Optimal Stateless Model Checking under the Release-Acquire Semantics

PAROSH AZIZ ABDULLA, Uppsala University, Sweden
MOHAMED FAOUZI ATIG, Uppsala University, Sweden
BENGT JONSSON, Uppsala University, Sweden
TUAN PHONG NGO, Uppsala University, Sweden

We present a framework for the efficient application of stateless model checking (SMC) to concurrent programs running under the Release-Acquire (RA) fragment of the C/C++11 memory model. Our approach is based on exploring the possible *program orders*, which define the order in which instructions of a thread are executed, and *read-from* relations, which specify how reads obtain their values from writes. This is in contrast to previous approaches, which also explore the possible *coherence orders*, i.e., orderings between conflicting writes. Since unexpected test results such as program crashes or assertion violations depend only on the read-from relation, we avoid a potentially significant source of redundancy. Our framework is based on a novel technique for determining whether a particular read-from relation is feasible under the RA semantics. We define an SMC algorithm which is provably optimal in the sense that it explores each program order and read-from relation exactly once. This optimality result is strictly stronger than previous analogous optimality results, which also take coherence order into account. We have implemented our framework in the tool Tracer. Experiments show that Tracer can be significantly faster than state-of-the-art tools that can handle the RA semantics.

CCS Concepts: • **Software and its engineering → Formal software verification**;

Additional Key Words and Phrases: software model checking, weak memory models, C/C++11, Release-Acquire

ACM Reference Format:
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking under the Release-Acquire Semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (November 2018), 29 pages. https://doi.org/10.1145/3276505

1 INTRODUCTION

Ensuring correctness of concurrent programs is difficult since one must consider all the different ways in which threads can interact. 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; so that 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 quickly reproduce the concurrency bugs it detects.

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; Tuan Phong Ngo, Uppsala University, Sweden, tuan-phong.ngo@it.uu.se.

2475-1421/2018/11-ART135
https://doi.org/10.1145/3276505

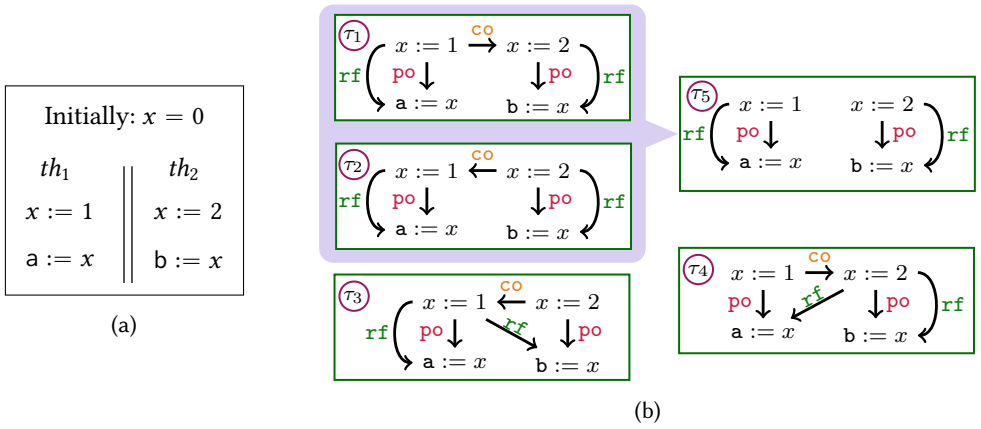


Fig. 1. (a) A simple concurrent program and (b) Shasha-Snir traces and weak traces.

SMC has been implemented in tools, such as VeriSoft [Godefroid 2005], CDSChecker [Demsky and Lam 2015; Norris and Demsky 2016], Chess [Musuvathi et al. 2008], Concuerror [Christakis et al. 2013], rInspect [Zhang et al. 2015], and Nidhugg [Abdulla et al. 2015a], and successfully applied to realistic concurrent programs [Godefroid et al. 1998; 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 1990], adapted to SMC as *dynamic partial order reduction* (DPOR). DPOR was first developed for concurrent programs that execute under the standard model of Sequential Consistency (SC) [Abdulla et al. 2014; Flanagan and Godefroid 2005; Sen and Agha 2007]. In recent years, DPOR has been adapted to hardware-induced weak memory models, such as TSO and PSO [Abdulla et al. 2015a; Zhang et al. 2015], and language-level concurrency models, such as the C/C++11 memory model [Kokologiannakis et al. 2018; Norris and Demsky 2016]. 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. Under SC, such equivalence classes are called *Mazurkiewicz traces* [Mazurkiewicz 1986]; for weak memory models, the natural generalization of Mazurkiewicz traces are called *Shasha-Snir traces* [Shasha and Snir 1988]. A Shasha-Snir trace characterizes an execution of a program by three relations between events; (i) po ("program order") totally orders the events of each thread, (ii) co ("coherence") totally orders the writes to each shared variable, and (iii) rf ("read-from") connects each write with the reads that read its value. Under weak memory models, the co and rf relations need not be derived from the global order in which events occur in an execution (as is the case under SC). Each particular weak memory model therefore imposes restrictions on how these relations may be combined.

As an illustration, Figure 1a shows a simple program with two threads, th_1 and th_2, that communicate through 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 many memory models, including SC, TSO, PSO, RA, and POWER, executions of the program in Figure 1a fall into four equivalence classes, represented by the four possible Shasha-Snir traces τ_1, τ_2, τ_3, and τ_4 in

Figure 1b[1]. A DPOR algorithm based on Shasha-Snir traces (e.g., [Abdulla et al. 2014, 2016b]) must thus explore at least four executions. However, it is possible to reduce this number further. Namely, a closer inspection reveals that the two traces τ_1 and τ_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 τ_1 and τ_2 have the same program order (po) and read-from (rf) relation. Their only difference is how writes are ordered by co, but this is not relevant for the computed results.

The preceding example illustrates that there is a potential for improving the efficiency of DPOR algorithms by using a weaker equivalence induced only by po and rf. In this example, the improvement is modest (reducing the number of explored traces from four to three), but it can be significant, sometimes even exponential, for more extensive programs. Several recent DPOR techniques try to exploit the potential offered by such a weaker equivalence [Chalupa et al. 2018; Huang 2015; Huang and Huang 2016; Norris and Demsky 2016]. However, except for the minimal case of an acyclic communication graph [Chalupa et al. 2018], they are far from optimally doing this, since they may still explore a significant number of different executions with the same rf relation. Therefore, the challenge remains to define a more efficient DPOR algorithm, that is optimal in the sense that it explores precisely one execution in each equivalence class induced by po and rf.

In this paper, we present a fundamentally new approach to defining DPOR algorithms, which optimally explores only the equivalence classes defined by the program order and read-from relations. Our method is developed for the Release-Acquire (RA) fragment [Lahav et al. 2016] of the C/C++11 memory model. RA is a useful and well-behaved fragment of the C/C++11 memory model, which strikes a good balance between performance and programmability. In the RA semantics, all writes are `release` accesses, while all reads are `acquire` accesses. RA allows high-performance implementations, while still providing sufficiently strong guarantees for fundamental concurrent algorithms (such as the read-copy-update mechanism [Lahav et al. 2016]). Our DPOR algorithm is based on the above weakening of Shasha-Snir traces, called *weak traces*, which are defined by only the program order and read-from relations of an execution. For example, the program in Figure 1a has three weak traces, shown as τ_3, τ_4, and τ_5. Our DPOR algorithm is provably optimal for weak traces, in the sense that it explores *precisely* one execution for each weak trace that is *RA-consistent*, i.e., that can be extended with some coherence relation that satisfies the constraints of the RA semantics.

A significant challenge for our DPOR algorithm is to efficiently determine those continuations of a currently explored trace that lead to some RA-consistent trace. E.g., for the program in Figure 1a, letting both threads read from the write of the other thread leads to RA-inconsistency. We solve this problem by defining a *saturation* operation, which extends a weak trace with a *partial* coherence relation, which contains precisely those coherence edges that must be present in any corresponding Shasha-Snir trace. During exploration, the DPOR algorithm maintains a saturated version of the currently explored weak trace, and can therefore examine *precisely* those weak traces that are RA-consistent, without performing useless explorations. When a read event is added, the algorithm determines the set of write events from which it can obtain its value while preserving RA-consistency, and branches into a separate continuation for each such write event. When a write event is added, the algorithm merely adds it to the trace. It can be proven that this preserves RA-consistency, and also keeps the trace saturated (a slight modification is needed for atomic read-modify-write events). The algorithm must also detect if some previous read may read from a newly added write, and then backtrack to allow the write to be performed before that read.

We prove that our DPOR algorithm does not perform any useless work, in the sense that (i) any exploration eventually leads to a terminated RA-consistent execution, i.e., the algorithm never

[1] A Shasha-Snir trace also includes events that write initial values, but these can be ignored for this example.

blocks because it discovers that it is about to perform redundant or wasted explorations, and (ii) each RA-consistent weak trace is explored precisely once. Our saturation technique and the DPOR algorithm presented in this paper can be extended to cover atomic read-modify-write (RMW) operations and locks (space does not permit inclusion in this paper). This extension also satisfies the same strong optimality results (in (i) and (ii)).

In summary, we prove a stronger notion of optimality than related SMC approaches: we prove optimality concerning weak traces, whereas others only prove it w.r.t. total traces (i.e., Mazurkiewicz traces or Shasha-Snir traces); moreover, our technique can be extended to cover RMWs.

We have implemented our saturation operation and DPOR algorithm in a tool, called Tracer, and applied it to many challenging benchmarks. We compare our tool with other state-of-the-art stateless model checking tools running under the RA semantics. The experiments show that Tracer always generates optimal numbers of executions w.r.t. weak traces in all benchmarks. On many benchmarks, this number is much smaller than the ones produced by the other tools. The results also show that Tracer has better performance and scales better for more extensive programs, even in the case where it explores the same number of executions as the other tools.

2 BASIC NOTATION

We let $\mathbb{N}$ denote the set of natural numbers. Fix a set A. If A is finite then we use $|A|$ to denote the size of A. For a binary relation R on A, we write $a_1 \ [R] \ a_2$ to denote that $\langle a_1, a_2 \rangle \in R$. We use R^{-1} to denote the inverse of R, i.e. $a_1 \ [R^{-1}] \ a_2$ iff $a_2 \ [R] \ a_1$. We use R^+ and R^* to denote the transitive closure and the reflexive transitive closure of R, respectively. We write $a_1 \ [R]^+ a_2$ and $a_1 \ [R]^* a_2$ to denote that $a_1 \ [R^+] \ a_2$ resp. $a_1 \ [R^*] \ a_2$. We say that R is a partial order if it is irreflexive (i.e., $\neg(a \ [R] \ a)$ for all $a \in A$) and transitive (i.e., if $a_1 \ [R] \ a_2$ and $a_2 \ [R] \ a_3$ then $a_1 \ [R] \ a_3$ for all $a_1, a_2, a_3 \in A$). We say that R is total if, for all $a_1, a_2 \in A$, either $a_1 \ [R] \ a_2$ or $a_2 \ [R] \ a_1$. We use $\texttt{acyclic}(R)$ to denote that R is acyclic, i.e., there is no $a \in A$ such that $a \ [R]^+ \ a$. For a set $B \subseteq A$, we define $R|_B := R \cap (B \times B)$, i.e., it is the restriction of R to B. For binary relations R_1 and R_2 on A, we use $R_1; R_2$ to denote the composition of R_1 and R_2, i.e., $a_1 \ [R_1; R_2] \ a_2$ iff there is an $a_3 \in A$ such that $a_1 \ [R_1] \ a_3$ and $a_3 \ [R_2] \ a_2$. For sets A and B, we use $f : A \to B$ to denote that f is a (possibly partial) function from A to B. We use $f[a \leftarrow a']$ to denote the function f' such that $f'(a) = a'$ and $f'(b) = f(b)$ if $b \neq a$. We use A^* to denote the set of finite words over A, and use ϵ to denote the empty word. We use $|w|$ to denote the length of w, use $w[i]$ to denote the i^{th} element of w, and use $last(w)$ for $w[|w|]$. For words $w_1, w_2 \in A^*$, we use $w_1 \bullet w_2$ to denote the concatenation w_1 and w_2.

3 MODEL

Programs. We consider a program $\mathcal{P}$ consisting of a finite set $\mathcal{T}$ of *threads* (*processes*) 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, $\mathsf{a}, \mathsf{b}, \mathsf{c}$ for registers, and e for expressions. *Global statements* are either write $x := e$ to a shared variable or read $\mathsf{a} := x$ from a shared variable. *Local statements* only access and affect the local state of the thread and include assignments $\mathsf{a} := e$ 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. For readability reason, we do not consider atomic read-modify-write (RMW) operations and fences. The local state of a thread $th \in \mathcal{T}$ is defined as usual by its program counter and the contents of its registers.

Configurations. A *configuration (global state)* of $\mathcal{P}$ is made up of the local states of all the threads. Note that the values of the shared variables are not part of a configuration. The reason is that, under the RA semantics (and many other weak memory models), different threads may have different "local views" of the shared variables, i.e., they may see different values of the shared variables at a given point during the program execution. Therefore, it is not possible to assign a unique value to a shared variable. In existing operational semantics for such weak memory models including the RA semantics (e.g., [Kaiser et al. 2017]), a write instruction does not explicitly modify the values of the shared variables. Instead, a write instruction is added to a "pool" of write instructions that have been issued by the threads. A read instruction can fetch its value from a set of available write instructions in the pool and update its local view accordingly. The sets of write instructions that are available to reading threads depend on the particular memory model.

Following Lahav et al. [2016], we define an operational semantics for $\mathcal{P}$ as a labeled transition relation over configurations. Each transition corresponds to one thread performing a local or global statement. A transition between two configurations γ and γ' is of form $\gamma \overset{\ell}{\rightsquigarrow} \gamma'$, where the label ℓ describes the interaction with shared variables. The label ℓ is one of three forms: (i) $\langle th, \varepsilon \rangle$, indicating a local statement performed by thread th, which updates only the local state of th, (ii) $\langle th, \mathtt{W}, x, v \rangle$, indicating a write of the value v to the variable x by the thread th, which also updates the program counter of th, and (iii) $\langle th, \mathtt{R}, x, v \rangle$ indicating a read of v from x by the thread th into some register, while also updating the program counter of th. Observe that since the shared variables are not part of a configuration, the threads do not interact with each other in the transition relation. In particular, there is no constraint on the values that are used in transitions corresponding to read statements. This will obviously make illegal program behaviors possible. We remedy this problem by associating runs with so-called *traces*, which (among other things) represent how reads obtain their values from writes. A particular memory semantics (such as the RA semantics) is formulated by imposing restrictions on these traces, thereby also restricting the possible runs that are associated with them.

Since local statements are not visible to other threads, we will not represent them explicitly in the transition relation considered in our DPOR algorithm. Instead, we let each transition represent the combined effect of some finite sequence of local statements by a thread followed by a global statement by the same thread. More precisely, for configurations γ and γ' and a label ℓ which is either of the form $\langle th, \mathtt{W}, x, v \rangle$ or of the form $\langle th, \mathtt{R}, x, v \rangle$, we let $\gamma \xrightarrow{\ell} \gamma'$ denote that we can reach γ' from γ by performing a sequence of transitions labeled with $\langle th, \varepsilon \rangle$ followed by a transition labeled with ℓ. Defining the relation $\rightarrow$ in this manner ensures that we take the effect of local statements into account while avoiding consideration of interleavings of local statements of different threads in the analysis. Such optimization is common in tools (e.g., Verisoft [Godefroid 1997]).

We introduce some extra notation. We use $\gamma \xrightarrow{\langle th, \mathtt{R}, x, * \rangle} *$ to denote that $\gamma \xrightarrow{\langle th, \mathtt{R}, x, v \rangle} \gamma'$ for some value $v \in \mathbb{V}$ and configuration γ'. We use $\gamma \rightarrow \gamma'$ to denote that $\gamma \xrightarrow{\ell} \gamma'$ for some ℓ and define $\mathsf{succ}(\gamma) := \{\gamma' \mid \gamma \rightarrow \gamma'\}$, i.e., it is the set of successors of γ w.r.t. $\rightarrow$.

A configuration γ is said to be *terminal* if $\mathsf{succ}(\gamma) = \varnothing$, i.e., no thread can execute a global statement from γ. A *run* ρ from γ is a sequence $\gamma_0 \xrightarrow{\ell_1} \gamma_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} \gamma_n$ such that $\gamma_0 = \gamma$. We say that ρ is *terminated* if γ_n is terminal. We let $\mathit{Runs}(\gamma)$ denote the set of runs from γ.

Events. An event corresponds to a particular execution of a statement in a run of $\mathcal{P}$. A *write event* e is a tuple $\langle id, th, \mathtt{W}, x, v \rangle$, where $id \in \mathbb{N}$ is an event identifier, $th \in \mathcal{T}$ is a thread, $x \in \mathbb{X}$ is a variable, and $v \in \mathbb{V}$ is a value. This event corresponds to thread th writing the value v to variable x. The identifier id denotes that th has executed $id-1$ events before e in the corresponding run. A *read event* e is a tuple $\langle id, th, \mathtt{R}, x \rangle$, where id, th, and x are as for a write event. This event

corresponds to thread th reading some value to x. Note that a read event e does not specify the particular value it reads. This value will be defined in a trace by specifying a write event from which e fetches its value. For an event e of form $\langle id, th, \mathtt{W}, x, v\rangle$ or $\langle id, th, \mathtt{R}, x\rangle$, we define $e.id := id$, $e.thread := th$, $e.type := \mathsf{t}$ where $\mathsf{t} \in \{\mathtt{W}, \mathtt{R}\}$, $e.var := x$, and $e.val := v$ (the latter is not defined for a read event). For each variable $x \in \mathbb{X}$, we assume a special write event $\mathtt{init}_x = \langle -, -, \mathtt{W}, x, 0\rangle$, called the *initializer* event for x. This event is not performed by any of the threads in $\mathcal{T}$, and writes the value 0 to x. We define $\mathsf{E}_{init} := \{\mathtt{init}_x \mid x \in \mathbb{X}\}$ as the set of initializer events.

If E is a set of events, we define subsets of E characterized by particular attributes of its events. For instance, for a thread th, we let $\mathsf{E}^{th,\mathtt{W}}$ denote $\{e \in \mathsf{E} \mid e.thread = th \wedge e.type = \mathtt{W}\}$.

Traces. A *trace* τ is a tuple $\langle \mathsf{E}, \mathsf{po}, \mathtt{rf}, \mathsf{co}\rangle$, where E is a set of *events* which includes the set E_{init} of initializer events, and where po (program order), $\mathtt{rf}$ (read-from), and co (coherence order) are binary relations on E that satisfy:

- $e\ [\mathsf{po}]\ e'$ if $e.thread = e'.thread$ and $e.id < e'.id$, i.e., po totally orders the events of each individual thread. As mentioned above, each event corresponds to an execution of a program statement. The program order then reflects the order in which the statements of a given thread are executed. Note that po does not relate the initializer events.
- $e\ [\mathtt{rf}]\ e'$ if e is a write event and e' is a read event on the same variable, which obtains its value from e. The inverse of relation $\mathtt{rf}$, denoted $\mathtt{rf}^{-1}$, must be a total function on $\mathsf{E}^{\mathtt{R}}$. We sometimes view $\mathtt{rf}$ as the union of a read-from relation $\mathtt{rf}^x$ for each variable $x \in \mathbb{X}$.
- co is a union $\mathsf{co} = \cup_{x\in\mathbb{X}}\mathsf{co}^x$, where co^x is a relation on $\mathsf{E}^{\mathtt{W},x}$ (including $\mathtt{init}_x$), subject to the constraint that $\mathtt{init}_x$ is before all other write events in $\mathsf{E}^{\mathtt{W},x}$. Thus co does not relate write events on different variables. The relation co^x reflects how the threads view the order on the write events on x. If $e_1\ [\mathsf{co}^x]\ e_2$ then all threads share the view that e_1 has occurred before e_2.

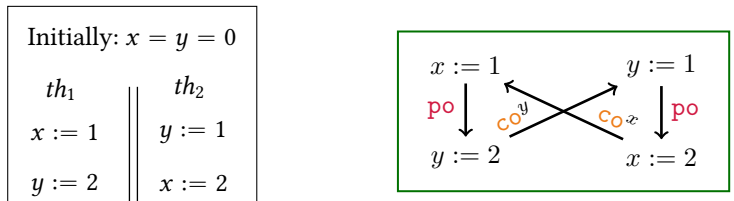
Note that (in contrast to po), each co^x can be an arbitrary relation; it need not even be transitive. We say that a trace $\langle \mathsf{E}, \mathsf{po}, \mathtt{rf}, \mathsf{co}\rangle$ is *total* if co^x is a strict total order on $\mathsf{E}^{\mathtt{W},x}$ for each $x \in \mathbb{X}$. We use $\mathtt{total}(\tau)$ to denote that τ is total. A total trace is also called a *Shasha-Snir trace*. We sometimes use *partial trace* to denote an arbitrary trace, when we want to emphasize that it need not be total.

As depicted in Figure 1b, we can view $\tau = \langle \mathsf{E}, \mathsf{po}, \mathtt{rf}, \mathsf{co}\rangle$ as a graph whose nodes are E and whose edges are defined by the relations po, $\mathtt{rf}$, and co. We define $|\tau| := |\mathsf{E}|$, i.e., it is the number of events in τ. We define the *empty trace* $\tau_\varnothing := \langle \mathsf{E}_{init}, \varnothing, \varnothing, \varnothing\rangle$, i.e., it contains only the initializer events, and all the relations are empty.

Associating Traces with Runs. We can now define when a trace can be associated with a run. Consider a run ρ of form $\gamma_0 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n} \gamma_n$, where $\ell_i = \langle th_i, \mathsf{t}_i, x_i, v_i\rangle$, and let $\tau = \langle \mathsf{E}, \mathsf{po}, \mathtt{rf}, \mathsf{co}\rangle$ be a trace. We write $\rho \models \tau$ to denote that the following conditions are satisfied:

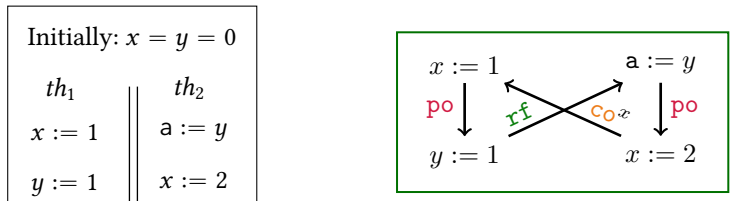
- $\mathsf{E}\backslash\mathsf{E}_{init} = \{e_1, \ldots, e_n\}$, i.e., each non-initializer event corresponds exactly to one label in ρ.
- If $\ell_i = \langle th_i, \mathtt{W}, x_i, v_i\rangle$, then $e_i = \langle id_i, th_i, \mathtt{W}, x_i, v_i\rangle$, and if $\ell_i = \langle th_i, \mathtt{R}, x_i, v_i\rangle$, then $e_i = \langle id_i, th_i, \mathtt{R}, x_i\rangle$. An event and the corresponding label perform identical operations (write or read) on the same variables. In the case of a write, they also agree on the written value.
- $id_i = |\{j \mid (1 \leqslant j \leqslant i) \wedge (th_j = th_i)\}|$. The identifier of an event shows how it is ordered relative to the other events performed by the same thread.
- If $e_i\ [\mathtt{rf}]\ e_j$, then $x_i = x_j$ and $v_i = v_j$. A read event fetches its value from an event that writes the same value to the same variable.
- If $\mathtt{init}_x\ [\mathtt{rf}]\ e_i$, then $v_i = 0$, i.e., e_i reads the initial value of x which is assumed to be 0.

Release-Acquire Semantics. Following Lahav et al. [2016], we define the Release-Acquire semantics by defining the set of runs whose associated total traces do not contain certain forbidden cycles.



(a) (b)

Fig. 2. (a) The program 2+2W and (b) a Shasha-Snir trace.



(a) (b)

Fig. 3. (a) The program S and (b) a Shasha-Snir trace.

Given a trace $\tau = \langle \mathsf{E}, \mathtt{po}, \mathtt{rf}, \mathtt{co} \rangle$, we define the derived relation $\mathtt{fr}$ (from-read) by $\mathtt{fr} := \cup_{x \in \mathbb{X}} \mathtt{fr}^x$, where $\mathtt{fr}^x := (\mathtt{rf}^x)^{-1}; \mathtt{co}^x$. Intuitively, if $e \ [\mathtt{fr}^x] \ e'$ then the write e' overwrites the value read by the read e (since e' is coherence-order-after the write event from which e gets its value).

Definition 3.1. For a trace τ, let $\tau \models \text{RA}$ denote that the relation $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}^x \cup \mathtt{fr}^x$ is acyclic for each $x \in \mathbb{X}$. We define $[\![\gamma]\!]^{\mathtt{Total}}_{\text{RA}} := \{\tau \mid \exists \rho \in \mathit{Runs}(\gamma).\ \rho \models \tau \wedge \mathtt{total}(\tau) \wedge \tau \models \text{RA}\}$, i.e., $[\![\gamma]\!]^{\mathtt{Total}}_{\text{RA}}$ is the set of total traces generated under RA from a given configuration γ.

To illustrate the RA semantics, Figure 2a shows a simple program, known as 2+2W [Alglave et al. 2014], with two threads th_1 and th_2 that communicate through two shared variables x and y. Thread th_1 writes 1 to x and then 2 to y. Symmetrically, th_2 writes 1 to y and then 2 to x. We would like to check whether the writes $x := 2$ and $y := 2$ can be placed before the writes $x := 1$ and $y := 1$ in the corresponding coherence order relation (co). Figure 2b gives the corresponding trace τ_1. To improve readability, we use a simplified notation for events. More precisely, we represent an event in a trace by the corresponding program instruction. For instance, we write $x := 1$ instead of $\langle 1, th_1, \mathtt{W}, x, 1 \rangle$. We observe that the read-from relation $\mathtt{rf}$ and the from-read relation $\mathtt{fr}$ are empty in τ_1. Since the relations $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}^x \cup \mathtt{fr}^x$ and $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}^y \cup \mathtt{fr}^y$ are acylic, it follows by Definition 3.1, that $\tau_1 \models \text{RA}$.

To see the role of read-from relations in the RA semantics, Figure 3a gives another program, known as S [Alglave et al. 2014]. In a similar manner to 2+2W, the program S has two threads th_1 and th_2 that communicate through two shared variables x and y. Thread th_1 writes 1 to both x and y. Thread th_2 reads from y and stores the value to a local register a and then writes 2 to x. We would like to check whether it is possible to place the write $x := 2$ before the write $x := 1$ in coherence order (co) and make $\mathtt{a} := y$ read from the write $y := 1$ in the read-from relation ($\mathtt{rf}$). Figure 2b gives the corresponding trace τ_2. Since the relation $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}^x \cup \mathtt{fr}^x$ is cyclic, it follows by Definition 3.1, that $\tau_2 \not\models \text{RA}$.

Since the result computed by a run is uniquely determined by its associated trace, we can analyze a concurrent program under the RA semantics by exploring runs and their associated traces until all traces in $[\![\gamma]\!]^{\mathtt{Total}}_{\mathrm{RA}}$ have been generated. One can even define an algorithm which is optimal in that it explores each trace in $[\![\gamma]\!]^{\mathtt{Total}}_{\mathrm{RA}}$ exactly once [Kokologiannakis et al. 2018]. However, it is possible to be more efficient than this, by exploiting the observation that the results computed in a run, including, e.g., the outcome of assert statements, depend only on the program order (po) and read-from (rf) relations, irrespective of the co relation. In fact, the sequence of local states of a thread and the values it writes to shared variables are affected only by the sequence of values it reads from shared variables. Hence two runs, whose associated traces have the same po and rf relations, can be regarded as equivalent. An algorithm based on the set $[\![\gamma]\!]^{\mathtt{Total}}_{\mathrm{RA}}$ will in general be redundant, in that it explores several traces that are equivalent in this sense.

Weak Traces. The above observation suggests to base the analysis on a weaker notion of trace. Define a *weak trace* to be a trace whose co relation is empty. For a trace $\tau = \langle \mathsf{E}, \mathsf{po}, \mathtt{rf}, \mathsf{co} \rangle$, we define its *weakening* by $\mathit{weak}(\tau) := \langle \mathsf{E}, \mathsf{po}, \mathtt{rf}, \emptyset \rangle$, obtained by removing all coherence edges. We define the set $[\![\gamma]\!]^{\mathtt{Weak}}_{\mathrm{RA}} := \left\{ \mathit{weak}(\tau) \mid \tau \in [\![\gamma]\!]^{\mathtt{Total}}_{\mathrm{RA}} \right\}$. In other words, $[\![\gamma]\!]^{\mathtt{Weak}}_{\mathrm{RA}}$ is the set of weakenings of the traces in the total semantics. Let us introduce a term for such weakenings. For traces $\tau = \langle \mathsf{E}, \mathsf{po}, \mathtt{rf}, \mathsf{co} \rangle$ and $\tau' = \langle \mathsf{E}', \mathsf{po}', \mathtt{rf}', \mathsf{co}' \rangle$, let $\tau \sqsubseteq \tau'$ denote that $\mathsf{E} = \mathsf{E}'$, $\mathsf{po} = \mathsf{po}'$, $\mathtt{rf} = \mathtt{rf}'$ and $\mathsf{co}^x \subseteq (\mathsf{co}')^x$ for each $x \in \mathbb{X}$. We say that a (partial) trace τ is *RA-consistent* if there is a total trace τ' with $\tau \sqsubseteq \tau'$ such that $\tau' \models \mathrm{RA}$. In particular, a total trace τ is RA-consistent if and only if $\tau \models \mathrm{RA}$. Also, $[\![\gamma]\!]^{\mathtt{Weak}}_{\mathrm{RA}}$ is the set of RA-consistent weak traces of γ.

We can now analyze a program by exploring runs until all weak traces in $[\![\gamma]\!]^{\mathtt{Weak}}_{\mathrm{RA}}$ have been generated. In the next sections, we will present such an analysis algorithm, which is also *optimal* in the sense that it explores each trace in $[\![\gamma]\!]^{\mathtt{Weak}}_{\mathrm{RA}}$ exactly once. Such an algorithm must overcome the challenges of (i) *consistency*, i.e., examining only RA-consistent weak traces (note that many weak traces are not RA-consistent; e.g., for the program in Figure 1a, letting both threads read from the write of the other thread is not possible under the RA semantics), and (ii) *non-redundancy*, i.e., to generate each weak trace only once, thereby avoiding unnecessary explorations. We solve the consistency challenge in §4, by defining a *saturated semantics*, equivalent with the standard one, in which runs are associated with *partial* traces that contain precisely those coherence edges that must be present in any corresponding total trace. Based on the saturated semantics, the non-redundancy challenge is solved by the design of our DPOR algorithm in §5.

4 SATURATED SEMANTICS

In this section, we address the challenge of exploring only RA-consistent trace by defining a new semantics for RA, called the *saturated semantics*. The saturated semantics is the basis for our DPOR algorithm in §5, which generates precisely the weak traces in $[\![\gamma]\!]^{\mathtt{Weak}}_{\mathrm{RA}}$ for a given configuration γ.

The saturated semantics solves the consistency challenge by making it easy to maintain RA-consistency. We define the semantics in two steps. First, in §4.1, we define the notion of a *saturated trace* as a partial trace which extends a weak trace, whose (partial) coherence relation contains precisely the edges that occur in all RA-consistent total extensions of that weak trace. We show that a saturated trace is RA-consistent iff it does not contain a cycle that is forbidden by the RA semantics (cf. Theorem 4.2). Then, in §4.2, we present two efficient operations that allow adding a new write (resp. read) event to a trace while preserving both saturation and RA-consistency. We use these operations as the basis to define our saturated semantics. Finally, we show a key theorem (cf. Theorem 4.8) that the saturated semantics coincides with the RA semantics on weak traces.

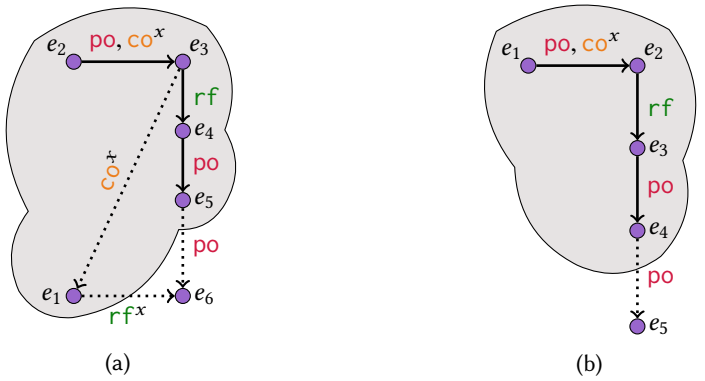


Fig. 4. (a) Adding a read event e_6 and (b) a write event e_5. The dotted arrows are added due to the new event.

4.1 Saturated Traces

Definition 4.1. A trace τ is *saturated* if, for all variables $x \in \mathbb{X}$, whenever $e, e' \in \mathsf{E}^{\mathsf{W},x}$ with $e \neq e'$ such that $e\ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x]^+\ e''$ and $e'\ [\mathsf{rf}^x]\ e''$, then $e\ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x]^+\ e'$.

To motivate this definition, note that whenever $e\ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x]^+\ e''$ and $e'\ [\mathsf{rf}^x]\ e''$ then (under RA) any coherence edge between e and e' must (if present) be directed from e to e'; otherwise $e'\ [\mathsf{co}^x]\ e$ and $e'\ [\mathsf{rf}^x]\ e''$ would imply $e''\ [\mathsf{fr}^x]\ e$, implying $e\ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x \cup \mathsf{fr}^x]^+\ e$ (by $e\ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x]^+\ e''\ [\mathsf{fr}^x]\ e$), thereby violating the RA semantics.

The following theorem shows an essential property of saturated traces, namely that if such a trace does not contain cycles violating the RA semantics then it is RA-consistent.

Theorem 4.2. *For a partial trace τ, if τ is saturated and $\tau \models \mathrm{RA}$, then τ is RA-consistent.*

Proof. Assume that $\tau = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co} \rangle$ is saturated and $\tau \models \mathrm{RA}$. We show that there is a total trace $\tau' = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co}' \rangle$ such that $\tau \sqsubseteq \tau'$ and $\tau' \models \mathrm{RA}$. The lemma then follows immediately.

We define a sequence of traces $\tau_0 \sqsubseteq \tau_1 \sqsubseteq \tau_2, \ldots$ inductively such that $\tau_0 = \tau$, τ_i is saturated and $\tau_i \models \mathrm{RA}$. For $i > 0$, if $\tau_i = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co}_i \rangle$ is not total then τ_{i+1} is derived from τ_i by adding the pair $\langle e_i, e_i' \rangle$ to the coherence order, where $e_i, e_i' \in \mathsf{E}^{\mathsf{W},x}$ for some $x \in \mathbb{X}$ with $\neg(e_i'\ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}_i^x]^+\ e_i)$ and $\neg(e_i\ [\mathsf{co}_i^x]\ e_i')$. Such events e_i and e_i' exist since τ_i is not total and since $\tau_i \models \mathrm{RA}$ by the induction hypothesis. Also, by the induction hypothesis we know that τ_i is saturated. It follows that τ_{i+1} is saturated. Since $\tau_i \models \mathrm{RA}$ it follows that $\tau_{i+1} \models \mathrm{RA}$. Notice that, by construction, we have $|\tau_i| < |\tau_{i+1}|$. It follows that there is a j such that τ_j is total. Define $\tau' := \tau_j$. This concludes the proof the the lemma. □

4.2 Saturated Semantics

Next, we introduce two notions that are relevant when adding a new read event to a trace.

4.2.1 Readability and Visibility. (i) *Readability* identifies the write events e' from which read event e can fetch its value, and *visibility* is used to add new coherence-order edges that are implied by the fact that the new event e reads from e'. Below, we fix a trace $\tau = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co} \rangle$.

Definition 4.3. For a thread $th \in \mathcal{T}$ and a variable $x \in \mathbb{X}$, the set of *readable events* $\mathcal{R}(\tau, th, x)$ is defined to be the set of events $e \in \mathsf{E}^{\mathsf{W},x}$ such that there are no events $e' \in \mathsf{E}^{\mathsf{W},x}$ and $e'' \in \mathsf{E}^{th}$ with $e\ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x]^+\ e'$ and $e'\ [\mathsf{po} \cup \mathsf{rf}]^*\ e''$.

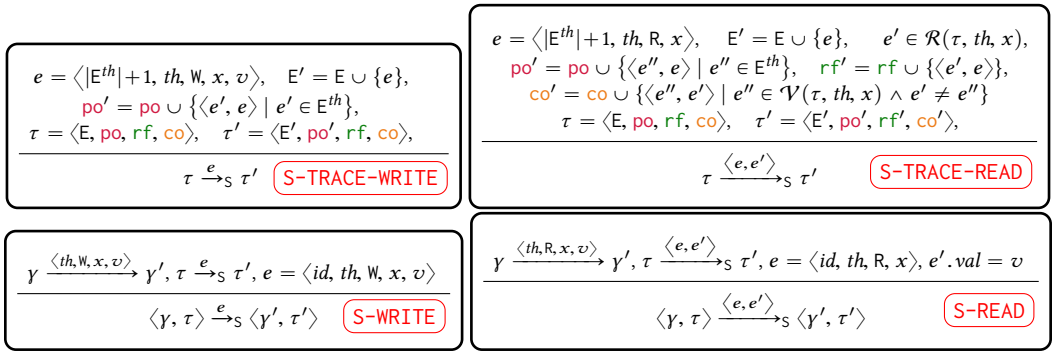


Fig. 5. The saturated transition relation.

Intuitively, $\mathcal{R}(\tau, th, x)$ contains all write events on x that are not hidden from thread th by other write events on x. A new read event on x that is added to th can fetch its value from any write event in $\mathcal{R}(\tau, th, x)$. In fact, if th is saturated then $\mathcal{R}(\tau, th, x)$ is precisely the set of write events from which e can read without introducing a cycle that is forbidden by the RA semantics.

To illustrate this, let τ be as in Figure 4a, where the read e_6 is about to be added, and let e_2, e_3, and e_1 be write events on x. The dotted arrows (explained later) represent edges that will be added to the traces due to the new event e_6. Let th be the thread of e_5 and e_6. Then $e_1, e_3 \in \mathcal{R}(\tau, th, x)$, while $e_2 \notin \mathcal{R}(\tau, th, x)$. Note that reading from e_2 would destroy RA-consistency, since any corresponding total trace must have e_2 $[\mathsf{co}^x]$ e_3 and hence e_6 $[\mathsf{fr}^x]$ e_3, inducing the cycle e_6 $[\mathsf{fr}^x]$ e_3 $[\mathsf{rf}]$ e_4 $[\mathsf{po}]$ e_6 thereby violating the RA semantics.

Definition 4.4. For a thread $th \in \mathcal{T}$ and a variable $x \in \mathbb{X}$, the *visible events* $\mathcal{V}(\tau, th, x)$ is defined to be the set of events e in $\mathcal{R}(\tau, th, x)$ such that there is an $e' \in \mathsf{E}^{th}$ with e $[\mathsf{po} \cup \mathsf{rf}]^*$ e'.

In other words, the set contains all the readable events on x that can reach an event in th through po and rf edges. The point of $\mathcal{V}(\tau, th, x)$ is that if a new read event e on x is added to th, which reads from an event e' then the resulting trace is saturated by adding a coherence-order edge from each $e'' \in \mathcal{V}(\tau, th, x)$ to e'. As illustration, in Figure 4a (again before adding e_6), we have $e_3 \in \mathcal{V}(\tau, th, x)$ while $e_2 \notin \mathcal{V}(\tau, th, x)$. This means that if we let e_6 read from e_1 then the saturation of the resulting trace adds a coherence-order edge from e_3 to e_1; on the other hand, no edge is added from e_2 to e_1.

4.2.2 Saturated Semantics. We define the saturated semantics as a transition relation $\rightarrow_{\mathsf{S}}$ on traces (the first two rules of Figure 5). The transition rules correspond to adding a new write or read event to a trace τ and obtaining a new trace τ'. Each transition is labeled by an *observation* α which is either a write event e, or a pair $\langle e, e'\rangle$, consisting of a read event e that reads from a write event e'. We define $\alpha.event := e$. As we will explain below, an important property of the transition relation is that if the original trace τ is saturated and has no cycles forbidden by the RA-semantics, then the new trace τ' will satisfy the same conditions. Therefore, by Theorem 4.2 it follows that all traces generated according to the saturated semantics are RA-consistent. Indeed, the semantics generates precisely those traces that are saturated and contain no cycles violating the RA semantics.

Rule S-TRACE-WRITE describes that a saturated trace is extended with a write event e by (i) adding a new write event e, whose identity reflects that it is the most recent event performed by its thread, (ii) making e last in the program order of its thread, and (iii) keeping the read-from and coherence relations. We observe that if τ is saturated and $\tau \models \text{RA}$, then τ' will be saturated and $\tau' \models \text{RA}$. The

reason is that e does not have any successors w.r.t. $(\mathsf{po}' \cup \mathsf{rf} \cup \mathsf{co})^+$ in τ', and hence the only cycles in τ' are those that are already in τ. Since no cycle in τ violates the RA semantics by assumption, no cycle in τ' will break the RA semantics either. Figure 4b illustrates how to apply the rule to extend a saturated trace with a new write event e_5.

Rule S-TRACE-READ describes that a saturated trace is extended with a read event e that reads from a write event e' by (i) adding the new read event e, (ii) ensuring that the write event e' is readable for the thread th w.r.t. the variable x in τ, (iii) extending the program order in the same manner as in S-TRACE-WRITE, (iv) extending the read-from relation to reflect that e reads from e', and (v) extending the coherence order by an edge from each visible event to e'. If τ is saturated then τ' will also be saturated since we add all the additionally needed coherence-order edges, namely the ones from the events in $\mathcal{V}(\tau, th, x)$ to e'. Furthermore, if τ is saturated and $\tau \models \text{RA}$ then $\tau' \models \text{RA}$. The reason is that any new cycle violating the RA semantics in τ' would include an edge from an event $e'' \in \mathcal{V}(\tau, th, x)$ to e'. However, such a cycle implies that $e' \ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x]^+ \ e''$, i.e., $e' \notin \mathcal{R}(\tau, th, x)$ which is a contradiction. Figure 4a illustrates how to apply the rule to extend a saturated trace with a new read event e_6.

The rules S-WRITE and S-READ describe how the transition relation $\rightarrow_{\mathsf{S}}$ on saturated traces induces a corresponding transition relation on pairs of configurations and saturated traces, in the natural way. We use $\langle \gamma, \tau \rangle \rightarrow_{\mathsf{S}} \langle \gamma', \tau' \rangle$ to denote that $\langle \gamma, \tau \rangle \xrightarrow{\alpha}_{\mathsf{S}} \langle \gamma', \tau' \rangle$ for some α, and use $\xrightarrow{*}_{\mathsf{S}}$ to denote the reflexive transitive closure of $\rightarrow_{\mathsf{S}}$. We define $\mathsf{succ}_{\mathsf{S}}(\gamma, \tau) := \{\langle \gamma', \tau' \rangle \mid \langle \gamma, \tau \rangle \rightarrow_{\mathsf{S}} \langle \gamma', \tau' \rangle\}$, i.e., it is the set of successors of the pair $\langle \gamma, \tau \rangle$ w.r.t. $\rightarrow_{\mathsf{S}}$.

4.2.3 Properties of the Saturated Transition Relation. We describe three properties of the saturated semantics, namely efficiency, deadlock-freedom, and correctness.

Efficiency. The sets $\mathcal{R}(\tau, th, x)$ and $\mathcal{V}(\tau, th, x)$ can both be computed in polynomial time. To see this, suppose we are given a $\tau = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co} \rangle$, a thread th, and a variable x. A polynomial time algorithm for computing the set $\mathcal{R}(\tau, th, x)$ can be defined consisting of the following three steps:

(1) Compute the transitive closure of the relation $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x$ using, e.g., the Floyd-Warshall algorithm [Cormen et al. 2009]. This will cost $O(|\mathsf{E}|^3)$ time.
(2) Compute the set of events e' such that $e' \ [\mathsf{po} \cup \mathsf{rf}]^* \ e''$ for some $e'' \in \mathsf{E}^{th}$. This will cost $O(|\mathsf{E}|^2)$ time.
(3) For each event $e \in \mathsf{E}^{\mathsf{W},x}$, check whether there is an event e' in the set computed in step (ii) with $e \ [\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}^x]^+ \ e'$. If not, add the event e to the set $\mathcal{R}(\tau, th, x)$. This will cost $O(|\mathsf{E}|^2)$ time.

The set $\mathcal{V}(\tau, th, x)$ can be computed similarly. This gives the following lemma.

Lemma 4.5. *For a trace τ, a thread th, and a variable x, we can compute the sets $\mathcal{R}(\tau, th, x)$ and $\mathcal{V}(\tau, th, x)$ in polynomial time.*

By Lemma 4.5 it follows that we can compute each step of $\rightarrow_{\mathsf{S}}$ in polynomial time. This property is not satisfied by all memory models. For instance, calculating the successors of a trace in the SC semantics amounts to solving an NP-complete problem [Chalupa et al. 2018].

Deadlock-Freedom. The saturated semantics is deadlock free in the sense that if a thread can perform a transition from a configuration then there is always a corresponding move in the saturated semantics. This is captured by the following lemma (which follows immediately from the definitions).

Lemma 4.6. *For a configuration γ and trace τ, if $\tau \models \text{RA}$ and $\mathsf{succ}(\gamma) \neq \emptyset$ then $\mathsf{succ}_{\mathsf{S}}(\gamma, \tau) \neq \emptyset$.*

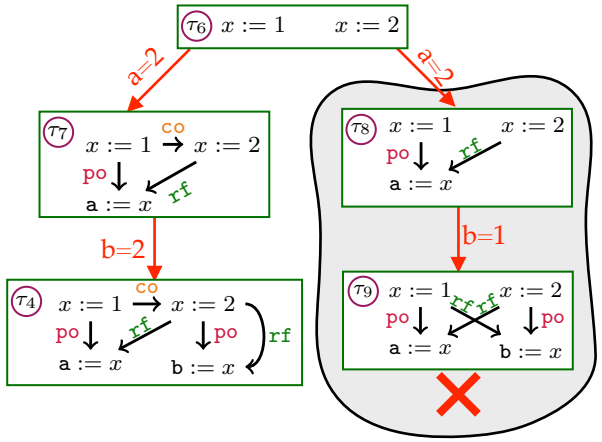


Fig. 6. Example illustrating the saturated semantics.

Correctness. The following lemma states the fact that the saturated semantics preserves saturation and RA-consistency.

Lemma 4.7. *If τ is saturated, $\tau \models \text{RA}$, and $\tau \xrightarrow{\alpha}_{\text{S}} \tau'$ then τ' is saturated and $\tau' \models \text{RA}$.*

Define $[\![\gamma]\!]^{\text{S}}_{\text{RA}} := \left\{ \tau \mid \exists \gamma'.\ \langle \gamma, \tau_\varnothing \rangle \xrightarrow{*}_{\text{S}} \langle \gamma', \tau \rangle \right\}$, i.e., it is the set of traces that can be generated by sequences of $\rightarrow_{\text{S}}$-transitions, starting from γ and the empty trace. The following theorem states that the RA semantics and the saturated semantics coincide on weak traces, i.e., $[\![\cdot]\!]^{\text{S}}_{\text{RA}}$ and $[\![\cdot]\!]^{\text{Total}}_{\text{RA}}$ generate the same set of weak traces.

Theorem 4.8. *For any configuration γ, we have $\left\{ \mathit{weak}(\tau) \mid \tau \in [\![\gamma]\!]^{\text{S}}_{\text{RA}} \right\} = [\![\gamma]\!]^{\text{Weak}}_{\text{RA}}$.*

4.3 Example

We will give an example (Figure 6) to illustrate why saturation is important in the semantics, and how the semantics preserves saturation of traces. To that end, we will consider the simple program in Figure 1a again. To make the presentation easier to read, we will use a simplified notation for events and observations. First, we represent an event in a trace by the corresponding program instruction. For instance, we will write $x := 1$ instead of $\langle 1, th_1, \mathtt{W}, x, 1 \rangle$, and we will write $\mathtt{a} := x$ instead of $\langle 2, th_1, \mathtt{R}, x \rangle$. We write an observation $\langle e, e' \rangle$ as $\mathtt{a} = 2$ if e corresponds to a read statement of the form $\mathtt{a} := x$ and e' corresponds to a write statement of the form $x := 2$. In this particular example, we can use this simplified notation since all the statements in the program are different.

Suppose that we have generated the trace τ_6 containing the write events $x := 1$ and $x := 2$, corresponding to th_1 and th_2 running their first instructions respectively. The trace τ_6 is trivially saturated since it does not contain any read events. The event $x := 1$ is visible to the thread th_1, and the event $x := 2$ is visible to the thread th_2. Both events are readable by both th_1 and th_2. Suppose that th_1 executes the read instruction $\mathtt{a} := x$ and it chooses to read from the event $x := 2$. This corresponds to performing the observation $\mathtt{a} = 2$. Since $x := 1$ is visible to th_1 our saturated semantics will add a co-edge from $x := 1$ to $x := 2$ which means that the resulting trace τ_7 is saturated. In τ_7, the only readable event by th_2 is $x := 2$ (since $x := 1$ is now hidden by $x := 2$). If th_2 performs the read instruction $\mathtt{b} := x$ then it can only read from the event $x := 2$, corresponding to performing the observation $\mathtt{b} = 2$ and leading to the trace τ_4 (the same trace as the one in

Figure 1b). Since the only visible event in τ_4 is $x := 2$, the semantics will not add any new co-edges. Notice that τ_4 is already saturated.

Next, we will show why saturation is crucial for the semantics. Let us return to the scenario where we are in τ_6 and th_1 performs its read instruction $\mathsf{a} := x$ and it chooses to read from the event $x := 2$. Suppose that we do not add the co-edge from $x := 1$ to $x := 2$ and thus obtain τ_8 which is not saturated. If th_2 performs the read instruction $\mathsf{b} := x$ then it cannot deduce from τ_8 that it cannot read from $x := 1$. However, if th_2 reads from $x := 1$, we obtain τ_9 which is not RA-consistent. The reason is that regardless of whether we put a co-edge from $x := 1$ to $x := 2$ or the opposite, we will obtain a cycle that is forbidden by the RA semantics.

Redundancy. While the saturated semantics generates only RA-consistent traces, it still suffers from the redundancy problem. More precisely, several runs of the program may produce the same weak trace. Consider again the simple program in Figure 1a. Consider two runs, namely ρ_1 where first th_1 executes $x := 1$ and then th_2 executes $x := 2$, and ρ_2 where first th_2 executes $x := 2$ and then th_1 executes $x := 1$. The runs ρ_1 and ρ_2 have the same trace, namely τ_6 in Figure 6, and hence exploring both of them in the analysis would be wasteful. While this particular scenario is quite simple, redundant explorations may have quite complex forms. The DPOR algorithm in §5 aims to obtain an optimal search, i.e., consider only one run per weak trace of the input program.

5 DPOR ALGORITHM

In this section, we present our DPOR algorithm. For a terminating program, it systematically explores the whole set of weak traces that can be generated according to the saturated semantics. The algorithm is sound, complete, and optimal in the sense that it produces each weak trace corresponding to a terminated run precisely once. We achieve optimality by combining the saturated semantics with an exploration algorithm ensuring that no two generated traces will have same program order and read-from relations. Moreover, the algorithm is deadlock-free in the sense that the exploration only ends at points when the considered program configuration is terminal. We will first give a detailed description of the algorithm in §5.1, then provide a complete example that illustrates the main ingredients of the algorithm in §5.3, and finally state its properties in §5.4.

5.1 Algorithm

The DPOR algorithm explores the program according to the saturated semantics using the main procedure DfVisit, starting with an input configuration, a trace, and a sequence of observations.

For a write event, the algorithm merely adds the event and continues with the next event. For a read event e, the algorithm also adds e, but it continues in several separate branches each corresponding to a different write event from which e can fetch its value. Besides, the algorithm must handle the case that e may read from a write event e' that will only be added to the trace later in the exploration. We say that e' has been *postponed* w.r.t. e. When e' is eventually generated, the DPOR algorithm will detect it and *swap* it with e, thus making it possible for e to read from e'. To swap e and e', we also need to include the sequence of observations that are "necessary" for generating e'. We will refer to such a sequence as a *schedule* (cf. the DeclarePostponed procedure, §5.1.2). All the generated schedules will eventually be executed thus swapping e with all the write events that are postponed w.r.t. it (cf. the RunSchedule procedure, §5.1.3).

5.1.1 DfVisit - the Main Procedure (Algorithm 1). The depth-first exploration is given in Algorithm 1. The $\textsc{DfVisit}(\gamma, \tau, \pi)$ procedure explores all RA-consistent weak traces of the program $\mathcal{P}$ that are generated from a configuration γ and a saturated trace τ, where γ and τ have been generated by performing a sequence of observations π. As we describe below, the sequence π is used for swapping read events with write events that are postponed w.r.t. them. Initially, the

Algorithm 1: DfVisit(γ, τ, π)

Input: γ is a configuration, $\tau = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co} \rangle$ is a trace, π is an observation sequence.

1 **if** $\exists \gamma \xrightarrow{\langle th, \mathtt{W}, x, v \rangle} \gamma'$ **then** // handle a write event
2 **let** e be $\langle |\mathsf{E}^{th}|+1, th, \mathtt{W}, x, v \rangle$ and τ' be such that $\langle \gamma, \tau \rangle \xrightarrow{e}_{\mathsf{S}} \langle \gamma', \tau' \rangle$ // follow S-WRITE
3 DfVisit$(\gamma', \tau', \pi \bullet e)$
4 DeclarePostponed$(\tau', \pi \bullet e)$
5 **else if** $\exists \gamma \xrightarrow{\langle th, \mathtt{R}, x, * \rangle} *$ **then** // handle a read event
6 **let** e be $\langle |\mathsf{E}^{th}|+1, th, \mathtt{R}, x \rangle$
7 $\mathtt{Schedules}(e) \leftarrow \emptyset$; $\mathtt{Swappable}(e) \leftarrow \mathtt{true}$
8 **for** $e', \gamma', \tau' : \langle \gamma, \tau \rangle \xrightarrow{\langle e, e' \rangle}_{\mathsf{S}} \langle \gamma', \tau' \rangle$ **do** DfVisit$(\gamma', \tau', \pi \bullet \langle e, e' \rangle)$ // follow S-READ
9 **for** $\beta \in \mathtt{Schedules}(e)$ **do** RunSchedule$(\gamma, \tau, \pi, \beta)$

procedure is called with an initial configuration γ, the empty trace $\tau_\emptyset$, and the empty observation sequence ϵ.

In a call, if γ has no successors, then DfVisit returns to its caller. Otherwise, it considers an enabled write or read statement. If a write statement is enabled, then one such write statement is selected non-deterministically, and the corresponding event e is created (at line 2). This event e is added to the trace according to the saturated semantics and also added to the sequence of observations, whereafter DfVisit is called recursively to continue the exploration (at line 3). After the recursive call has returned, the algorithm calls DeclarePostponed, which finds the read events e' in the input exploration sequence π, which would be able to read from the write e if e were performed before e'. For each such a read event e', DeclarePostponed creates a *schedule* for e', which is a sequence of observations that can be explored from the point where e' was performed, to let the write e occur before e' so that e' can read from e (see a detailed description in §5.1.2).

If a read statement is enabled, then a read event e is created, and the set $\mathtt{Schedules}(e)$ is initialized to the empty set (at line 7). The set $\mathtt{Schedules}(e)$ will be gradually updated by the DeclarePostponed procedure when subsequent writes are explored. We also associate a Boolean flag $\mathtt{Swappable}(e)$ for each read event e. The flag indicates whether e is *swappable*, i.e., whether following write events should consider e for swapping or not. The reason for including this flag is to prevent DeclarePostponed from swapping read events that occur in a schedule; this would lead to redundant explorations and a violation of the optimality of the algorithm. After that, for each already generated write event e' from which e can read its value, DfVisit is called recursively to continue the exploration (at line 8). After these calls have returned, the set of schedules collected in $\mathtt{Schedules}(e)$ for the read event e is considered. Each such a schedule is explored by the RunSchedule procedure, thereby allowing e to fetch its value from the corresponding write event.

5.1.2 The DeclarePostponed Procedure (Algorithm 2). This procedure inputs a trace τ (the trace that has been built up to this point) and a sequence of observations π whose last element is a write event e. The algorithm finds the closest e' to e in π for which e can be considered as a postponed write. The algorithm then adds a schedule to the set $\mathtt{Schedules}(e')$ to allow e' to read from e in a new exploration. The criterion for considering such a read event e' is that it satisfies four conditions, namely (i) e' reads from the same variable to which e writes, (ii) e' does not precede e w.r.t. the relation $(\mathsf{po} \cup \mathsf{rf})^+$, (iii) e' is swappable, and (iv) e' is closest to e. The first three conditions are checked at line 4. The condition (iv) is satisfied by the break statement at line 11 which makes the procedure stop after finding the first read event satisfying the first three conditions.

Algorithm 2: DeclarePostponed(τ, π)

Input: $\tau = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co} \rangle$ is a trace and π is an explored observation sequence.

1 **let** e be $last(\pi)$ and x be $e.var$
2 **for** $i \leftarrow |\pi| - 1$ **to** 1 **do** // look for the closest event e' that has postponed e
3 **let** e' be $\pi[i].event$
4 **if** $e' \in \mathsf{E}^{\mathsf{R},x} \wedge \neg(e' \, [\mathsf{po} \cup \mathsf{rf}]^+ e) \wedge \mathtt{Swappable}(e')$ **then**
5 $\beta \leftarrow \epsilon$
6 **for** $j \leftarrow i + 1$ **to** $|\pi| - 1$ **do** // get all events after e' in π and precedes e in $(\mathsf{po} \cup \mathsf{rf})^+$
7 **let** e'' be $\pi[j].event$
8 **if** $e'' \, [\mathsf{po} \cup \mathsf{rf}]^+ e$ **then** $\beta \leftarrow \beta \bullet \pi[j]$
9 **if** $\nexists \beta' \in \mathtt{Schedules}(e') \,.\, \beta' \approx \beta \bullet e \bullet \langle e', e \rangle$ **then**
10 $\mathtt{Schedules}(e') \leftarrow \mathtt{Schedules}(e') \cup \{\beta \bullet e \bullet \langle e', e \rangle\}$ // allow e' to read from e
11 **break**

Algorithm 3: RunSchedule$(\gamma, \tau, \pi, \beta)$

Input: γ is a configuration, τ is a trace, π is an explored-observation sequence, and β is a schedule.

1 **if** $\beta \neq \epsilon$ **then** // explore the sequence of observations one by one
2 **let** present β in the form $\alpha \bullet \beta'$ and e be $\alpha.event$
3 **choose** $\gamma', \tau' : \langle \gamma, \tau \rangle \xrightarrow{\alpha}_{\mathsf{S}} \langle \gamma', \tau' \rangle$ // follow S-WRITE and S-READ
4 **if** $e.type = \mathsf{R}$ **then** $\mathtt{Swappable}(e) \leftarrow \mathtt{false}$
5 RunSchedule$(\gamma', \tau', \pi \bullet \alpha, \beta')$
6 **else** DfVisit(γ, τ, π)

After finding such a read event e', a schedule β is created. The schedule consists of all the events following e' in π and which precede e w.r.t. the relation $(\mathsf{po} \cup \mathsf{rf})^+$ (at line 6). The schedule β has the write event e and then the pair $\langle e', e \rangle$ at the end, thereby achieving the goal of making e' read from e (at line 10).

The new schedule β is added to the set $\mathtt{Schedules}(e')$ only if the latter does not already contain a schedule β' which is *equivalent* to β. We consider β and β' to be equal, denoted $\beta \approx \beta'$, if they contain the same set of observations (possibly in different orders). Notice that if $\beta \approx \beta'$ and we execute β or β' then we reach identical traces which means that β need not be added to the set $\mathtt{Schedules}(e')$ if β' is already in $\mathtt{Schedules}(e')$, and that having both β and β' in $\mathtt{Schedules}(e')$ would in fact violate the optimality condition.

5.1.3 The RunSchedule Procedure (Algorithm 3). The procedure inputs a configuration γ, a trace τ, a sequence of observations π, and a schedule β. It explores the sequence of scheduled observations in β one by one, by calling itself recursively (at line 5), after which DfVisit is called to continue the exploration beyond the end of π (at line 6). Note that each observation corresponds to a statement which by construction is enabled. During this exploration, read events are declared non-swappable by setting the corresponding entry in `Swappable` to false.

5.2 Complexity

Our DPOR algorithm spends *polynomial time* for each explored trace. This complexity is estimated using the following facts.

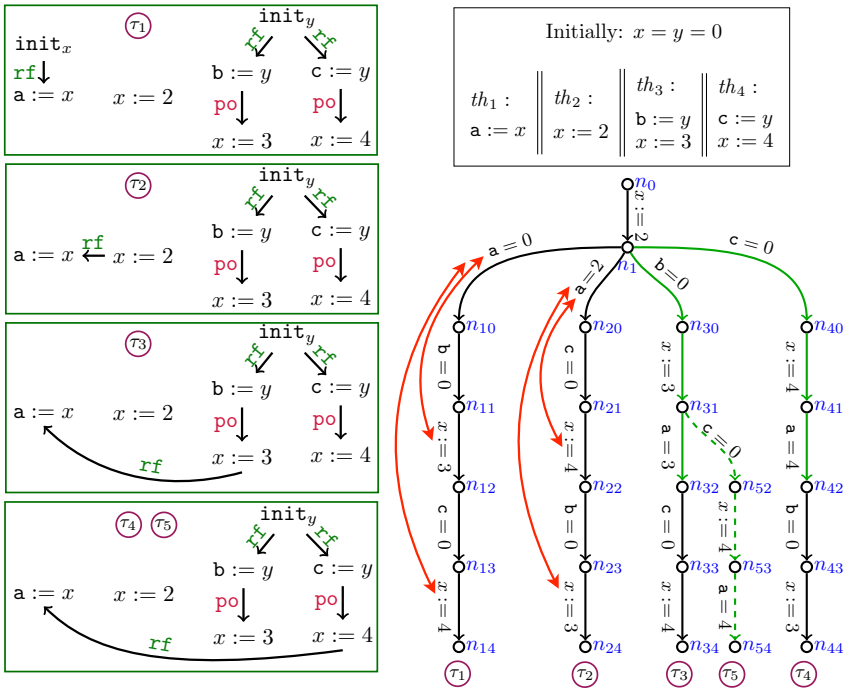


Fig. 7. Example illustrating the DPOR algorithm.

(1) Each single schedule is at most *linear* (in the size of the program), since its length is bounded by the length of the program.
(2) It is indeed possible (in pathological cases) that the exploration will produce an exponential number of schedules. However, each of these schedules will result in a different weak trace; therefore this happens only when the number of weak traces of the program is exponential. Thus (importantly) the effort in constructing these schedules is *never wasted*.
(3) Even if the number of schedules is exponential, it can be checked in *polynomial* (at most quadratic) time whether a new schedule is equivalent to an existing one. This can be done by organizing the schedules into a tree and systematically compare the new schedule from the beginning to the schedules in the tree (the essential operation is to compare read-from edges).

5.3 Example

Figure 7 illustrates the recursion tree corresponding to a run of the DPOR algorithm on a simple program consisting of four threads $\{th_1, th_2, th_3, th_4\}$ and two shared variables $\{x, y\}$. We will use the same simplified notation for events and observations as in the example in §4.3.

The nodes in the tree represent the recursive calls of DfVisit. The node n_0 represents the first call with an empty trace and an empty input observation sequence. From n_0, DfVisit selects the only available write statement $x := 2$ (of th_2), making a recursive call and moving to n_1 (line 1 of DfVisit). At node n_1, one of the three read statements, namely $\mathsf{a} := x$ is selected (line 5 of DfVisit). There are two possible write events from which the event $\mathsf{a} := x$ can read, namely the initial write event $\texttt{init}_x$ which gives the value 0 and the event $x := 2$ which gives the value 2. Accordingly, when DfVisit runs the for-loop at line 8, it will eventually create two branches leading to the

nodes n_{10} and n_{20}. In the first iteration of the for-loop, a recursive call is made leading to the node n_{10} from where the statement $\mathsf{b} := y$ is selected. There is one write event from which $\mathsf{b} := y$ can read its value, namely $\mathtt{init}_y$ so there will be no branching in this case, and the successor node will be n_{11}. At n_{11} the only available write statement $x := 3$ is selected after which $\mathsf{c} := y$ (reading from $\mathtt{init}_y$), and $x := 4$ are executed. Notice that the trace obtained at node n_{14} is τ_1 and the observation sequence is $x := 2\ \mathsf{a} = 0\ \mathsf{b} = 0\ x := 3\ \mathsf{c} = 0\ x := 4$. The recursive call to DfVisit from n_{13} to n_{14} will return immediately to n_{13} since there are no enabled statements left in the program, after which DeclarePostponed will be called (line 4 of DfVisit). The latter will detect that $x := 4$ is postponed w.r.t. $\mathsf{a} := x$ (line 4 of DeclarePostponed). The fact that $x := 4$ is postponed w.r.t. $\mathsf{a} := x$ is marked by the red arrow in Figure 7. The DeclarePostponed procedure executes the for-loop at line 2 and creates a schedule β_1 given by $\mathsf{c} = 0\ x := 4\ \mathsf{a} = 4$ (the event $\mathsf{c} = 0$ precedes $x := 4$ w.r.t. po). The schedule β_1 is added to the set of schedules of $\mathsf{a} := x$. Similarly, when the recursive call returns to the node n_{11}, the schedule β_2 containing the sequence $\mathsf{b} = 0\ x := 3\ \mathsf{a} = 3$ will be added to the set of schedules of $\mathsf{a} := x$.

When the calls of the nodes in the left-most branch (nodes n_{10}–n_{14}) have all returned and we are at node n_1 again, DfVisit will make a recursive call to node n_{20} corresponding to the second iteration of the for-loop at line 8. From n_{20}, the execution will continue similarly to the branch n_{10}–n_{14}, creating the branch n_{20}–n_{24} obtaining the trace τ_2. In particular, two schedules will be obtained namely $\mathsf{c} = 0\ x := 4\ \mathsf{a} = 4$ at n_{21} and $\mathsf{b} = 0\ x := 3\ \mathsf{a} = 3$ at n_{23}. However, these two schedules are identical (and hence trivially equivalent) to the schedules β_1 and β_2 respectively, and therefore they will not be added to the set of schedules of $\mathsf{a} := x$.

When the recursive call from n_{20} has returned to n_1, the for-loop at line 8 will terminate and DfVisit moves to line 9 where it considers the set of schedules of $\mathsf{a} := x$ (in some order). In the example, it selects β_2 first and calls RunSchedule, which will create the nodes n_{30}, n_{31}, and n_{32}. In particular, RunSchedule will set $\mathtt{Swappable}(\mathsf{a} := x)$ to false (line 4). From n_{32} the call returns to DfVisit (line 6 of RunSchedule) and the nodes n_{33} and n_{34} will be created leading to the trace τ_3. Although the write event $x := 4$ is potentially postponed w.r.t. $\mathsf{a} := x$, the corresponding schedule will not be added since the Swappable flag of $\mathsf{a} := x$ is false (line 4 of DeclarePostponed). This means that the algorithm will not create the dotted branch (n_{52}–n_{54}) in Figure 7.

When the recursive call returns from n_{30} to n_1 the schedule β_1 will be run in a similar manner to β_2 resulting the right-most branch (n_{40}–n_{44}) and the trace τ_4.

Optimality. The test at line 9 of DeclarePostponed is necessary. It prevents adding the schedules from n_{23} and n_{21}. These two schedules would lead to duplicates of the traces τ_3 and τ_4. Checking the status of Swappable is also necessary (line 4 of DeclarePostponed). In our example, this prevents adding the nodes n_{52}–n_{54}. Adding these nodes would result in τ_5 which is identical to τ_4.

5.4 Properties of the DPOR Algorithm

Soundness The algorithm is sound in the sense is that if we initiate DfVisit from a configuration γ, the empty trace $\tau_\varnothing$, and the empty sequence of observations ϵ, then $\tau \in [\![\gamma]\!]^{\mathsf{S}}_{\mathrm{RA}}$ for all explored traces τ. This follows from the fact that the exploration uses the $\rightarrow_{\mathsf{S}}$ relation (cf. lines 2 and 8 in Algorithm 1 and line 3 in Algorithm 3).

Optimality The algorithm is optimal in the sense that, for any two different recursive calls to DfVisit with arguments $\langle \gamma_1, \tau_1, \pi_1 \rangle$ and $\langle \gamma_2, \tau_2, \pi_2 \rangle$, if γ_1 and γ_2 are terminal then $\mathit{weak}(\tau_1) \neq \mathit{weak}(\tau_2)$. This follows from the following properties: (i) Each time we run the for-loop at line 8 in Algorithm 1, the read event e will read from a different write event. (ii) In each schedule $\beta \in \mathtt{Schedules}(e)$ at line 9 in Algorithm 1, the event e reads from a write event e' that is different

from all the write events from which e reads at line 8. (iii) Any two schedules added to $\texttt{Schedules}(e)$ at line 10 in Algorithm 2 will have a read event reading from two different write events.

Deadlock-Freedom If DfVisit is called with parameters $\langle \gamma, \tau, \pi \rangle$ where γ is not terminal then it will produce a subsequent recursive call. This follows directly from Lemma 4.6.

Completeness The algorithm is complete in the sense that for any input configuration γ it will produce all weak traces corresponding to terminating runs from γ. More precisely, for any configuration γ, terminated run $\rho \in \mathit{Runs}(\gamma)$, and total trace τ with $\tau \models \rho$, $\textsc{DfVisit}(\gamma, \tau_\emptyset, \epsilon)$ will produce a recursive call $\textsc{DfVisit}(\gamma', \tau', \pi)$ for some terminal γ', τ', and π where $\mathit{weak}(\tau') = \mathit{weak}(\tau)$.

6 EXPERIMENTAL EVALUATION

In this section, we describe the implementation of our new techniques, including the saturated semantics in §4 and the new optimal DPOR algorithm w.r.t. weak traces in §5, as a tool called Tracer (weak TRace ACquirE Release). We evaluate the effectiveness of Tracer in practice, by comparing its performance with two other stateless model checking tools running under the Release-Acquire semantics, namely CDSChecker [Norris and Demsky 2016] and Rcmc [Kokologiannakis et al. 2018].

The Tracer tool Tracer can be used to unit test portions of concurrent code written in the Release-Acquire fragment of the C/C++11 memory model to discover which behaviors the memory model allows. By analyzing the set of weak traces generated during the exploration of programs, Tracer can detect runtime errors such as user-provided assertion violations, deadlocks (with standard definition as in [Silberschatz et al. 2012]), and buggy accesses on uninitialized memory locations. We emphasize that all deadlocks detected by Tracer actually occur in the provided programs and they are not due to our DPOR algorithm.

Tracer is constructed as dynamically-linked shared libraries which implement the C/C++11 `acquire` and `release` accesses types in the `<atomic>` library [ISO 2012] and portions of the other thread-support libraries of C/C++11 (e.g., `<threads>` and `<mutex>`). In more detail, it instruments any concurrency-related API calls such as write and read accesses to shared atomic variables, thread creations and thread joinings in input programs into function calls provided by self-implemented utility libraries. At runtime, Tracer determines the next possible transitions and the values returned by atomic memory operations following the saturated semantics in §4. Furthermore, Tracer controls the exploration of the input program until it has explored the whole set of weak traces, using the DPOR algorithm in §5. To verify the correctness of the instrumented libraries in Tracer, we substitute the corresponding headers (e.g., for atomic operations) in GCC and Clang with the headers of the libraries in Tracer. We observe no behavioral differences in our benchmarks when we compile and run them using GCC or Clang with and without the instrumented libraries. Finally, we note that Tracer partly uses some source code from CDSChecker to preprocess and handle input programs at the first step in its model checking progress.

CDSChecker and Rcmc tools We compare Tracer with two other state-of-the-art stateless model checking tools running under the RA semantics, namely CDSChecker [Norris and Demsky 2016] and Rcmc [Kokologiannakis et al. 2018]. CDSChecker implements a DPOR algorithm that supports concurrent programs running under the RA semantics, but, as illustrated in §6.1–§6.6, might generate a lot of redundant executions in a less efficient way than Tracer. Rcmc has two options: Rc11 and Wrc11. The Rc11 option generates only consistent executions (w.r.t. RA) by maintaining total coherence orders. Meanwhile, the Wrc11 option maintains partial coherence orders, which may create inconsistent executions which are then not validated (see §6.1). Rcmc/Rc11

is optimal in the sense that, in the absence of RMWs, they explore each consistent Shasha-Snir trace [Shasha and Snir 1988] exactly once. Observe that Rcmc has limited support for thread creation and joining [Personal-com 2018]. Thus, we do some engineering transformations to simulate these operations.

Experiment setup We compare Tracer with CDSChecker and Rcmc on six different categories of benchmarks as follows.

(1) In §6.1, we apply Tracer, CDSChecker, and Rcmc to a set consisting of thousands of litmus tests (i.e., small programs) taken from [Alglave et al. 2014]. These experiments are mainly used for two purposes: (i) to provide sanity checks of the correctness of the used tools, and (ii) to compare the performance of these tools on small programs running under RA.
(2) In §6.2, we run the tools on two standard programs taken from [Flanagan and Godefroid 2005]. These benchmarks have been widely used to compare different stateless model-checking tools (e.g. [Abdulla et al. 2014; Kokologiannakis et al. 2018; Saarikivi et al. 2012]).
(3) In §6.3, we run the tools on a collection of concurrent benchmarks taken from the TACAS competition on Software Verification [SV-COMP 2018]. These benchmarks are C/C++ programs with 50-100 lines of code and used by many tools (e.g. [Abdulla et al. 2015a, 2016b; Alglave et al. 2013a; Huang and Huang 2016]).
(4) In §6.4, we evaluate the tools on two synthetic programs (one of them is taken from [Norris and Demsky 2016]). We use these benchmarks to show the benefits of using weak traces used by Tracer (see §3) instead of Shasha-Snir traces [Shasha and Snir 1988] used by Rcmc.
(5) In §6.5, we apply the tools to three parameterized programs to study their behaviors when increasing the numbers of read and write operations and threads in the programs.
(6) In §6.6, we use the tools to analyze more extensive and more challenging benchmarks, containing several implementations of concurrent data structures. Some of them were used in the evaluation of CDSChecker [Demsky and Lam 2015; Norris and Demsky 2016] and Rcmc [Kokologiannakis et al. 2018]. The other benchmarks are high-performance starvation-free algorithms in [Ramalhete and Correia 2016, 2018].

We conduct all experiments on a Debian 4.9.30-2+deb9u2 machine with an Intel(R) Core(TM) i7-3720QM CPU (2.60 GHz) and 16 GB of RAM. Tracer and CDSChecker have been compiled with GCC v6.3.0. We use the artifact-evaluated version of Rcmc [Kokologiannakis et al. 2018] that is compiled with Clang and LLVM v3.8.1. We use the argument `-u 0` in all benchmarks for both Tracer and CDSChecker that provides 0 for `release` atomic loads from uninitialized memory locations. All these tools were run on deterministic programs with bounded executions. This is achieved by unrolling loops in any tested program up to a specific bound. Therefore, we do not need `-m` and `-y` arguments to control memory liveness and thread yield-based fairness for Tracer and CDSChecker as in [Norris and Demsky 2016] and `-unroll` argument for Rcmc. Finally, all experiments have been set up with a 1-hour timeout.

6.1 Evaluation using Litmus Tests

We apply Tracer, CDSChecker, and Rcmc on a set of litmus tests, taken from [Alglave et al. 2014]. Litmus tests are standard benchmarks used by many tools running on weak memory models (e.g. [Abdulla et al. 2016b; Alglave et al. 2014; Kokologiannakis et al. 2018; Sarkar et al. 2011]). Typically, each litmus test contains at most 4 threads with less than 5 shared variables. In a litmus test, threads concurrently execute small sets of instructions. After all threads have finished their executions, an assertion is validated to check whether the set of some behaviors presented by the assertion is allowed. Since litmus tests are small but contain all kinds of dependencies such as address dependency, data dependency, and control dependency (see [Alglave et al. 2014]), they are

Table 1. Comparison of the three tools when running on the litmus tests. The column *Wrong answer* corresponds to the number of wrong answers. The *Execs* (resp. *Average Execs*) column is the total number of explored executions (resp. the average explored executions per test). The *Time* (resp. *Average Time*) column is the total time (resp. the average time per execution) for running the tests without accumulating the compilation time. For Rcmc, these running time are approximate because we cannot measure only the running time and exclude the compilation time. We believe that the compilation time of Rcmc is far smaller than the running time. The best number of executions and running time of the tools are given in bold font.

Tool	Wrong answer	Execs	Time	Average Execs (*Execs*/*Nums of tests*)	Average Time (*Time*/*Execs*)
CDSChecker	0	2 383 704	6m28s	295.59	**0.16ms**
Rcmc/Rc11	0	727 942	14m00s	90.27	1.15ms
Rcmc/Wrc11	81	668 574	12m15s	82.91	1.10ms
Tracer	0	**521 288**	**3m03s**	**64.64**	0.35ms

usually used to provide sanity checks of the correctness of the tools and to compare the performance of these tools on small benchmarks. Observe that all the above dependencies in litmus tests can be handled under the RA semantics by executing all events following the program order and using the read-from relations. We translate litmus tests into C/C++11 by considering all writes and reads as `release` and `acquire` atomic accesses, and all synchronizations as RMW accesses (implemented by `atomic_fetch_add` API calls) to a particular fence variable. For an unsafe test, we stop all tools as soon as they find the first bug since Rcmc does so.

Table 1 summarizes the results of different tools on 8064 litmus tests. First of all, all tools, except Rcmc/Wrc11, are consistent w.r.t. the RA semantics. Rcmc/Wrc11 is inconsistent, generating 81 wrong answers[2]. A *wrong answer* denotes that the outcome of the tool in a test is different from the expected outcome[3]. In the above 81 wrong answers, Rcmc/Wrc11 reports unsafe results for the corresponding safe tests. The inconsistency of Rcmc/Wrc11 is due to the imprecise handling of coherence orders. In fact, Rcmc/Wrc11 sometimes does not generate coherence orders that are needed to preserve RA consistency, e.g., the coherence edges in the trace τ_7 in Figure 6. The missing coherence orders help Rcmc/Wrc11 to reduce the number of executions, but can also lead to inconsistent executions. Indeed, Rcmc/Wrc11 allows the inconsistent trace τ_9 in Figure 6 for the program given in Figure 1a. Secondly, Tracer explores fewer executions and has a better performance than the other tools. For example, it generates 22% fewer executions than Rcmc/Wrc11 (that can be observed based on the columns *Execs* and/or *Average Execs* in Table 1) which in turn is exploring fewer executions than the remaining tools. Moreover, Tracer is also 4 times faster than Rcmc/Wrc11. Thirdly, we compare three tools on *the average running time spent by each tool on the exploration of an execution* (the *Average Time* column in Table 1). The average running time per execution is calculated using the formula

$$AverageTime := Time/Execs$$

i.e. it is the average time a tool needs to explore one execution. We observe that based on *AverageTime*, Tracer is about 3 times faster than Rcmc but slower than CDSChecker. The reason is that CDSChecker can explore prefixes of some redundant executions before ending them and therefore CDSChecker spends very little time on these small incomplete executions. Meanwhile,

[2]Based on our reported results, newer versions of Rcmc can fix some of the 81 wrong answers [Personal-com 2018].
[3]We generated the expected outcomes using the Herd tool together with the set of RA-axioms provided in [Lahav et al. 2016]. Since Herd is not a stateless model checker and it is significantly slower in these tests, we do not include Herd in our comparison.

Table 2. Comparison of the performance of the three tools when running on two standard benchmarks: `Indexer(N)` [SV-COMP 2018] and `Filesystem(N)` [Flanagan and Godefroid 2005]. The *LB* column indicates how many times a loop is unrolled. If a tool runs out of time, we put *t/o*. The best number of executions and running time for each benchmark are given in bold font.

Program	LB	Executions Explored			Total Running Time		
		CDSChecker	Rcmc	Tracer	CDSChecker	Rcmc	Tracer
Indexer(13)	20	190	**64**	**64**	0.15s	0.62s	**0.04s**
Indexer(14)	20	3 075	**512**	**512**	2.72s	11.68s	**0.38s**
Indexer(15)	20	48 261	**4 096**	**4 096**	45.32s	4m22s	**2.98s**
Indexer(16)	20	740 889	*t/o*	**32 768**	12m21s	*t/o*	**25.78s**
Filesystem(16)	20	27	**8**	**8**	0.04s	0.05s	**0.01s**
Filesystem(19)	20	728	**64**	**64**	0.52s	2.02s	**0.03s**
Filesystem(22)	20	19 678	**512**	**512**	13.53s	2m29s	**0.33s**
Filesystem(25)	20	531 415	*t/o*	**4096**	7m04s	*t/o*	**3.12s**

all explored executions by Tracer and Rcmc are complete since all litmus tests only have assertions at the end of programs.

Since Rcmc/Wrc11 is inconsistent w.r.t. the RA semantics (cf. Table 1), we exclude it from further comparisons. For the sake of convenience, in the following, we use Rcmc for Rcmc/Rc11.

6.2 Evaluation using two Standard Benchmarks

We compare Tracer with CDSChecker and Rcmc on two standard benchmarks, namely `Indexer(N)`[4] and `Filesystem(N)`, introduced in [Flanagan and Godefroid 2005]. These benchmarks have been widely used to compare different DPOR techniques under the SC semantics, e.g. [Abdulla et al. 2014; Flanagan and Godefroid 2005; Saarikivi et al. 2012]. We parameterize these benchmarks by the number of threads. For `Indexer(N)`, we use 13, 14, 15, and 16 threads. For `Filesystem(N)`, we use 16, 19, 22, and 25 threads. The reason of the parameterization is that more threads generate more conflicts between their write and read operations, and therefore more non-equivalent executions are needed to be explored by different tools. For example, there will be no conflicts between the operations of threads in `Indexer(N)` with 11 threads and in `Filesystem(N)` with 13 threads.

In Table 2, we report the number of executions that the three tools explore as well as the time needed by the tools to explore them. We observe that Tracer always examines optimal numbers of executions w.r.t. weak traces and has a better performance than CDSChecker and Rcmc in all benchmarks. The difference between Tracer and the other tools becomes more evident when we increase the number of threads. In both `Indexer(N)` and `Filesystem(N)`, Rcmc can discover the same numbers of executions as Tracer; however, its total running time and the average time per execution are two orders of magnitude slower than Tracer on average. Observe that the used version of `Indexer(N)` program contains lock primitives (as it is the case in SV-COMP [2018]). This version is different from the one used in [Kokologiannakis et al. 2018] and this explains the difference in the obtained performance for Rcmc between the one reported in Table 2 and the one published in [Kokologiannakis et al. 2018]. Based on our discussion with Rcmc's authors [Personal-com 2018], we conjecture that this performance issue is due to the way Rcmc handles lock primitives. Finally, CDSChecker explores more executions than Tracer in all benchmarks, and it is about 20 times slower than Tracer.

[4] We do experiments with the `Indexer(N)` benchmark from SV-COMP [2018] since it is similar to the original one and widely used in the verification community. The `Indexer(N)` in [Kokologiannakis et al. 2018] implements a different version.

Table 3. Comparison of the performance of the three tools when running on the TACAS Competition on Software Verification (SV-COMP) benchmarks. The *LB* column and the *t/o* entry have the same meanings as in Table 2.

Program	LB	Executions Explored			Total Running Time		
		CDSChecker	Rcmc	Tracer	CDSChecker	Rcmc	Tracer
Pthread_demo	10	184 758	**184 756**	**184 756**	24.96s	1m03s	**18.00s**
Gcd	8	8 814 044	**1 162 333**	**1 162 333**	38m20s	7m07s	**2m48s**
Fibonacci	6	*t/o*	**525 630**	**525 630**	*t/o*	**31.06s**	57.95s
Szymanski	6	*t/o*	26 037 490	**12 209 410**	*t/o*	44m52s	**14m54s**
Dekker	10	7 306 447	**3 121 870**	**3 121 870**	15m25s	5m12s	**4m52s**
Lamport	8	*t/o*	6 580 870	**3 372 868**	*t/o*	14m40s	**6m58s**
Sigma(5)	5	1 279	945	**120**	0.09s	0.16s	**0.01s**
Peterson	6	*t/o*	**1 897 228**	**1 897 228**	*t/o*	3m16s	**3m15s**
Stack_true	12	2 704 157	**2 704 156**	**2 704 156**	19m03s	54m25s	**10m12s**
Queue_ok	12	581 790	*t/o*	**362 880**	33m27s	*t/o*	**12m52s**

6.3 Evaluating using SV-COMP Benchmarks

We compare Tracer with CDSChecker and Rcmc on the set of concurrent benchmarks from SV-COMP [2018] (the TACAS Software Verification competition 2018), cf. Table 3. These benchmarks consist of ten programs that are written in C/C++ with 50-100 lines of code and used by many tools (e.g. [Abdulla et al. 2015a, 2016b; Alglave et al. 2013a; Huang and Huang 2016]). The primary challenge in these benchmarks is to handle a large number of executions that are needed to be explored. For example, Rcmc generates 26 million executions for the `Szymanski` benchmark. As stated in §6.1, for unsafe benchmarks, Rcmc stops the exploration as soon as it detects the first assertion violation. To fairly compare the efficiency of different DPOR approaches, as it was done in [Abdulla et al. 2014; Kokologiannakis et al. 2018], we remove all the assertions in the benchmarks to let all tools *exhaustively* explore all possible executions.

From Table 3, we can see that Tracer explores smaller numbers of executions than the two other tools in all cases. As a consequence, Tracer has the best performance in 9 of 10 examples. Meanwhile, CDSChecker times out in 4 of 10 cases. The main reason for these timeouts is that CDSChecker needs to explore a large number of executions. This can be seen in the `Gcd` example where the number of executions generated by CDSChecker is huge, compared to those produced by Rcmc and Tracer. On two benchmarks, `Pthread_demo` and `Stack_true`, where the three tools generate almost the same numbers of executions, CDSChecker is faster than Rcmc but still less efficient than Tracer. The only benchmark in which Tracer does not have the best performance is `Fibonacci`. In this benchmark, the numbers of executions explored by Tracer and Rcmc are exactly the same, and Rcmc is slightly faster. However, in general Tracer generates 40% fewer number of executions and it is about 3 times faster than Rcmc. Interestingly, Tracer is 2 times faster than Rcmc in the average time per execution, which coincides with the litmus tests in §6.1.

6.4 Evaluation using Synthetic Benchmarks

Next, we expose more differences between the three tools on two synthetic benchmarks given in Figure 8. The first one is `N_writers_a_reader(N)` benchmark, taken from [Norris and Demsky 2016]. The results, for 7, 8, 9, and 10 threads, are given in Table 4. Since this benchmark does not contain any loops, we do not use loop unrolling and therefore do not show the LB column as in Tables 2 and 3. Since Rcmc is optimal w.r.t. Shasha-Snir traces in the absence of RMWs, the number of executions explored by Rcmc is exactly (factorial) $(N+1)!$ here. This number corresponds to

```
/* initially: x=0 */
atomic_int x;

/* N writers*/
void writers(void *arg)
{
  /* tid is from 1 to N */
  int tid = *((int *)arg);
  x.store(tid, release);
}

void reader(void *arg)
{
  int a = x.load(acquire);
}
```

```
/* initially: x=0 */
atomic_int x;

/* 2 writers */
void writers(void *arg)
{
  for (int i=0; i<N; i++)
      x.store(1, release);
}

void reader(void *arg)
{
  int a = x.load(acquire);
  int b = x.load(acquire);
}
```

Fig. 8. C/C++11 codes of `N_writers_a_reader(N)` (left) and `Redundant_co(N)` (right).

Table 4. Comparison of the performance of the three tools when running on two synthetic programs. The *t/o* entry has the same meaning as in Table 2.

Program	Executions Explored			Total Running Time		
	CDSChecker	Rcmc	Tracer	CDSChecker	Rcmc	Tracer
N_writers_a_reader(7)	**8**	40 320	**8**	0.01s	0.46s	**0.00s**
N_writers_a_reader(8)	**9**	362 880	**9**	0.01s	4.19s	**0.00s**
N_writers_a_reader(9)	**10**	3 628 800	**10**	0.01s	46.13s	**0.00s**
N_writers_a_reader(10)	**11**	39 916 800	**11**	0.01s	9m35s	**0.00s**
Redundant_co(5)	581	16 632	**91**	0.03s	0.39s	**0.01s**
Redundant_co(10)	10 631	42 678 636	**331**	0.64s	23m56s	**0.02s**
Redundant_co(15)	59 056	*t/o*	**721**	4.57s	*t/o*	**0.06s**
Redundant_co(20)	197 231	*t/o*	**1 261**	20.27s	*t/o*	**0.14s**

the number of possible combinations of all feasible po, rf, and total co. However, one can easily see that there are only $N + 1$ possible values for the read, presented by $\mathtt{a} = 0, \mathtt{a} = 1, \cdots, \mathtt{a} = N$. In fact, the total coherence order is irrelevant in this benchmark and only the read-from relation is important. Therefore, in contrast to Rcmc, Tracer, which is optimal w.r.t. weak traces, precisely explores $N + 1$ executions (i.e., linear).

The second benchmark is called `Redundant_co(N)` and its code is given in Figure 8. We use this example to emphasize more the benefit of using weak traces. As depicted in Table 4, while the number of Shasha-Snir traces and the number of explored traces by Rcmc is $O(N!)$ (i.e., factorial), Tracer explores only $O(N^2)$ executions[5]. For instance for `Redundant_co(10)`, Tracer finishes the exploration of the program in less than 1 second, while Rcmc examines two orders of magnitude more executions and needs almost half an hour to complete. Since CDSChecker also maintains a partial coherence order as Tracer but not in an optimal way, CDSChecker can explore the same number of executions for `N_writers_a_reader(N)` as Tracer but not in `Redundant_co(N)`.

6.5 Evaluation using Parameterized Benchmarks

Table 5 reports more experimental results of Tracer, CDSChecker, and Rcmc on three parameterized benchmarks: `Sigma(N)` (presented in Table 3), `Control_flow(N)` (used in [Abdulla et al. 2014]), and `Exponential_bug(N)` (presented in Figure 2 of [Huang 2015]). In `Sigma(N)`

[5]The exact number of weak traces in `Redundant_co(N)` is $3N^2 + 3N + 1$.

Table 5. Comparison of the performance of the three tools when running on three parameterized benchmarks. The *LB* column and the *t/o* entry have the same meanings as in Table 2.

Program	LB	Executions Explored			Total Running Time		
		CDSChecker	Rcmc	Tracer	CDSChecker	Rcmc	Tracer
Sigma(6)	6	25 357	10 395	**720**	2.20s	1.96s	**0.04s**
Sigma(7)	7	605 714	135 135	**5 040**	1m02s	29.06s	**0.40s**
Sigma(8)	8	16 667 637	2 027 025	**40 320**	33m18s	8m02s	**3.28s**
Sigma(9)	9	*t/o*	*t/o*	**362 880**	*t/o*	*t/o*	**33.59s**
Control_flow(6)	0	896	55 440	**77**	0.09s	1.97s	**0.01s**
Control_flow(8)	0	4 608	11 007 360	**273**	0.53s	7m58s	**0.03s**
Control_flow(10)	0	22 528	*t/o*	**1 045**	3.27s	*t/o*	**0.16s**
Control_flow(12)	0	106 496	*t/o*	**4 121**	19.10s	*t/o*	**0.79s**
Exponential_bug(6)	6	983 386	1 203 446	**15 601**	1m18s	56.75s	**0.96s**
Exponential_bug(7)	7	2 250 290	2 833 112	**22 841**	3m13s	2m26s	**1.46s**
Exponential_bug(8)	8	4 844 378	6 158 718	**32 313**	7m15s	5m28s	**2.23s**
Exponential_bug(9)	9	9 896 954	12 526 576	**44 428**	15m29s	11m48s	**3.22s**

and `Control_flow(N)`, N is the number of threads used in these benchmarks. Meanwhile, in `Exponential_bug(N)`, N is the number of times a thread writes to a specific variable. In a similar way to Tables 2–4, Table 5 shows that Tracer always explores smaller numbers of executions than CDSChecker and Rcmc in all benchmarks, and it has the best performance. The differences between Tracer and the other tools becomes more clear when we increase the number of threads. For example, in `Sigma(N)` benchmark, the number of executions explored by Tracer increases only 8 times at each step, while it increases by 23 and 15 times for CDSChecker and Rcmc respectively. As a consequence, when there are more than 7 threads, the verification tasks of `Sigma(N)` are very challenging for CDSChecker and Rcmc but can still be handled by Tracer in less than 1 minute.

6.6 Evaluation using Concurrent Data Structure Benchmarks

We apply Tracer to fifteen concurrent data structure algorithms, namely `Correia_Ramalhete(N)`, `Correia_Ramalhete_turn(N)`, `Tidex(N)`, `Tidex_nps(N)`, `CLH_c11(N)`, `CLH_rwlock_c11(N)`, `MPSC_c11(N)`, `Ticket_mutex_c11(N)`, `Linux_lock(N)`, `Barrier(N)`, `Seqlock(N)`, `MPMC_queue(N)`, `MCS_lock(N)`, `Cliffc_hashtable(N)`, `Concurrent_hashmap(N)` `Chase_Lev_dequeue(N)`, and `SPSC_queue(N)`. The first four algorithms are high-performance starvation-free mutual exclusion locks in [Ramalhete and Correia 2016, 2018]. Meanwhile, `CLH_c11(N)`, `CLH_rwlock_c11(N)`, and `MPSC_c11(N)` are C/C++11 implementations of CLH queue locks [Magnusson et al. 1994] and `Ticket_mutex_c11(N)` is a C/C++11 implementation of Ticket Lock [Mellor-Crummey and Scott 1991], taken from [Ramalhete and Correia 2018]. Other algorithms were used in the previous evaluations of CDSChecker [Demsky and Lam 2015; Norris and Demsky 2016] and Rcmc [Kokologiannakis et al. 2018]. Since these data structures can have huge state-spaces, we limit the number N of used threads in these algorithms to four. We also use loop unrolling when it is necessary. Moreover, to run all algorithms under the RA semantics, as it was done in §6.1–§6.5, we consider all write and read operations as `acquire` and `release` atomic accesses.

Table 6 gives the results on the four verification tasks derived from `Linux_lock(N)`, `Ticket_mutex_c11(N)`, and `Correia_Ramalhete(N)` algorithms. We observe that both Tracer and Rcmc have good performance on these benchmarks. Tracer always generates fewer executions than Rcmc and CDSChecker, which coincides with our previous observations in §6.1–§6.5. To be

Table 6. Comparison of the performance of the three tools when running on concurrent data structure benchmarks. The *LB* column and the *t/o* entry have the same meanings as in Table 2.

Program	LB	Executions Explored			Total Running Time		
		CDSChecker	Rcmc	Tracer	CDSChecker	Rcmc	Tracer
Linux_lock(2)	6	47	**21**	**21**	0.03s	0.02s	**0.01s**
Linux_lock(3)	6	14 187 799	**412 814**	**412 814**	16m01s	36.36s	**33.21s**
Ticket_mutex_c11(3)	10	8 054	**4 026**	**4 026**	0.69s	50.76s	**0.25s**
Correia_Ramalhete(3)	5	**5 355**	**5 355**	**5 355**	1.10s	**0.77s**	0.92s

Table 7. Comparison of the performance of CDSChecker and Tracer when running on eight concurrent data structure benchmarks. The *LB* column and the *t/o* entry have the same meanings as in Table 2.

Program	LB	Executions Explored		Total Running Time	
		CDSChecker	Tracer	CDSChecker	Tracer
Barrier(3)	10	62 649	**31 944**	5.45s	**1.88s**
Seqlock(3)	5	17 792	**14 864**	1.17s	**0.77s**
MPMC_queue(3)	5	621 882	**239 254**	2m11s	**44.50s**
MCS_lock(2)	5	92 210	**70 072**	8.91s	**5.19s**
Cliffc_hashtable(4)	0	9 520	**4 576**	2.68s	**0.94s**
Concurrent_hashmap(4)	0	110	**42**	0.01s	**0.00s**
Chase_Lev_deque(2) (**deadlock**)	0	162 306	**20 852**	20.65s	**2.10s**
SPSC_queue(2) (**deadlock**)	3	754	**57**	0.18s	**0.01s**

more precise, Tracer and Rcmc explore the same number of executions in the four benchmarks and Tracer has a slightly better performance than Rcmc in 3 of 4 cases. In `Correia_Ramalhete(3)`, the numbers of executions explored by CDSChecker, Rcmc, and Tracer are exactly the same, and Rcmc is slightly faster than CDSChecker and Tracer. Moreover, when we increase the number N of threads in `Linux_lock(N)` from 2 to 3, CDSChecker significantly increases the number of explored executions and the total running time. We note that the reported results on these benchmarks for Rcmc are similar to the results published in [Kokologiannakis et al. 2018]. The reported results for CDSChecker are not the same as the ones reported in [Norris and Demsky 2016] because all benchmarks in Table 6 are loop-unrolled and therefore we need not use the `-m` and `-y` arguments to control memory liveness and thread yield-based fairness for CDSChecker as in [Norris and Demsky 2016].

In the following, we show the performance of Tracer and CDSChecker on the remaining algorithms. We exclude Rcmc from the comparison since the tool currently cannot handle these algorithms [Personal-com 2018]. As it was done in §6.3, we let Tracer and CDSChecker exhaustively explore all possible executions even in the cases where they detect some errors.

Table 7 compares Tracer and CDSChecker on eight algorithms taken from [Norris and Demsky 2016]. We observe that both Tracer and CDSChecker handle these benchmarks quite well. In all examples, Tracer generates fewer executions than CDSChecker and has a better performance, which is also consistent with our previous observations. Interestingly, both tools can detect two deadlock errors in `Chase_Lev_deque(2)` and `SPSC_queue(2)`. The former is due to our weakening of `seq-cst` atomic accesses used in the original benchmark to `acquire` and `release` atomic accesses. The later is a well-known known bug in [SPSC-bug 2008].

Table 8 compares Tracer and CDSChecker on the remaining six algorithms taken from [Ramalhete and Correia 2018]. In a similar way to Table 7, both tools handle the benchmarks quite well. Moreover, we also observe that Tracer explores fewer executions than CDSChecker and

Table 8. Comparison of the performance of CDSChecker and Tracer when running on six concurrent data structure benchmarks. The *LB* column and the *t/o* entry have the same meanings as in Table 2.

Program	LB	Executions Explored		Total Running Time	
		CDSChecker	Tracer	CDSChecker	Tracer
Tidex(3)	5	4 676	**748**	0.37s	**0.04s**
Tidex_nps(2)	5	10 257	**10 254**	6m15s	**22.58s**
CLH_c11(3)	10	2 562	**732**	0.22s	**0.06s**
CLH_rwlock_c11(3)	10	20	**6**	0.01s	**0.00s**
MPSC_c11(3)	10	12 937	**4 824**	1.20s	**0.36s**
Correia_Ramalhete_turn(3) (**mutex broken**)	2	441 494	**96 184**	1m17s	**12.78s**

has a better performance. Interestingly, although both tools generate almost the same number of executions in `Tidex_nps(2)`, Tracer is significantly faster than CDSChecker. Finally, both tools can efficiently detect a violation of the mutual exclusion property in `Correia_Ramalhete_turn(3)`. In the same way as the two deadlocks in Table 7, this violation is due to our weakening of `seq-cst` atomic accesses used in the original benchmark to `acquire` and `release` atomic accesses.

6.7 Conclusions of the Experiments

As expected, the experiments show that, for many benchmarks, the number of weak traces is significantly smaller than the number of total traces. On these examples, Tracer (which generates optimal numbers of executions w.r.t. weak traces), is much more efficient than CDSChecker and Rcmc. The results also show that Tracer has better performance and scales better to more extensive programs, even on benchmarks where the numbers of total and weak traces are equal, in which case Tracer explores the same number of executions as the other tools.

7 RELATED WORK

Since the pioneering work of Verisoft [Godefroid 1997, 2005] and CHESS [Musuvathi et al. 2008], stateless model checking (SMC), coupled with (dynamic) partial order reduction techniques (e.g. [Abdulla et al. 2014; Flanagan and Godefroid 2005; Rodríguez et al. 2015]) has been successfully applied to real life programs [Godefroid et al. 1998; Kokologiannakis and Sagonas 2017]. The method of [Abdulla et al. 2014] is optimal w.r.t. Mazurkiewicz traces (i.e., Shasha-Snir traces) under SC while our algorithm is designed for RA and it is optimal w.r.t. weak traces. With modifications, SMC techniques have been subsequently applied to the TSO and PSO weak memory models [Abdulla et al. 2015a; Demsky and Lam 2015; Zhang et al. 2015], and POWER [Abdulla et al. 2016b]. Common to all these approaches is that they explore at least one execution for each Mazurkiewicz [Mazurkiewicz 1986] or Shasha-Snir [Shasha and Snir 1988] traces. This induces a limit on the efficiency of the corresponding tools.

Several recent DPOR techniques try to exploit the potential offered by a weaker equivalence than Mazurkiewicz and Shasha-Snir traces [Chalupa et al. 2018; Huang 2015; Huang and Huang 2016; Norris and Demsky 2016]. Maximal causality reduction (MCR) is a technique based on exploring the possible *values* that reads can see, instead of the possible value-producing writes, as in our approach. MCR has been developed for SC [Huang 2015] and TSO and PSO [Huang and Huang 2016]. It may in some cases explore fewer traces than our approach, but it relies on potentially costly calls to an SMT solver to find new executions. In practice, MCR [Huang 2015] may not be optimal w.r.t. its partitioning (see [Chalupa et al. 2018]) while our algorithm is provably optimal w.r.t. weak traces. Moreover, it remains to be seen whether MCR can be adapted to the RA semantics, and how it would compare. Chalupa et al. [2018] proposes a DPOR algorithm using a similar equivalence

(that is based on po and rf relations) as our weak trace but under SC. Except for the minimal case of an acyclic communication graph [Chalupa et al. 2018], they may still explore a significant number of different executions with the same rf relation. Furthermore, checking the consistency of a trace under SC is an NP-complete problem [Chalupa et al. 2018]. As shown in our results, checking the consistency of a trace under the RA semantics can be efficiently done by polynomial time algorithms (cf. §4).

Recently, SMC was also adapted to (variants of) the C/C++11 memory model, which includes RA, producing the tools CDSChecker [Norris and Demsky 2016] and Rcmc [Kokologiannakis et al. 2018]. CDSChecker maintains a coherence order that need not be total, but not in an optimal way. It can generate inconsistent executions, which must afterward be validated. Rcmc has two options: Rc11 and Wrc11. Under Rc11, it maintains only total coherence orders; under Wrc11 it does not, which may generate RA-inconsistent executions (which are then not validated). Rcmc is optimal under the criterion of total coherence order, but only in the absence of RMW operations. In contrast, our technique is provably optimal w.r.t. weak traces, and including RMWs. On the other hand, CDSChecker and Rcmc also cover the different access modes of the C/C++11 memory model.

Bounded model checking can adapt to various weak memory models (e.g. [Alglave et al. 2013b,a; Torlak et al. 2010]). There is no report on using them for RA, but experiments for POWER [Abdulla et al. 2016b] concluded that Nidhugg is at least as efficient as [Alglave et al. 2013b].

Beyond SMC techniques for weak memory models, there have been many works related to the verification of programs running under weak memory models (e.g., [Abdulla et al. 2016a, 2017, 2015b; Atig et al. 2010; Burckhardt et al. 2007; Kuperstein et al. 2011; Liu et al. 2012]). Some works propose algorithms and tools for monitoring and testing programs running under weak memory models (e.g., [Burckhardt and Musuvathi 2008; Burnim et al. 2011; Liu et al. 2012]).

8 CONCLUSIONS AND FUTURE WORK

We have presented a new approach to defining DPOR algorithms which is optimal in the sense that it generates at most one trace with a given program order and read-from relation. We have instantiated the approach for the RA fragment of C/C++11. Our tool demonstrates that our method is substantially more efficient than state-of-the-art tools that handle the same fragment.

Although we only consider the RA semantics in this paper, we believe that our approach is general and can be extended to other memory models. For instance, we can extend the approach to the SRA (Strong RA) semantics [Lahav et al. 2016] by a modification of the sets of readable and visible events (cf. §4). It is interesting to see whether we can also handle the *relaxed* fragment of C/C++11 by employing a swapping mechanism for event speculations similar to the one we have proposed in this paper for treating postponed write events (cf. §5). For other models such as SC, our saturation scheme is necessarily not complete, since saturation for such models amounts to solving an NP-complete problem [Chalupa et al. 2018]. However, we believe that by maintaining saturated traces and only running the costly operations mentioned in [Chalupa et al. 2018] "by demand", we can substantially improve efficiency even under the SC semantics.

ACKNOWLEDGMENTS

We thank Brian Demsky, Peizhao Ou, Konstantinos Sagonas, Michalis Kokologiannakis, Carl Leonardsson, Magnus Lång, and the OOPSLA'18 reviewers for their helpful feedback. This work was carried out within the UPMARC Linnaeus centre of excellence (Uppsala Programming for Multicore Architectures Research Center).

REFERENCES

P.A. Abdulla, S. Aronis, M. Faouzi Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. 2015a. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS) (LNCS)*, Vol. 9035. Springer, London, UK, 353–367.

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Symposium on Principles of Programming Languages, (POPL)*. ACM, San Diego, CA, USA, 373–384.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2016a. The Benefits of Duality in Verifying Concurrent Programs under TSO. In *CONCUR 2016*. 5:1–5:15.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2017. Context-Bounded Analysis for POWER. In *TACAS 2017*. 56–74.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016b. Stateless Model Checking for POWER. In *Computer Aided Verification, (CAV) (LNCS)*, Vol. 9780. Springer, Toronto, ON, Canada, 134–156.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Ngo Tuan Phong. 2015b. The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO. In *ESOP 2015*. 308–332.

Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013b. Software Verification for Weak Memory via Program Transformation. In *European Symposium on Programming, (ESOP) (LNCS)*, Vol. 7792. Springer, Rome, Italy, 512–532.

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013a. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Computer Aided Verification, (CAV) (LNCS)*, Vol. 8044. Springer, Saint Petersburg, Russia, 141–157.

J. Alglave, L. Maranget, and M. Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74.

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *POPL 2010*. 7–18.

S. Burckhardt, R. Alur, and M.M.K. Martin. 2007. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation, (PLDI)*. ACM, San Diego, California, USA, 12–21.

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

J. Burnim, K. Sen, and C. Stergiou. 2011. Testing concurrent programs on relaxed memory models. In *International Symposium on Software Testing and Analysis, (ISSTA)*. ACM, Toronto, ON, Canada, 122–132.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-centric dynamic partial order reduction. *PACMPL* 2, POPL (2018), 31:1–31:30.

M. Christakis, A. Gotovos, and K. Sagonas. 2013. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *International Conference on Software Testing, Verification and Validation, (ICST)*. IEEE, Luxembourg, Luxembourg, 154–163.

Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. 1999. State Space Reduction Using Partial Order Techniques. *STTT* 2, 3 (1999), 279–287.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. The MIT Press.

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*. ACM, Pittsburgh, PA, USA, 20–36.

C. Flanagan and P. Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages, (POPL)*. ACM, Long Beach, California, USA, 110–121.

P. Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Ph.D. Dissertation. University of Liège. Also, volume 1032 of LNCS, Springer.

P. Godefroid. 1997. Model Checking for Programming Languages using VeriSoft. In *Principles of Programming Languages, (POPL)*. ACM Press, Paris, France, 174–186.

Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design* 26, 2 (2005), 77–101.

P. Godefroid, B. Hammer, and L. Jagadeesan. 1998. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*. 124–133.

Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *Programming Language Design and Implementation, (PLDI)*. ACM, Portland, OR, USA, 165–174.

Shiyou Huang and Jeff Huang. 2016. Maximal causality reduction for TSO and PSO. In *Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*. ACM, Amsterdam, The Netherlands, 447–461.

ISO. 2012. *C++ Specification Standard*. International Organization for Standardization. 1338 (est.) pages.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *Proceedings of the 31st European Conference on Object-Oriented Programming, ECOOP 2017*. 17:1–17:29.

M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. 2018. Effective Stateless Model Checking for C/C++ Concurrency. In *POPL*. to appear, available from http://plv.mpi-sws.org/rcmc/.

M. Kokologiannakis and K. Sagonas. 2017. Stateless model checking of the Linux kernel's hierarchical read-copy-update (tree RCU). In *Symposium on Model Checking of Software, (SPIN)*. ACM, Santa Barbara, CA, USA, 172–181.

Michael Kuperstein, Martin T. Vechev, and Eran Yahav. 2011. Partial-coherence abstractions for relaxed memory models. In *Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA, 187–198.

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 649–662.

F. Liu, N. Nedev, N. Prisadnikov, M.T. Vechev, and E. Yahav. 2012. Dynamic synthesis for relaxed memory models. In *Programming Language Design and Implementation (PLDI)*. ACM, 429–440.

Peter S. Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*. 165–171.

A. Mazurkiewicz. 1986. Trace Theory. In *Advances in Petri Nets*.

John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65.

M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*. USENIX Association, 267–280.

B. Norris and B. Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51.

Doron A. Peled. 1993. All from one, one for all, on model-checking using representatives. In *Computer Aided Verification, (CAV) (LNCS)*, Vol. 697. Elounda, Greece, 409–423.

Personal-com. 2018. Personal communication with authors of RCMC.

Pedro Ramalhete and Andreia Correia. 2016. Tidex: a mutual exclusion lock. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. 52:1–52:2.

Pedro Ramalhete and Andreia Correia. 2018. https://github.com/pramalhe/ConcurrencyFreaks. [Online; accessed 2018-07-10].

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based Partial Order Reduction. In *CONCUR 2015*. 456–469.

O. Saarikivi, K. Kähkönen, and K. Heljanko. 2012. Improving Dynamic Partial Order Reductions for Concolic Testing. In *Application of Concurrency to System Design, (ACSD)*. IEEE, Hamburg, Germany, 132–141.

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 175–186.

K. Sen and G. Agha. 2007. A Race-Detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs. In *Haifa Verification Conference*. 166–182. LNCS 4383.

D. Shasha and M. Snir. 1988. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Programming Languages and Systems* 10, 2 (April 1988), 282–312.

Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2012. *Operating System Concepts* (9th ed.). Wiley Publishing.

SPSC-bug. 2008. https://groups.google.com/forum/#!msg/comp.programming.threads/nSSFT9vKEe0/7eD3ioDg6nEJ. [Online; accessed 2018-04-11].

SV-COMP. 2018. Competition on Software Verification. https://sv-comp.sosy-lab.org/2018. [Online; accessed 2017-11-10].

E. Torlak, M.andana Vaziri, and J. Dolby. 2010. MemSAT: checking axiomatic specifications of memory models. In *Programming Language Design and Implementation (PLDI)*. ACM, Toronto, Ontario, Canada, 341–350.

A. Valmari. 1990. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets (LNCS)*, Vol. 483. 491–515.

Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. In *Programming Language Design and Implementation (PLDI)*. ACM, Portland, OR, USA, 250–259.