for earlier read events.

Example 4.1 (ReadsFrom-SMC). Figure 2 illustrates the exploration tree corresponding to an application of ReadsFrom-SMC on the program in Fig. 1. We use the same simplified notation for events, executions, and witnesses as in Sect. 2.

First, the algorithm starts with an empty trace and an empty execution. In the exploration phase (lines 2 and 3), it generates the first execution E_1 and trace τ_1 where $E_1 = \tau_1 := x{=}1\ a{=}1\ x{=}2\ b{=}2$.

Then, the new-source-detection phase (lines 4–11) is applied to τ_1, searching for new ways to change the sources of two unmarked read events $b{=}2$ and $a{=}1$. For $b{=}2$, MayRead$(\tau_1, b = 2, x = 1)$ returns true because the write $x = 1$ does not happen-after the read $b = 2$. At line 7, a prefix $\tau'' := x = 1\ a = 1\ x = 2$ of τ_1 is collected that contains all events preceding $b = 2$ in τ_1. Because

$x = 1$ is the first event of thread t_2, the sequence π of all events in τ_1 that succeed $b = 2$ and causally precede $x = 1$ is empty, i.e., $\pi = \langle\rangle$. Therefore, at line 9 the algorithm builds $\sigma := b = 1$. Furthermore, it generates a witness $E_2 := x=2\ x=1\ a=1\ b=1$ for the trace $\tau_2 := \tau'' \cdot \sigma$ by the procedure GetWitness(τ_2, E_1). Finally, the pair $\langle\sigma, E_2\rangle$ is inserted into *schedules*(τ'') to obtain the lower dashed branch in Fig. 2. In an analogous way, the algorithm can construct a witness for the read event $a = 1$ and its possible source $x = 2$ in τ_1.

Finally, in the recursive-exploration phase (lines 12–15), the algorithm extends each generated witness into a new complete consistent trace as explained in Sect. 2. □

5 CHECKING CONSISTENCY: THE GETWITNESS PROCEDURE

In this section, we describe the realization of the procedure GetWitness(τ, E), which generates a witness for the trace τ if τ is consistent, and otherwise returns $\langle\rangle$. The execution E acts as a guidance for ordering write events to the same variable when generating the witness. For this to be meaningful, a precondition for GetWitness(τ, E) is that τ is obtained from a cut of E by changing the source of one read event. Note that this precondition is satisfied by each call on line 10 of ReadsFrom-SMC.

The procedure GetWitness(τ, E) is structured into three phases:

P-I. **(Saturation)** In this phase, the rf and $\leq_\tau$ relations of τ are extended, using two saturation rules, to a *saturated-happens-before* relation shb, which extends $\leq_\tau \cup$ rf by orderings that must be respected by any witness of τ. If shb is cyclic, then τ is inconsistent and GetWitness returns $\langle\rangle$, otherwise it moves to the next phase.

P-II. **(Witness Construction)** In this phase, GetWitness(τ, E) attempts to construct a witness for τ by extending shb to a relation cohb which for each variable totally orders the write events to that variable. Whenever possible, the relation cohb orders writes in the same way as in E, i.e., E is used as guidance for the witness construction. If the phase succeeds in constructing cohb without introducing cycles, a topological sort of τ which respects cohb will generate a witness, which is returned by GetWitness(τ, E). If the phase does not succeed, GetWitness(τ, E) moves to the next phase.

P-III. **(Decision Procedure)** In this phase, GetWitness(τ, E) employs a decision procedure, called ConsistencyDecision(τ), described in Section 5.3, which checks whether τ is consistent. This decision procedure runs in polynomial time in the length of the program and exponential time in the number of threads. If τ is consistent, GetWitness(τ, E) returns a witness for it.

We again remark that no consistency check has ever reached phase P-III on any program that we have tried so far. This furthermore implies that whenever the saturation of phase P-I did not find a cycle, then the trace was consistent. We next describe these three phases in more detail.

5.1 Phase P-I: Saturation

The procedure Saturate(R) extends any ordering relation R on the events of a trace τ by additional orderings, which must be respected by any witness for τ that respects R. In phase P-I, the saturation procedure is used to extend the relation $\leq_\tau \cup$ rf. In phase P-II, it extends relations that are obtained by successive strengthenings of the ordering of events.

We define Saturate(R) := shb, where shb, called *saturated-happens-before relation*, is the smallest transitive relation on the events in τ which includes R, and which for any combination of a read event e_R and different write events e_W, e'_W which are all to the same variable, satisfies the following two rules:

R1. If e_W [shb] e_R and e'_W [rf] e_R, then e_W [shb] e'_W.
R2. If e'_W [shb] e_W and e'_W [rf] e_R, then e_R [shb] e_W.

Let us motivate these rules. Rule R1 says that whenever e_W happens-before e_R, and e_R reads from e'_w, then e_W must happen-before e'_W; see Fig. 3. This is necessary, since e'_W [rf] e_R implies that e'_W must be the last write to x that precedes e_R; since e_W [shb] e_R we must then have e_W [shb] e'_W. Rule R2 says that whenever e'_W happens-before e_W, and e_R reads from e'_w, then e_R must happen-before e_W; again see Fig. 3. This is necessary, since e'_W [rf] e_R implies that e'_W must be the last write to x that precedes e_R; since e'_W [shb] e_W we must then have e_R [shb] e_W.

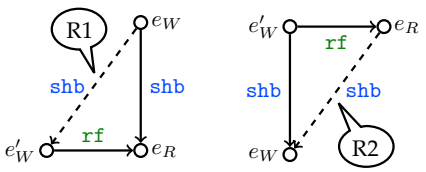

Fig. 3. Illustrating the saturation procedure. The dashed arrows are edges added by the procedure.

The saturation procedure can be performed in time polynomial in the length of the trace τ. This saturation procedure is in part inspired by the saturation procedure used by Abdulla et al. [2018] for the Release-Acquire (RA) semantics. There are two main differences between the two procedures:

- Rule R1 is a weakening of the saturation rule for RA. More precisely, in the case of RA, the rule is of the form: If e_W $[\leq_\tau \cup \texttt{rf}]^+$ e_R and e'_W [rf] e_R, then e_W [shb] e'_W. In other words, the definition of shb is not applied recursively as in the case of SC. This also affects the manner in which saturation can be implemented in the case of RA, since a trace can be saturated by making a single pass over the set of events in the trace. Such an approach is not possible under SC due the recursive of definition of the shb relation.
- Rule R2 is not needed for RA.

Example 5.1 (Saturation). We illustrate how Saturate($\leq_{\tau_2} \cup$ rf) is computed on the trace τ_2 (i.e., the trace to the right of τ_1 in Fig. 2). The trace τ_2 contains four events: $x = 1$, $x = 2$, $a = 1$, and $b = 1$. The relations $\leq_{\tau_2}$ and rf are as shown in Fig. 4. The relation shb includes $\leq_{\tau_2}$ and rf, and additionally the dashed arrow, which is obtained by applying the rule R1. We note that the resulting relation shb is acyclic. This does not in general allow to conclude that τ is consistent; see Example 5.2 below. However, in this particular case, shb totally orders all the writes to x. Therefore, by Lemma 5.3, τ is consistent. In the general case when shb does not totally order all the writes to each variable, phase P-II will try to extend shb to an ordering with this property. □

Incompleteness of the Saturation Procedure. The saturation procedure Saturate($po \cup$ rf) is in general incomplete in the sense that there may exist orders between events that are implied by the SC semantics but that are not detected by the procedure. Incompleteness implies that we may have saturated traces that do contain cycles but that nevertheless violate the SC semantics. In other words, passing the test performed by the saturation procedure is necessary but not sufficient for checking consistency of a given trace. Therefore, we may need to run the extra decision procedure of Phase P-III in order to ensure consistency. In contrast, the saturation procedure for the RA semantics, as described by Abdulla et al. [2018], is sound and complete, and hence no extra phases are required for checking consistency in the case of RA.

Despite the general incompleteness of the saturation procedure, it succeeded in spotting all inconsistent traces in all the examples that we have tried. We believe that inconsistent traces which are accepted by the test arise only for artificial programs which are specifically designed to exhibit certain complex interaction patterns that do not appear in "normal" applications. To illustrate this, we show the simplest such an example that we have been able to construct.

Example 5.2 (The saturation procedure is incomplete). We give an example that illustrates the incompleteness of the saturation procedure. The example consists of two steps. The first step shows that the saturation procedure is incomplete, i.e., there may be orders on the events that are implied

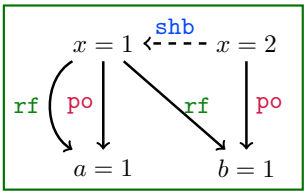

Fig. 4. Applying the saturation procedure to the trace τ_2 in Fig. 2. The dashed arrow is an added edge.

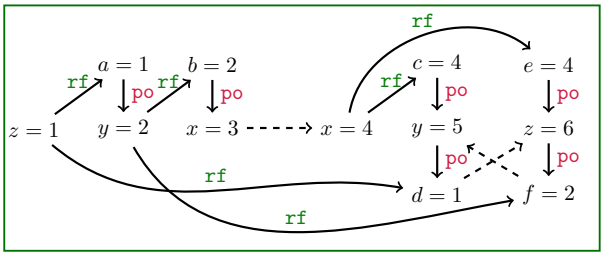

Fig. 5. Saturation is insufficient to generate all necessary orders between events. Here x, y, and z are shared variables while a, b, c, d, e, and f are registers of read events.

by the SC semantics but not detected by the procedure. The second step shows that there are saturated traces that do not contain cycles but that nonetheless violate the SC semantics.

In the first step, let us consider the trace τ given in Fig. 5. The trace uses three shared variables x, y, and z, and contains six write events: $x=3$, $x=4$, $y=2$, $y=5$, $z=1$, and $z=6$. Since all the write events use different values, we will represent a read event by the value it reads. For instance, we write $a=1$ to denote the execution of the read statement $a=z$ with source $z=1$. Then, the trace τ contains six read events: $a=1$, $b=2$, $c=4$, $d=1$, $e=4$, and $f=2$. Figure 5 shows the relations $\leq_\tau$ and $\mathtt{rf}$ between the events. Here, the saturation procedure does not extend the relation $\leq_\tau \cup \mathtt{rf}$ with any more edges, i.e., $\mathtt{shb} \equiv (\leq_\tau \cup \mathtt{rf})$. However, there is a hidden order between the events $x=4$ and $x=3$ that is respected by any linearization of τ which is an execution, but that is not generated by the saturation procedure on τ. To see this, we will explain why ordering these two events in the reverse direction would lead to a violation of the SC semantics. Consider any linearization E of τ such that E is an execution, and such that E orders $x=3$ before $x=4$, as indicated by the dashed arrow from $x=3$ to $x=4$. Since $z=1$ happens-before $x=3$ we know that E orders $z=1$ before $x=3$. Similarly, $x=4$ happens-before $z=6$ and hence E orders $x=4$ before $z=6$. Since $x=3$ is ordered before $x=4$ (by assumption), we have that $z=1$ is ordered before $z=6$. Since E orders $z=1$ before $z=6$, $z=1$ $[\mathtt{rf}]$ $d=1$, and E is an execution, it follows that E orders $d=1$ before $z=6$ (as indicted by the dashed arrow). In a similar way, E orders $f=2$ before $y=5$. Finally, since $y=5$ $[\leq_\tau]$ $d=1$ and $z=6$ $[\leq_\tau]$ $f=2$, E orders $y=5$ before $d=1$, and $z=6$ before $f=2$. This means that E contains a cycle which contradicts the fact that E is a linearization of τ.

In the second step, we extend τ to τ' by adding more events (following a similar pattern in τ) that induce another hidden order, now from $x=3$ to $x=4$, that is also not generated by the saturation procedure on τ'. This means that any linearization of τ' which is an execution must order both $x=3$ before $x=4$, and $x=4$ before $x=3$, which means that τ' is inconsistent. However, the saturation does not realize any order between the two write events $x=3$ and $x=4$, and hence it cannot figure out that τ' is inconsistent. □

5.2 Phase P-II: Witness Construction

If the saturated-happens-before relation shb constructed in phase P-I is acyclic, then phase P-II attempts to show that τ is consistent by constructing a witness for τ. The key problem is to find an extension cohb of shb which for each variable totally orders the write events to that variable (an ordering between writes to the same variable is called a *coherence order* in the literature). If such an extension is acyclic, τ is consistent by Lemma 5.3, and a witness can be obtained by a topological sort. Phase P-II makes an attempt to generate an ordering between writes by mimicking the ordering of the corresponding writes in the guiding witness E, which is supplied as an argument

Algorithm 2: ConstructWitness.

```
1  ConstructWitness(τ, E)
2      shb := Saturate(≤_τ ∪ rf) where ≤_τ and rf are extracted from τ
3      cohb := shb
4      for write events e_W, e'_W ∈ τ : e_W.var = e'_W.var and (e_W, e'_W) ∉ cohb and (e'_W, e_W) ∉ cohb do
5          if (e_W, e'_W) ∈ E then cohb := cohb ∪ (e_W, e'_W)
6          else cohb := cohb ∪ (e'_W, e_W)
7          cohb := Saturate(cohb)
8          if cohb is cyclic then return "unknown"
9      E' := TopologicalSort(cohb)
10     return ⟨"consistent", E'⟩
```

to GetWitness. The intuition is that E is a witness for a trace, which differs from τ by a single $\texttt{rf}$ relation, and therefore there ought to be a witness for τ which is quite similar to E, at least in how it orders writes to the same variable.

A pseudocode of phase P-II is given as the procedure ConstructWitness(τ, E) in Algorithm 2. First, the procedure assigns the ordering produced by the saturation procedure in phase P-I to shb. Then, the construction starts by letting cohb be shb. It uses the given execution E as guidance for ordering write events to the same variable. As long as there exist two write events e_W and e'_W to the same variable which are not ordered by cohb, it performs the following steps (lines 4–8): (i) let cohb order the writes in same way as in E, (ii) extend cohb by the Saturate procedure, and (iii) if cohb becomes cyclic, abort the witness construction and return "unknown". If, after completing this loop, cohb orders all pairs of write events to the same variable and no cycle arose in the loop, a topological sort of τ which respects cohb will then generate a witness E' for τ (line 9). In this case, the test returns "consistent" and also the witness E'.

Lemma 5.3. *Let τ be a trace, and let* cohb *be a transitive ordering on the events of τ which extends $\leq_\tau \cup \texttt{rf}$ and respects the rule R1 and the rule R2 of the saturation procedure. If* cohb *totally orders any two writes to the same variable and is acyclic, then τ is consistent and any topological sort (i.e., linearization) of τ which respects* cohb *will produce a witness for τ.*

Proof. The nontrivial property to establish is that any topological sort of τ which respects cohb results in an execution where each read reads the most recent write to the same variable. This follows by noting that whenever e_W $[\texttt{rf}]$ e_R, and e'_W is another write to the same variable, then either e'_W is before e_W by the rule R1 or after e_R by the rule R2 . □

The following lemma shows that the test ConstructWitness(τ, E) spends polynomial time to construct a witness for τ using the guidance from the execution E.

Lemma 5.4. *For a trace τ and an execution E, the ConstructWitness(τ, E) test is polynomial time.*

5.3 Phase P-III: Decision Procedure

If neither of the previous two phases succeeded in determining whether a trace is consistent, then phase P-III settles this question by a decision procedure, called ConsistencyDecision, whose running time is polynomial in the length of the program and exponential in the number of threads. For a given trace τ, if τ is consistent then the procedure returns a witness for τ, otherwise it returns $\langle\rangle$. The idea of ConsistencyDecision(τ) is first to construct a graph representing the possible configurations and transitions between them in any execution which generates a witness of τ.

Algorithm 3: ConsistencyDecision.

Initial call: ConsistencyDecision(τ)

```
1  ConsistencyDecision(τ)
2      G := BuildGraph(τ)
3      let v_init ∈ G : ∀t ∈ 𝕋. v_init(t).id = 1
4      let v_target ∈ G : ∀t ∈ 𝕋. v_target(t).type = term
5      if ∃ path P in G from v_init to v_target then return ⟨a witness E for τ from P⟩
6      else return ⟨⟩
7  BuildGraph(τ)
8      V := {v : 𝕋 → τ^term | (v(t) = e) → (e.th = t)}
9      E := ∅
10     for v1, v2 ∈ V do
11         if ∃t ∈ 𝕋. v2(t).id = v1(t).id+1 and (∀t' ∈ 𝕋 \ {t} . v1(t') = v2(t')) then
12             let e be v1(t) and let x be e.var
13             if e.type = W then
14                 if ∃e_R ∈ τ. e_R.var = x and e_R.id ≥ v1(e_R.th).id and
15                                 (init_x [rf] e_R or (∃e_W ∈ τ. e_W [rf] e_R and e_W.id < v1(e_W.th).id)) then
16                     isSC := false
17                 else isSC := true
18             else if e.type = R then
19                 let e_W be the event such that e_W [rf] e
20                 isSC := (e_W = init_x or e_W.id < v1(e_W.th).id)
21             if isSC then E := E ∪ {⟨v1, v2⟩}
22     return ⟨V, E⟩
```

Thereafter, consistency is determined by checking whether this graph has a path from the initial to the final configuration. By construction, such a path corresponds to a witness for τ.

The pseudocode of ConsistencyDecision(τ) is given as Algorithm 3. The graph is constructed by the procedure BuildGraph(τ). Let $\mathbb{T}$ be the set of threads that occur in the events in τ. Let $\tau^{\texttt{term}}$ be τ extended by one extra event for each thread t, denoted $\texttt{term}^t$, which intuitively indicates that t has terminated (i.e., its last event in τ has been executed). Formally, $\texttt{term}^t$ is of form $\langle id + 1, t, \texttt{term}, -, - \rangle$ where id is the largest identifier of an event $e \in \tau$ with $e.th = t$. We define $\texttt{term}^t.id := id + 1$, $\texttt{term}^t.th := t$, and $\texttt{term}^t.type := \texttt{term}$.

The procedure BuildGraph(τ) builds a graph G at line 2. Let us present G in the form of $\langle V, E \rangle$. A vertex v in V maps each thread in $\mathbb{T}$ to an event in $\tau^{\texttt{term}}$ of that thread at line 8. Intuitively, v represents the configuration where for each thread t, the event $v(t)$ is the next one to be executed by t (if $v(t) = \texttt{term}^t$ then thread t has terminated). The set E is built by the for-loop at line 10. For each pair of vertices v_1 and v_2, an edge should be added from v_1 to v_2 if it is possible for a thread to perform its next event in v_1 according to the SC semantics and arrive at v_2. This is possible under the following two conditions: (i) v_2 must be a "successor" of v_1 in the sense that v_2 results from one thread performing an event in v_1, and (ii) the execution of that event satisfies the SC semantics.

Condition (i) is checked by the if-statement at line 11. It checks whether there is a thread t, which performs an event, i.e., the identifier of the event $v_2(t)$ is one more than the identifier of $v_1(t)$, and checks that all other threads have not moved, i.e., they have identical events in v_1 and v_2.

Condition (ii) is checked using the Boolean flag *isSC*, which is true if and only if the transition corresponding to the edge from v_1 to v_2, in which thread t performs e, satisfies SC semantics. For

this, it is necessary and sufficient to check that whenever a read event e_R on a variable x reads from a write event e_W, i.e., e_W [rf] e, then (a) e_R is performed after e_W, and (b) no other write e'_W to x is performed after e_W and before e_R. Whenever e is a write event, we check that e does not correspond to the event e'_W in condition (b), i.e., that there is no later read e_R which reads from a write e_W performed before e, which is overwritten by e. Note that a special case is when e_W is the initializer event $\texttt{init}_x$ for x. Whenever e is a read event, we check condition (a), i.e., that the write from which it reads has already been performed.

Once the graph G has been built, we check whether there exists a path P from the initial vertex v_{init}, in which no thread has executed any event, to the target vertex v_{target}, in which all threads have terminated at line 5.

The correctness of the ConsistencyDecision procedure follows from the fact that a path from v_{init} to v_{target} represents a linearization of the set events resulting in an equivalent execution which shows the consistency of τ. The fact that this execution respects SC semantics follows from the rules for adding edges in the for-loop at line 10. On the other hand, a linearization of the set events corresponding to an execution E defines a path from v_{init} to v_{target} where the events occur in the same order as in E, and each event corresponds to an added edge in the graph.

The time complexity of ConsistencyDecision follows from observing that the size of the constructed graph is polynomial in the length of the program and exponential in the number of threads, and that reachability can be checked in polynomial time in the size of the graph.

6 PROPERTIES OF THE SMC ALGORITHM

In this section, we show that the ReadsFrom-SMC algorithm is sound, complete, and optimal.

Theorem 6.1. *The algorithm ReadsFrom-SMC has the following properties:*

- Soundness: *each complete trace explored by ReadsFrom-SMC is a consistent trace of the program.*
- Completeness: *ReadsFrom-SMC explores each consistent complete trace of the program,*
- Optimality: *ReadsFrom-SMC never explores a complete trace more than once.*

To prove the correctness properties of Algorithm 1, formulated in the above theorem, we first observe that Soundness and Optimality are easy to establish:

- Soundness follows from the observation that each trace that is explored at line 3 has a witness, and is hence consistent.
- Optimality follows by observing that the exploration tree branches only for a read event e_R, and that in each branch, e_R reads from a different source. Thus, the same trace can never appear in in two different branches.

Completeness requires more efforts to establish. It follows from Theorem 6.4 below, for which we first need a couple of lemmas.

Lemma 6.2. *If a trace τ is consistent, then any trace τ' with $\tau' \subseteq \tau$ is consistent.*

The proof of Lemma 6.2 is obvious since $\tau' \subseteq \tau$ means that τ' is a cut of τ.

We will also need the following lemma, which states that if a trace contains a read e_R which reads from a write e_W, and e_W does not happen-before any other events in the trace τ', then the source of e_R can be changed to some other write event, and still preserve consistency. Define an event e to be *enabled* after a trace τ if, letting π denote the subsequence of events of $e.th$ in τ, the sequence $\pi \cdot e$ is the sequence of events in some run of $e.th$. That is, e is the next event to be performed by $e.th$ after τ. In the following lemma, we have adapted the notation to fit its application in the proof of Theorem 6.4.

Lemma 6.3. *Let τ'' be a consistent trace, and let e_R be a read event which is enabled after τ''. Let π be a sequence of events which contains a write event e'_W, such that (i) π does not contain any event by thread $e_R.th$, (ii) $\tau'' \cdot e_R[src := e'_W] \cdot \pi$ is a consistent trace, and (iii) e'_W does not happen-before any event in $\tau'' \cdot \pi$. Then there is a write event e''_W in $\tau'' \cdot \pi$, different from e'_W, such that $\tau'' \cdot e_R[src := e''_W] \cdot (\pi \setminus e'_W)$ is a consistent trace.*

Proof. Since $\tau'' \cdot e_R[src := e'_W] \cdot \pi$ is consistent, it has a witness E. Since neither e'_W nor e_R happens-before any event in $\tau'' \cdot \pi$, we can reorder E so that it is of form $E' \cdot e'_W \cdot e_R[src := e'_W]$ and is still a witness. This implies that there is a write event e''_W in E' such that $E' \cdot e_R[src := e''_W]$ is a witness. Thus, we have found a sought e''_W and $\tau'' \cdot e_R[src := e''_W] \cdot (\pi \setminus e'_W)$ is a consistent trace since it has a witness E'. □

We can now state the main theorem of completeness.

Theorem 6.4. *In each call to ReadsFrom-SMC(τ, E), each trace $\hat{\tau}$ which is a consistent complete extension of τ is explored at least once.*

Note that ReadsFrom-SMC(τ, E) is called only if τ is consistent. The main Theorem 6.1 follows from Theorem 6.4 by noting that ReadsFrom-SMC consists of the call ReadsFrom-SMC$(\langle\rangle, \langle\rangle)$.

Proof. Theorem 6.4 is proven by reverse induction on the length of τ. The base case, in which τ is a complete trace, is trivially established. For the inductive step, consider a call ReadsFrom-SMC(τ, E) for an arbitrary consistent trace τ. We perform a proof by contradiction: we assume that there is a complete and consistent extension $\hat{\tau}$ of τ which is not explored by ReadsFrom-SMC(τ, E). We let τ' be the longest prefix of $\hat{\tau}$ such that a call of form ReadsFrom-SMC$(\tau', -)$ is actually performed. Let τ'' be a longest prefix of any complete trace explored in that call whose events are contained in $\hat{\tau}$. Note that τ'' is consistent, but not complete, since then it would be equivalent to $\hat{\tau}$, which by assumption is not explored. Let E'' be such that $\tau'' \cdot E''$ is a complete trace explored by the algorithm. Since write events are completely determined by their happens-before predecessors, the first event in E'' must be a read event e_R. This means that e_R is not in $\hat{\tau}$. But, since e_R is enabled after τ'', an event of form $e_R[src := e_W]$ for some e_W in $\hat{\tau}$ must be contained in $\hat{\tau}$. Moreover, the event e_R must be unmarked in $\tau'' \cdot E''$. To see this, note that a read event e'_R becomes marked only as part of the sequence π at line 9 of ReadsFrom-SMC(τ, E), and only if it happens-before some read event that precedes it in the explored sequence. Since $\hat{\tau}$ does not contain e_R, the sequence τ'' that precedes e_R also cannot contain any event that happens-after e_R, whence e_R must be unmarked. Let $\pi'' = predecs(\hat{\tau}, e_W) \cap post(\hat{\tau}, e_R[src := e_W])$.

We claim that $e_R[src := e_W] \cdot \pi'''$ will be added to $schedules(\tau'')$ for some π''' with $\pi''' \equiv \pi''$. To prove this claim, we prove the following stronger claim.

Claim 1. *For all write events e'_W and sequences π such that $e'_W \in \tau'' \cdot \pi$ and such that*
(1) all events in π (except e'_W) happen-before e'_W, and
(2) $\tau'' \cdot e_R[src := e'_W] \cdot \pi$ is a consistent trace,
a sequence of form $e_R[src := e'_W] \cdot \pi'''$ for some π''' with $\pi''' \equiv \pi$ will be added to $schedules(\tau'')$.

Recall that τ'' is a consistent trace, which, among other things, implies that no event in π happens-before some event in τ''.

We prove Claim 1 by induction on the size of π. For $\pi = \langle\rangle$, the claim follows immediately, since then e'_W must be in τ''. For the inductive step, assume an arbitrary e'_W and π. Note that if $\pi \neq \langle\rangle$ then $e'_W \notin \tau''$ since τ'' is a trace. Then, the preconditions of Lemma 6.3 are satisfied. To see this, note that condition (i) follows from conditions (1) and (2) in the claim, that condition (ii) follows from condition (2), and that condition (iii) follows from condition (1) in the claim. The conclusion of the

lemma implies that there is a e''_W, different from e'_W, in $\tau'' \cdot \pi$, such that $\tau'' \cdot e_R[src := e''_W] \cdot (\pi \setminus e'_W)$ is a consistent trace. Let $\pi' = predecs(\pi, e''_W)$. It can now be checked that e''_W and π' satisfy the preconditions of Claim 1. Since π' is smaller than π, a sequence equivalent to $e_R[src := e''_W] \cdot \pi'$ with π' marked must be added to $schedules(\tau'')$, by the inductive hypothesis. This means that a call of form ReadsFrom-SMC$(\tau'' \cdot e_R[src := e''_W] \cdot \pi', -)$ will be performed. By the inductive hypothesis for Theorem 6.4, it will explore each consistent extension of $\tau'' \cdot e_R[src := e''_W] \cdot \pi'$. In particular, it will explore a trace which extends $\tau'' \cdot e_R[src := e''_W] \cdot \pi$. In the new-source-detection phase for the sequence in which e'_W first appears, a sequence of form $e_R[src := e'_W] \cdot \pi'''$ for some π''' with $\pi''' \equiv \pi$ will be added to $schedules(\tau'')$. since $\pi = predecs(\pi, e'_W)$. Note that the detection of the alternative source e'_W for e_R is not prevented by the marking of events, since the read event e_R is not marked by any execution of line 9, and since a write event is always unmarked when it is explored for the first time. This concludes the proof of Claim 1. □

Picking up the main line of the proof of Theorem 6.4, we have established that $e_R[src := e_W] \cdot \pi'''$ will be added to $schedules(\tau'')$ for some π''' with $\pi''' \equiv \pi''$. This means that a call ReadsFrom-SMC$(\tau'' \cdot e_R[src := e_W] \cdot \pi''', -)$ will be performed. By the inductive hypothesis, it will also explore $\hat{\tau}$, since $\hat{\tau}$ extends $\tau'' \cdot e_R[src := e_W] \cdot \pi'''$, and we have derived a contradiction. Theorem 6.4 is thus proven. □

By leveraging the observation that ConsistencyDecision procedure has polynomial running time in the length of the program, we infer that our SMC algorithm is polynomial time for programs with a fixed number of threads.

Theorem 6.5. *Given a program with fixed number of threads, the ReadsFrom-SMC algorithm spends polynomial time for each explored trace.*

Proof. The theorem follows from the polynomial complexity of both the ConstructWitness procedure (Lemma 5.4) and the ConsistencyDecision procedure; see end of Sect. 5.3. □

7 IMPLEMENTATION

We have implemented our algorithm by extending the Nidhugg tool with a new mode, called rfsc. Nidhugg takes C or C++ programs as input and works at the level of LLVM IR. Executions are checked for assertion violations and crashes, such as segmentation faults. Nidhugg permits applying automatic loop bounding, sometimes called loop unrolling, to the input program. This allows it to be used on programs which do not terminate in bounded time, and hence have an infinite trace space. As with any bounding technique, this makes the exploration exhaustive only up to the given bound. Bugs may be missed if they do not manifest in any trace within the bound.

In the rfsc mode of Nidhugg, saturation of a trace is implemented using *vector clocks* as follows: Initialize a work queue Q with the first event. While Q is not empty, dequeue an event e and recompute its vector clock c. If c is unchanged, do nothing. If e is now happens-after more writes, take the last such write e_W of each thread. If e is a read event, apply saturation rule R1 to e_W, e, and $e.src$. If e is a write event, apply saturation rule R2 to e_W, e, and all readers of e_W. Finally, add all successors of e, as well as the target of any new shb edges to Q, and repeat until the queue empties.

For performance, Nidhugg/rfsc does not rebuild the trace from scratch each time it needs to check consistency. Rather, it represents traces with persistent immutable data structures, and caches the saturated trace for any consistent read. This way, to check consistency for a new read event, it only needs to add a few new events and re-saturate. However, the saturation algorithm, as presented, does not allow this *incremental saturation* use case, as it might fail to infer some edges implied by rule R2, in particular when the two writers are in the cached trace prefix, and the reader is not. To solve this, a special pass is added to infer these "missed" edges, that operates in an analogous way,

but performs saturation "in reverse", using vector clocks that encode the downward-reachable set of an event rather than the opposite. This "reverse saturation" is only performed for a subset of events, namely such readers, their source writes, and any successor events of these.

Nidhugg/rfsc implements an extension of the algorithm to atomic read-modify-write (RMW) and compare-and-exchange (CAS) operations. The only modification is in line 5 of Algorithm 1. If e_R is an RMW and e_W will be an RMW if it reads from $e_R.src$ (in the case of CAS) we also check the condition that $e_W.src = e_R$. In this case, we will attempt to "reverse" e_R and e_W, by using $e_R[src := e_W[src := e_R.src]]$ in σ. Naturally, we must also consider failed CAS operations for e_W.

Nidhugg/rfsc also implements an extension of the algorithm to mutexes, with both blocking and non-blocking ("trylock") locking operations. Non-blocking locking operations and unlock operations can be treated like RMWs, and do not need any changes to the algorithm. Blocking lock operations require two extensions. The first one is when the algorithm would consider reversing an unlock operation with a blocking lock operation e_L. Instead the lock e'_L, blocking or not, from which the unlock reads is replaced by $e_L[src := e'_L.src]$ in the trace fed to GetWitness(). The second one is for the case when an execution terminates with some threads still blocking on a lock. This is not necessarily an error. For example, it can be caused by failing an `assume` statement. In such cases, the algorithm considers "replacing" the last successful lock of every mutex with each blocked lock event of that mutex.

8 PERFORMANCE EVALUATION

In this section, we compare the performance of Nidhugg/rfsc with other SMC algorithms and tools that were publicly available in April 2019, when this paper was submitted. Two of these tools, Nidhugg and DC-DPOR, have been chosen because they implement state-of-the-art DPOR algorithms for SMC under Sequential Consistency; namely Optimal DPOR, Optimal DPOR with Observers, and Data-Centric DPOR. We briefly review these three algorithms. Optimal DPOR [Abdulla et al. 2017] is optimal in the sense that it explores each Mazurkiewicz trace *exactly once* and never initiates redundant exploration. Similar to ReadsFrom-SMC, Data-Centric DPOR [Chalupa et al. 2018] is based on the `rf` equivalence, but does not come with any optimality claims other than for the very restricted case when the underlying communication graph (i.e., the graph whose nodes are the program threads and two nodes are connected by an edge if they access a common shared variable) is acyclic. In other situations, DC-DPOR can explore a considerable number of partial executions, which later may be deemed redundant. (This is clearly shown in the tables that we will soon present.) Finally, Optimal DPOR with Observers [Aronis et al. 2018] is an algorithm which is optimal in the sense that it explores each trace exactly once and will never initiate redundant partial exploration, but its equivalence relation is somewhere between Mazurkiewicz and `rf`. Specifically, its equivalence relation respects the conflicts between write events only if their ordering is observed by a read event in the same trace. Still, Optimal DPOR with Observers will *never* explore more executions than the number of Mazurkiewicz traces, not even partially.

The remaining two tools, CDSChecker and RCMC, are high-performance stateless model checkers for C/C++11 programs, which of course also handle the SC fragment of C11. The CDSChecker tool by Norris and Demsky [2016] employs a variant of the DPOR algorithm of Flanagan and Godefroid [2005], which is not optimal and may explore a significant number of redundant executions; in fact many more than the number of Mazurkiewicz traces as we will soon see. The RCMC tool by Kokologiannakis et al. [2018] provides an interesting approach to SMC of concurrent programs. Its algorithms perform SMC primarily under Release-Acquire, a memory model which is weaker than SC and where the problem of checking consistency is of polynomial complexity. Moreover, RCMC checks for SC consistency lazily, only when an assertion is violated, thereby avoiding the cost of this check when this is unnecessary. As we show by Example 8.1 below, such an approach

which checks for SC lazily risks exploring arbitrarily more traces than an approach which checks for SC consistency eagerly. Moreover, RCMC's algorithms are not optimal under SC, and may explore many executions for each Mazurkiewicz trace. The RCMC tool provides a mode, selectable with its `-wrc11` switch, that does not consider all the possible ways that concurrent writes to the same location could be totally ordered. This triggers an algorithm which in many cases achieves similar reduction as the `rf` equivalence, but in other cases it may unnecessarily consider many incoherent executions. In fact, as pointed out in the RCMC paper [Kokologiannakis et al. 2018, Section 5], there are programs for which each mode of RCMC can outperform the other arbitrarily.

Example 8.1 (Lazy vs. eager SC consistency checks). This example illustrates that an SMC algorithm which explores `rf` traces that are feasible under a weaker memory model, such as Release-Acquire, and applies an SC consistency check *lazily* only to traces that violate correctness, risks exploring arbitrarily more traces than an approach which checks for SC consistency eagerly.

Consider a program that is parametric in n, shown in Fig. 6. In this program, two threads race to sum up counters $c_1, \ldots, c_n$, but are synchronized using sequentially consistent accesses using a simplified variant (for presentation) of Dekker's mutual exclusion algorithm. In any SC-consistent trace, at most one thread will be accessing the counters $c_1, \ldots, c_n$, and thus there will only be three traces; one where the first thread gets the mutex, one where the second thread gets the mutex, and one where neither thread does. However, under a weaker memory model, such as RA, it is possible for both threads to enter the critical section, and thus there will be $\binom{2n}{n}$ additional traces for all the ways to interleave the accesses to $c_1, \ldots, c_n$. Thus an eager technique would explore a constant number of traces and spend time linear in the parameter n, whereas a lazy technique would explore a number of traces, and spend time exponential in n. □

$a = b = x = y = c_1 = \cdots = c_n = 0$

$x = 1;$	$y = 1;$
if $(y == 1)$ return;	if $(x == 1)$ return;
for i in $(1, n)$	for j in $(1, n)$
a += xhg$(c_i, 0)$;	b += xhg$(c_j, 0)$;

Fig. 6. Program with an exponential number of RA-consistent traces, but a constant number of SC-consistent traces.

Benchmarks and Platform. The benchmark programs we use in this section all are parametric in the number of threads, and are taken from SV-COMP [2019], SCTBench [Thomson et al. 2016], and from the papers that describe the tools we use [Abdulla et al. 2017; Aronis et al. 2018; Chalupa et al. 2018]. In order for these programs to be handled by all tools, and RCMC in particular, we needed to convert some of them to C11 with SC read and write accesses. The machine we used is a Dell server with two Intel(R) Xeon(R) Platinum 8168 CPUs (2.70GHz), has 192GB of RAM and ran Debian 9.6. All tools use `Clang` version 3.8.1 to translate the C source to LLVM IR. In all tables of this section, we report number of explored executions (both complete and partial/redundant) and the time this takes (in seconds).

As mentioned, the `rf` equivalence is coarser than the Mazurkiewicz one, and the ReadsFrom-SMC algorithm is optimal for it. So, by definition, Nidhugg/rfsc never explores more executions than other SMC tools. Besides confirming this, with our evaluation we want to answer the following questions: Q1) What is Nidhugg/rfsc's performance in programs that have the same number of `rf` traces as Mazurkiewicz traces? Q2) Are there programs where Nidhugg/rfsc beats each of the other tools by an exponential factor in the number of executions explored? Q3) What is Nidhugg/rfsc's performance in programs which fall somewhere between these two extremes? We answer these questions below.

8.1 Performance When Reads-From Coincides with Mazurkiewicz

First, note that in this section we examine programs that have the same number of Mazurkiewicz traces as `rf` traces; not necessarily programs on which all tools explore the same number of

Table 1. Performance on two programs where the number of Mazurkiewicz traces and traces under rf equivalence is the same. Note that tools that are not optimal sometimes explore a significant number of redundant executions ("Execs" columns). The Nidhugg column in this table stands for both Nidhugg/optimal and Nidhugg/observers: both these modes explore the same number of executions in time that is very similar.

	Nidhugg		DC-DPOR		CDSChecker		RCMC		RCMC/wrc11		Nidhugg/rfsc	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time
circular-buffer(7)	3432	1.13	80641	12.84	3432	0.18	3432	1.27	43178	11.73	3432	1.29
circular-buffer(8)	12870	4.55	303149	55.39	12870	0.75	12870	5.16	200242	61.94	12870	5.21
circular-buffer(9)	48620	18.93	1147421	245.82	48620	3.08	48620	21.26	916247	313.71	48620	21.19
lastzero(9)	1536	0.62	24537	5.50	18339	1.85	1536	0.04	2560	0.06	1536	0.62
lastzero(11)	7168	3.55	172563	49.29	184331	22.66	7168	0.19	12288	0.28	7168	3.35
lastzero(13)	32768	19.44	1200300	467.43	1888624	270.64	32768	0.87	57344	1.35	32768	17.46

Table 2. Performance on one more set of programs for which the number of Mazurkiewicz traces and traces under rf equivalence is the same. We show only one column for RCMC in this table, as numbers for the two modes of RCMC are similar for these programs; in particular, the number of executions is the same.

	Nidhugg/optim		Nidhugg/observ		DC-DPOR		CDSChecker		RCMC		Nidhugg/rfsc	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time
readers(9)	512	0.19	512	0.19	262144	61.87	831	0.08	512	0.04	512	0.18
readers(11)	2048	0.90	2048	0.94	4194304	1445.38	3327	0.39	2048	0.17	2048	0.87
readers(13)	8192	4.22	8192	4.28	67108864	31046.08	13311	1.88	8192	0.98	8192	3.92
casrot(8)	2048	0.64	2048	0.64	n/a	n/a	44832	3.07	2048	0.02	2048	0.65
casrot(9)	8597	2.96	8597	2.99	n/a	n/a	372735	29.32	8597	0.08	8597	3.04
casrot(10)	38486	14.58	38486	14.80	n/a	n/a	3456845	305.81	38486	0.36	38486	14.92
fib-bench(2)	140	0.02	140	0.02	578	0.06	n/a	n/a	153	0.01	140	0.03
fib-bench(4)	19605	4.62	19605	4.75	78737	11.46	n/a	n/a	34205	1.09	19605	5.23
fib-bench(6)	2364418	684.83	2364418	714.04	9514410	1827.93	n/a	n/a	8149694	306.66	2364418	1121.84

executions. Second, we point out that this is an important class of programs; in particular, all programs with properly synchronized SC accesses to shared variables fall in this class. So, perhaps for these programs it may be a better strategy to use an SMC tool whose algorithm is optimal for Mazurkiewicz traces (e.g., Nidhugg/optimal), or a tool that explores traces fast, either due to having been well engineered for this or due to performing SMC under a weaker memory model and paying the cost of SC checks only when this is needed (e.g., when an assertion is violated).

In fact, our results from this class of programs partially confirm that this is a good strategy. Refer to Tables 1 and 2. Nidhugg/rfsc explores the smallest number of executions in all benchmarks, but its performance is between four and forty times slower per trace than the fastest tool (CDSChecker for circular-buffer in Table 1, and RCMC for all other benchmarks in Tables 1 and 2) when that tool gets by with exploring only the same number of executions as Mazurkiewicz traces. On the other hand, the same evaluation also shows the following:

(i) With the exception of fib-bench, Nidhugg/rfsc has roughly similar performance as Nidhugg, a tool that uses DPOR-based algorithms that build traces by reversing races, which means that their traces are "by construction" consistent under SC and thus require no checks.
(ii) The cost per trace that Nidhugg/rfsc pays appears to be constant as the number of traces grows. This can be seen by comparing with RCMC which performs no checks in some programs. (Benchmarks lastzero, readers and casrot contain no assertions.) On the other hand,

Table 3. Performance on a set of benchmarks where the rf equivalence is significantly coarser than the Mazurkiewicz one. CDSChecker quickly times out or crashes for three out of these five programs. The comparison here is interesting mostly between Nidhugg/observers, DC-DPOR, RCMC/wrc11, and Nidhugg/rfsc.

	Nidhugg/optim		Nidhugg/observ		DC-DPOR		RCMC		RCMC/wrc11		Nidhugg/rfsc	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time
lastwrite(2)	2	0.00	2	0.00	3	0.00	2	0.01	2	0.01	2	0.00
lastwrite(7)	5040	1.01	7	0.00	8	0.00	5040	0.24	7	0.00	7	0.00
lastwrite(8)	40320	9.11	8	0.00	9	0.00	40320	15.52	8	0.00	8	0.00
lastwrite(9)	362880	92.97	9	0.00	10	0.00	362880	2007.57	9	0.01	9	0.00
sigma(5)	945	0.20	120	0.03	872	0.16	945	0.03	120	0.01	120	0.03
sigma(6)	10395	2.62	720	0.20	5912	1.36	10395	0.40	720	0.03	720	0.19
sigma(7)	135135	38.30	5040	1.61	46232	12.81	135135	21.98	5040	0.23	5040	1.49
sigma(8)	2027025	687.76	40320	14.14	409112	142.69	2027025	9255.23	40320	2.34	40320	13.41
floating-read(6)	5040	1.06	193	0.04	8	0.00	5040	0.18	7	0.01	7	0.00
floating-read(8)	362880	98.45	1025	0.30	10	0.00	362880	119.32	9	0.01	9	0.00
floating-read(12)	🕒	🕒	24577	10.59	14	0.08	🕒	🕒	13	0.01	13	0.00
floating-read(16)	🕒	🕒	524289	352.10	18	1.78	🕒	🕒	17	0.01	17	0.00
control-flow(5)	5160	1.33	193	0.05	8083	0.86	5160	0.16	43	0.01	43	0.01
control-flow(7)	720720	363.95	1025	0.33	174859	28.35	720720	30.92	143	0.01	143	0.05
control-flow(11)	🕒	🕒	24577	11.89	🕒	🕒	🕒	🕒	2071	0.15	2071	0.93
control-flow(13)	🕒	🕒	114689	65.81	†	†	🕒	🕒	8219	0.67	8219	4.21
opt-lock(11)	300090	96.76	170	0.04	663	0.05	414714	21.27	1497	0.06	127	0.04
opt-lock(13)	2056906	770.75	202	0.05	791	0.06	2842574	163.25	2083	0.09	151	0.05
opt-lock(100)	🕒	🕒	1594	2.15	6359	2.09	🕒	🕒	120490	29.91	1195	1.99
opt-lock(1000)	🕒	🕒	15994	504.06	63959	224.86	🕒	🕒	🕒	🕒	11995	227.46

on the circular-buffer benchmark that *does* contain assertions, Nidhugg/rfsc has performance almost identical to RCMC (and RCMC/wrc11).

(iii) No single tool is fastest overall: CDSChecker beats all other tools by more than six times on circular-buffer, while RCMC beats all other tools in all other benchmarks, not only in cases where it manages to explore the optimal number of executions, but also on fib-bench where it explores about 3.5 times more executions.

(iv) Tools that do not employ optimal algorithms indeed explore a significant number of redundant (or partial) executions and this can significantly hurt their performance; cf. DC-DPOR on e.g., readers (Table 2), and CDSChecker on lastzero (Table 1) and casrot (Table 2).

8.2 Performance When Reads-From Is Significantly Coarser Than Mazurkiewicz

Table 3 shows results from synthetic benchmarks for which the traces under the rf equivalence are significantly fewer than the Mazurkiewicz traces. As a consequence, three of the tools/modes (Nidhugg/optimal, RCMC and CDSChecker, which is not shown) explode in terms of explored executions and quickly time out. We therefore focus on the remaining four tools (Nidhugg/observers, DC-DPOR, RCMC/wrc11, and Nidhugg/rfsc) for this benchmark set.

(i) All four tools perform similarly on lastwrite, where DC-DPOR also reports one partially explored execution. In contrast, the number of partially explored executions of DC-DPOR on sigma is significant and hurts its performance.

(ii) The floating-read benchmark shows a case where Optimal DPOR with Observers [Aronis et al. 2018] is outperformed exponentially by the algorithms that the other three tools (DC-DPOR, RCMC/wrc11, and Nidhugg/rfsc) employ. We remark that this is quite interesting

Table 4. Performance on a set of benchmarks where the rf and the Mazurkiewicz equivalence differ but moderately. CDSChecker is slow and/or cannot handle these programs, so we do not show results for it.

	Nidhugg/optim		Nidhugg/observ		DC-DPOR		RCMC		RCMC/wrc11		Nidhugg/rfsc	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time
race-parametric(4)	3334	2.63	3334	2.69	6137	1.96	3334	0.55	4116	0.57	1107	1.13
race-parametric(5)	34904	27.72	34904	29.11	34548	14.08	34904	5.68	49104	6.68	8953	9.06
race-parametric(6)	372436	302.68	372436	314.63	270055	120.15	372436	62.04	608751	82.72	73789	75.73
race-parametric(7)	4027216	3406.95	4027216	3505.99	2248090	1320.72	4027216	696.83	7770620	1094.63	616227	660.83
exponential-bug(3)	36570	8.34	7714	1.81	8821	1.06	46764	0.53	8224	0.10	3634	0.97
exponential-bug(5)	345069	82.73	28411	7.35	21236	2.88	463982	4.79	20321	0.23	10131	2.95
exponential-bug(7)	2044724	529.98	76460	21.37	43659	6.48	2833112	28.68	41692	0.45	22694	7.07
parker(8)	3431	1.00	3171	0.99	14073	1.84	21390	3.66	55938	10.67	3059	1.23
parker(12)	10541	3.71	9759	3.63	43309	6.85	92814	26.68	213304	64.54	9407	4.71
parker(16)	23731	10.09	21995	9.53	97633	18.71	271710	123.19	575550	264.00	21195	12.65
lamport(2)	32	0.00	26	0.00	119	0.01	40	0.01	∞	∞	22	0.00
lamport(3)	20741	6.59	11598	4.12	60937	7.03	83732	9.93	∞	∞	8175	2.88

considering that this benchmark is from the Optimal DPOR with Observers paper, and that this is a program which also demonstrates significant differences between the Mazurkiewicz equivalence (cf. the Nidhugg/optimal and RCMC columns) and the equivalence of Optimal DPOR with Observers. Also interesting here is to note that DC-DPOR spends a considerable amount of time on floating-read(16), for reasons currently unknown to us.

(iii) The control-flow benchmark makes DC-DPOR explode, and shows one more case where ReadsFrom-SMC and Optimal DPOR with Observers differ significantly in terms of explored executions. On the other hand, RCMC/wrc11 is both optimal and the fastest tool (by 6.2 times) on this benchmark.

(iv) Finally, the opt-lock benchmark, taken from the DC-DPOR paper [Chalupa et al. 2018], shows a case where our algorithm is the only one which is optimal. (DC-DPOR explores the same number of complete executions, but it also explores about five times as many partial executions.) Also, notice how the number of traces grows for RCMC/wrc11 (e.g., for opt-lock(100)) in contrast to Nidhugg/rfsc, DC-DPOR, and also Nidhugg/observers, for which the increase in the number of redundant executions explored is very moderate.

8.3 Performance When Reads-From and Mazurkiewicz Differ Moderately

Finally, Table 4 contains results from programs that fall somewhere in between the two ends of the spectrum. In these programs,[2] the difference in the number of Mazurkiewicz and rf traces grows as the number of threads increases, but not as dramatically as in programs of the previous section. The results confirm that indeed Nidhugg/rfsc explores the least number of executions in all cases. Also, although it does not start as the fastest tool, Nidhugg/rfsc manages to outperform all other tools in all programs except on exponential-bug where it is beaten by RCMC/wrc11 (and DC-DPOR, slightly).

As a side comment, we note that Table 4 also confirms that no mode of RCMC beats the other all the time: RCMC/wrc11 is significantly better on exponential-bug, but worse in the other three programs (in fact, on lamport RCMC/wrc11 does not terminate).

[2]We mention that parker is slightly different from the version used in the DC-DPOR paper [Chalupa et al. 2018], as we are loop-unrolling the benchmark manually, to make the comparison fair.

Table 5. Number of explored executions until the bug is Hit and Total executions in the "corrected" program.

	Nidhugg/optim		Nidhugg/observ		DC-DPOR		Nidhugg/rfsc	
Benchmark	Hit Bug	Total	Hit Bug	Total	Hit Bug	Total	Hit Bug	Total
reorder_3_bad	9	56	9	47	13	384	2	21
reorder_4_bad	97	1 248	71	580	19	3 209	2	64
reorder_5_bad	1 513	40 032	681	6 565	32	29 596	4	145
reorder_10_bad	$> 2^{32}$	$> 2^{32}$	51 398 385	432 764 218	1 297	1 692 398 701	5	1 540

9 WHY THE READS-FROM EQUIVALENCE MATTERS FOR SMC

In this section, we provide some more evidence why the `rf` equivalence in general, and our ReadsFrom-SMC algorithm in particular, matter for SMC tools by examining two scenarios where they come in handy. The first is in quickly finding concurrency errors and verifying that they have been correctly fixed. The second concerns stateless model checking of approximate data structures.

9.1 Finding and Correcting Concurrency Errors Faster

All programs in Sect. 8 did not contain concurrency errors. In these programs, SMC tools need to explore the complete set of executions. In programs where a crash occurs or an assertion is violated under some thread schedulings, SMC tools can stop the exploration as soon as the problem occurs. How fast this happens depends on how frequent/rare the bug is, but also on the order in which tools choose to explore executions.

Starting from five programs of SCTBench [Thomson et al. 2016], the Systematic Concurrency Testing Benchmark Suite, we conducted an experiment in order to see whether the `rf` equivalence provides some advantage as far as bug finding is concerned. For our experiment, we chose the five reorder_N_bad benchmarks (N ∈ {3,4,5,10,20} is the number of created pthreads). These are programs adversarial for delay bounding; the smallest delay bound required for the bug to manifest is incremented as the thread count is incremented. Thomson et al. [2016] report that, although many systematic concurrency testing techniques locate the bug for small thread counts (N ∈ {3,4,5}), only probabilistic concurrency testing with larger d values [Burckhardt et al. 2010] manages to hit the bug on reorder_10_bad and reorder_20_bad. (This is on an experiment that runs each technique for 100 000 executions.) A recent paper by Yuan et al. [2018] reports an experiment with 10 000 executions, confirms that overall these are the second and third, in this order, most challenging programs from the SCTBench suite, and proposes a sophisticated partial order aware sampling technique that manages to locate the bug in reorder_10_bad with hit rate of approximately 3%. So, we were curious to see what the various SMC tools of Sect. 8 that handle C instead of C11 source do in these programs; in particular, to see how Nidhugg/rfsc performs in this task.

Refer to Table 5; initially look only at the "Hit Bug" columns. Nidhugg/rfsc manages to locate the bug in just the first few traces (five or less), while the number of executions for the other tools is significant. Of course, one could argue that Nidhugg/rfsc is "just lucky here." To see whether this is indeed the case, we "corrected" the programs by commenting out the single assertion,[3] and tried to run all tools to completion. The number of executions that the tools explore is shown under the "Total" columns of Table 5. For reorder_10_bad, besides Nidhugg/rfsc that finishes in just 1.26 seconds after exploring 1540 traces, we also managed to obtain numbers for Nidhugg/observers, which finishes in 5.6 days after exploring 432.7 million executions, and for DC-DPOR, which finishes in 6.8 days after exploring fully or partially 1.69 billion executions (120 593 170 of which are fully explored). In contrast, Nidhugg/optimal does not find the bug, let alone finish exploring

[3]We chose this "correction" method because it does not affect the number of executions that tools need to explore.

Table 6. Performance on benchmark where N threads append elements on an approximate data structure.

	Nidhugg/optim		Nidhugg/observ		DC-DPOR		RCMC		Nidhugg/rfsc	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time
approxds-append(4)	2460	1.05	1048	0.51	4751	0.60	2508	0.18	580	0.30
approxds-append(5)	127740	62.91	33125	20.69	154085	20.81	135900	35.11	9945	5.55
approxds-append(6)	9847080	5910.97	1334766	1003.34	6121579	1141.64	11226600	444456.16	198936	131.43

the "corrected" program, even after exploring more than 2^{32} executions in more than one and a half months. Note that this last experiment is *exactly* the next step that a developer who corrects a concurrency error needs to take: run the SMC tool to completion in order to see whether the bug has indeed been fixed or whether more bugs exist. It is nice to have a tool like Nidhugg/rfsc that finishes in seconds instead of several days (or months!).

We finish this section by mentioning what happens on reorder_20_bad. Actually, this program is not just a "scaled up" version of the previous programs, which only have one reader, but a program where the N spawned threads contain more readers, which makes locating the bug easier. For this reason, we do not include it in Table 5. But Nidhugg/rfsc finds the bug in reorder_20_bad after just two traces; all other tools we used did not manage to locate it even after many days.

9.2 Stateless Model Checking of Approximate Data Structures

The second experiment concerns approximate concurrent data structures with construction algorithms that execute without synchronization, as proposed by Rinard [2013]. The data races present in these algorithms may cause them to e.g., drop inserted or appended elements. Nevertheless, the "correctness" requirement is that their construction algorithms should not crash and should produce a data structure that is accurate enough for its clients to use successfully. The argument is that this is acceptable for many applications. Note that our aim here is simply to propose an efficient way for the testing and verification of such data structures to those who want to develop and/or use them.

For our experiment, we took the code of the paper [Rinard 2013, Page 4], wrote it in C11 with atomic accesses, and made it parametric in the number of threads that try to append elements. We then applied the various SMC tools of Sect. 8 to this program. The results we got are shown in Table 6. The superiority of Nidhugg/rfsc's performance on this experiment is clear. We note in passing that we could not get numbers for RCMC/wrc11 because the tool in this mode quickly starts consuming more and more memory when running this program.

10 RELATED WORK

Several influential tools such as Verisoft [Godefroid 1997, 2005] and CHESS [Musuvathi et al. 2008] implement stateless model checking for concurrent programs. SMC has been combined with (dynamic) partial order reduction (DPOR), e.g. by Flanagan and Godefroid [2005], Abdulla et al. [2014], and Rodríguez et al. [2015], and applied to real life programs [Godefroid et al. 1998; Kokologiannakis and Sagonas 2017]. The majority of SMC techniques are based on exploring at least one execution for each Mazurkiewicz trace. In 2014, Abdulla et al. introduced a method that is optimal with respect to Mazurkiewicz traces. As illustrated in the introduction, this approach has an inherent limit on the reduction that can be achieved.

In light of this, several more recent papers use a weaker equivalence relation than Mazurkiewicz traces. The Maximal Causality Reduction (MCR) algorithm by Huang [2015], which is based on the Maximal Causal Model (MCM) idea by Serbanuta et al. [2013], considers the possible *values*

that reads can see, instead of the possible value-producing writes, as we do in in our approach. This approach has the potential to explore fewer traces than our approach. An obstacle is that the problem of checking consistency for a particular assignment of values to reads can occur in an execution is more difficult than the problem of checking consistency of a particular `rf` relation, which can by done in polynomial time by our ConsistencyDecision procedure. MCR addresses this by relying on calls to an SMT solver to find new executions. Furthermore, MCR is not optimal in the sense that it may explore executions that are equivalent to previously explored ones, as shown by Chalupa et al. [2018]. Possible future work could be to investigate whether our optimal exploration strategy can be combined with the MCM idea in an efficient way.

In Sect. 8, we compared to DC-DPOR [Chalupa et al. 2018], Optimal DPOR with Observers [Aronis et al. 2018], as well as RCMC [Kokologiannakis et al. 2018] and CDSChecker [Norris and Demsky 2016]. We refer to that section for an overview of these algorithms, the equivalence classes that they are based on, and the optimality guarantees that some of these algorithms provide.

Context-Sensitive DPOR by Albert et al. [2017] is a technique for exploring less executions than the classical DPOR algorithm of Flanagan and Godefroid. It uses an external procedure to decide whether alternative schedulings would lead to identical states. In certain cases, it can achieve exponential reduction over the classical DPOR. However, since it needs to compare states, it is an inherently stateful technique, in contrast to other stateless model checking techniques that maintain information only about currently explored trace and how to initiate new explorations.

For the Release-Acquire semantics (RA), Abdulla et al. [2018] have developed a DPOR algorithm which is optimal with respect to the `rf` equivalence. The RA semantics is different from the SC semantics, and hence results obtained for SMC of programs running under RA cannot be carried over to the case of SC. Since RA is weaker than SC, one can in principle use DPOR algorithms developed for RA as an over-approximation for the verification of programs running under the SC semantics. However, such an over-approximation will not satisfy the soundness property and can potentially generate a significant number of false safety violations. Furthermore, as shown in Example 8.1, supplementing such an algorithm with a lazy consistency check may generate an exponentially larger number of redundant traces. Nevertheless, our algorithm for SMC under SC shares some principles with the one under RA by Abdulla et al. [2018]. Two main issues make the search algorithm more complicated in the case of SC. First, as described in Sect. 5, the saturation rules are simpler to formulate and can be implemented more efficiently in the case of RA. Second, due to the completeness of consistency checking procedure of RA, it can be carried out by simply searching for cycles in a saturated trace, rather than having three phases as described in Sect. 5.

The idea of characterizing executions by the `rf` relation (i.e., abstracting them as `rf` traces) has been exploited for runtime monitoring or predictive analysis by Sen et al. [2005], Wang and Stoller [2006], and Sinha et al. [2011]. These works present techniques for analyzing individual executions of a concurrent programs in order to detect correctness violations in executions that belong to the same or a neighbouring `rf` equivalence class. In contrast, our work presents a technique for exploring *all* `rf` equivalence classes of a concurrent program.

An algorithm similar to Algorithm 3 has been developed independently by Biswas and Enea [2019] for checking serializability of bounded-width histories. Their result also implies that checking sequential consistency of histories with a bounded number of threads is polynomial time.

11 CONCLUSION AND FUTURE WORK

We have presented a novel SMC algorithm for the verification of programs running under the SC semantics. The algorithm is optimal with respect to the `rf` equivalence, i.e., it never generates two program executions with the same reads-from relations. To that end, we have designed an efficient exploration algorithm (Sect. 4) and a check for consistency of a given `rf` relation (Sect. 5). The

SMC exploration algorithm satisfies three properties: soundness, completeness, and optimality. The consistency check consists of three phases. The first two phases are polynomial time in the size of the program, but an incomplete check for consistency. The third phase is a sound and complete procedure which decides consistency in polynomial time in the length of the program and exponential in the number of threads. An important finding is that the polynomial-time test represented by the first two phases is sufficient for checking consistency of `rf` relations on all programs that we have tried, implying that our consistency check is efficient in practice.

We have implemented our approach by extending the Nidhugg tool for C/C++ programs with a new mode, called rfsc. Our experimental results show that Nidhugg/rfsc, although slower than the fastest SMC tools in programs where tools happen to examine the same number of executions, always scales similarly or better than them, and outperforms them by an exponential factor in programs where the number of `rf` traces is smaller than the number of Mazurkiewicz traces. We also presented two non-trivial use cases where the new equivalence is particularly effective, as well as the significant performance advantage that the rfsc of Nidhugg offers compared to other state-of-the-art SMC and systematic concurrency testing tools.

We note that the approach presented here assumes that each read event reads from a unique write event in an execution. This is not true for all programs; for example it does not hold in the code of Linux's Read-Copy Update (RCU) mechanism that was model checked by Kokologiannakis and Sagonas [2017], which contains byte-wide write events that are then read as a (half) word. How to generalize the algorithm to handle such cases efficiently is left for future work.

Another interesting direction for future work is to use an even weaker relation than `rf` equivalence, while still maintaining a polynomial time complexity for checking consistency of a trace for a fixed number of threads. This would allow to design algorithms that are potentially more efficient but still sound for checking assertions in concurrent programs. Furthermore, we aim at improving the overhead cost per trace, e.g., by developed techniques based on net unfoldings [Esparza and Heljanko 2008], to improve our algorithm for checking consistency of traces. An interesting direction is to adapt our algorithm to work with symbolic states, thereby providing full correctness in the context of symbolic model checking; a challenge is how to combine a symbolic representation of states with ordering information in traces in an efficient manner.

ACKNOWLEDGMENTS

We thank the OOPSLA 2019 reviewers for helpful feedback, and Martin Rinard for pointing us to code that we used for a benchmark. This work has been partially supported by the Swedish Research Council through UPMARC and grant #621-2017-04812, and by the Swedish Foundation for Strategic Research through the aSSIsT project.

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Symposium on Principles of Programming Languages (POPL 2014)*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/2535838.2535845

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 9035. Springer, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM* 64, 4, Article 25 (Sept. 2017), 49 pages. https://doi.org/10.1145/3073408

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-From Equivalence under Sequential Consistency. https://doi.org/10.5281/zenodo.3401442 Artifact for the OOPSLA 2019 paper with the same title.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM on Program. Lang.* 2, OOPSLA (2018), 135:1–135:29. https://doi.org/10.

1145/3276505

Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. 2017. Context-Sensitive Dynamic Partial Order Reduction. In *Computer Aided Verification (LNCS)*, Vol. 10426. Springer, Berlin Heidelberg, 526–543. https://doi.org/10.1007/978-3-319-63387-9_26

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal Dynamic Partial Order Reduction with Observers. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference (LNCS)*, Vol. 10806. Springer, Cham, 229–248. https://doi.org/10.1007/978-3-319-89963-3_14

Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proc. ACM on Program. Lang.* 3, OOPSLA (2019).

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-centric Dynamic Partial Order Reduction. *Proc. ACM on Program. Lang.* 2, POPL (2018), 31:1–31:30. https://doi.org/10.1145/3158119

Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. 2013. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*. IEEE, Los Alamitos, CA, USA, 154–163. https://doi.org/10.1109/ICST.2013.50

Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. 1999. State Space Reduction Using Partial Order Techniques. *Software Tools for Technology Transfer* 2, 3 (1999), 279–287. https://doi.org/10.1007/s100090050035

Javier Esparza and Keijo Heljanko. 2008. *Unfoldings - A Partial-Order Approach to Model Checking*. Springer. https://doi.org/10.1007/978-3-540-77426-6

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages, (POPL)*. ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Ph.D. Dissertation. University of Liège. https://doi.org/10.1007/3-540-60761-7 Also, volume 1032 of LNCS, Springer.

Patrice Godefroid. 1997. Model Checking for Programming Languages using VeriSoft. In *Principles of Programming Languages, (POPL)*. ACM Press, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design* 26, 2 (March 2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

Patrice Godefroid, Robert Hammer, and Lalita Jagadeesan. 1998. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, 124–133. https://doi.org/10.1145/271771.271800

Matthew Hennessy and Robin Milner. 1980. On Observing Nondeterminism and Concurrency. In *Automata, Languages and Programming. ICALP 1980 (LNCS)*, Vol. 85. Springer, Berlin, Heidelberg, 299–309. https://doi.org/10.1007/3-540-10003-2_79

Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/2737924.2737975

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM on Program. Lang.* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

Michalis Kokologiannakis and Konstantinos Sagonas. 2017. Stateless Model Checking of the Linux Kernel's Hierarchical Read-Copy-Update (Tree RCU). In *Proceedings of International SPIN Symposium on Model Checking of Software (SPIN 2017)*. ACM, New York, NY, USA, 172–181. https://doi.org/10.1145/3092282.3092287

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Antoni Mazurkiewicz. 1987. Trace Theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency (LNCS)*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.), Vol. 255. Springer, Berlin Heidelberg, 279–324. https://doi.org/10.1007/3-540-17906-2_30

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérald Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, Berkeley, CA, USA, 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 10 (May 2016), 51 pages. https://doi.org/10.1145/2806886

Doron A. Peled. 1993. All from one, one for all, on model-checking using representatives. In *Computer Aided Verification (LNCS)*, Vol. 697. Springer-Verlag, London, UK, 409–423. https://doi.org/10.1007/3-540-56922-7_34

Martin Rinard. 2013. Parallel Synchronization-Free Approximate Data Structure Construction. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*. USENIX Association. https://www.usenix.org/conference/hotpar13/workshop-program/presentation/Rinard

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based Partial Order Reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015 (LIPIcs)*, Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 456–469. https://doi.org/10.4230/LIPIcs.CONCUR.2015.456

Koushik Sen and Gul Agha. 2007. A Race-Detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs. In *Hardware and Software, Verification and Testing (LNCS)*, Vol. 4383. Springer, Berlin Heidelberg, 166–182. https://doi.org/10.1007/978-3-540-70889-6_13

Koushik Sen, Grigore Rosu, and Gul Agha. 2005. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Formal Methods for Open Object-Based Distributed Systems (LNCS)*, Vol. 3535. Springer, Berlin Heidelberg, 211–226. https://doi.org/10.1007/11494881_14

Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. 2013. Maximal Causal Models for Sequentially Consistent Systems. In *Runtime Verification (RV) (LNCS)*, Shaz Qadeer and Serdar Tasiran (Eds.), Vol. 7687. Springer, Berlin Heidelberg, 136–150. https://doi.org/10.1007/978-3-642-35632-2_16

Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. 2011. Predictive analysis for detecting serializability violations through Trace Segmentation. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 99–108. https://doi.org/10.1109/MEMCOD.2011.5970516

SV-COMP. 2019. Competition on Software Verification. https://sv-comp.sosy-lab.org/2019. [Online; accessed 2019-03-24].

Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2016. Concurrency Testing Using Controlled Schedulers: An Empirical Study. *ACM Trans. Parallel Comput.* 2, 4, Article 23 (2016), 37 pages. https://doi.org/10.1145/2858651

Antti Valmari. 1991. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets 1990 (LNCS)*, Grzegorz Rozenberg (Ed.), Vol. 483. Springer-Verlag, London, UK, 491–515. https://doi.org/10.1007/3-540-53863-1_36

Liqiang Wang and Scott D. Stoller. 2006. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, NY, USA, 137–146. https://doi.org/10.1145/1122971.1122993

Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *Computer Aided Verification (LNCS)*, Vol. 10982. Springer International Publishing, Cham, 317–335. https://doi.org/10.1007/978-3-319-96142-2_20

Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 250–259. https://doi.org/10.1145/2737924.2737956