Stateless Model Checking Under a Reads-Value-From Equivalence

Pratyush Agarwal[1], Krishnendu Chatterjee[2], Shreya Pathak[1], Andreas Pavlogiannis[3], and Viktor Toman[2](✉)

[1] IIT Bombay, Mumbai, India
[2] IST Austria, Klosterneuburg, Austria
viktor.toman@ist.ac.at
[3] Aarhus University, Aarhus, Denmark

Abstract. Stateless model checking (SMC) is one of the standard approaches to the verification of concurrent programs. As scheduling non-determinism creates exponentially large spaces of thread interleavings, SMC attempts to partition this space into equivalence classes and explore only a few representatives from each class. The efficiency of this approach depends on two factors: (a) the coarseness of the partitioning, and (b) the time to generate representatives in each class. For this reason, the search for coarse partitionings that are efficiently explorable is an active research challenge.

In this work we present RVF-SMC, a new SMC algorithm that uses a novel *reads-value-from (RVF)* partitioning. Intuitively, two interleavings are deemed equivalent if they agree on the value obtained in each read event, and read events induce consistent causal orderings between them. The RVF partitioning is provably coarser than recent approaches based on Mazurkiewicz and "reads-from" partitionings. Our experimental evaluation reveals that RVF is quite often a very effective equivalence, as the underlying partitioning is exponentially coarser than other approaches. Moreover, RVF-SMC generates representatives very efficiently, as the reduction in the partitioning is often met with significant speed-ups in the model checking task.

1 Introduction

The verification of concurrent programs is one of the key challenges in formal methods. Interprocess communication adds a new dimension of non-determinism in program behavior, which is resolved by a scheduler. As the programmer has no control over the scheduler, program correctness has to be guaranteed under all possible schedulers, i.e., the scheduler is adversarial to the program and can generate erroneous behavior if one can arise out of scheduling decisions. On the other hand, during program testing, the adversarial nature of the scheduler is to hide erroneous runs, making bugs extremely difficult to reproduce by testing alone (aka Heisenbugs [1]). Consequently, the verification of concurrent programs rests on rigorous model checking techniques [2] that cover all possible program

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 341–366, 2021.
https://doi.org/10.1007/978-3-030-81685-8_16

behaviors that can arise out of scheduling non-determinism, leading to early tools such as VeriSoft [3,4] and CHESS [5].

To battle with the state-space explosion problem, effective model checking for concurrency is stateless. A stateless model checker (SMC) explores the behavior of the concurrent program by manipulating traces instead of states, where each (concurrent) trace is an interleaving of event sequences of the corresponding threads [6]. To further improve performance, various techniques try to reduce the number of explored traces, such as context bounded techniques [7–10] As many interleavings induce the same program behavior, SMC partitions the interleaving space into equivalence classes and attempts to sample a few representative traces from each class. The most popular approach in this domain is partial-order reduction techniques [6,11,12], which deems interleavings as equivalent based on the way that conflicting memory accesses are ordered, also known as the Mazurkiewicz equivalence [13]. Dynamic partial order reduction [14] constructs this equivalence dynamically, when all memory accesses are known, and thus does not suffer from the imprecision of earlier approaches based on static information. Subsequent works managed to explore the Mazurkiewicz partitioning optimally [15,16], while spending only polynomial time per class.

The performance of an SMC algorithm is generally a product of two factors: (a) the size of the underlying partitioning that is explored, and (b) the total time spent in exploring each class of the partitioning. Typically, the task of visiting a class requires solving a consistency-checking problem, where the algorithm checks whether a semantic abstraction, used to represent the class, has a consistent concrete interleaving that witnesses the class. For this reason, the search for effective SMC is reduced to the search of coarse partitionings for which the consistency problem is tractable, and has become a very active research direction in recent years. In [17], the Mazurkiewicz partitioning was further reduced by ignoring the order of conflicting write events that are not observed, while retaining polynomial-time consistency checking. Various other works refine the notion of dependencies between events, yielding coarser abstractions [18–20]. The work of [21] used a reads-from abstraction and showed that the consistency problem admits a fully polynomial solution in acyclic communication topologies. Recently, this approach was generalized to arbitrary topologies, with an algorithm that remains polynomial for a bounded number of threads [22]. Finally, recent approaches define value-centric partitionings [23], as well as partitionings based on maximal causal models [24]. These partitionings are very coarse, as they attempt to distinguish only between traces which differ in the values read by their corresponding read events. We illustrate the benefits of value-based partitionings with a motivating example.

1.1 Motivating Example

Consider a simple concurrent program shown in Fig. 1. The program has 98 different orderings of the conflicting memory accesses, and each ordering corresponds to a separate class of the Mazurkiewicz partitioning. Utilizing the reads-from abstraction reduces the number of partitioning classes to 9. However, when taking into consideration the values that the events can read and write, the number of cases to consider can be reduced even further. In this specific example, there is only a single behaviour the program may exhibit, in which both read events read the only observable value.

Thread_1	Thread_2	Thread_3
1. $w(x,1)$	1. $w(x,1)$	1. $w(x,1)$
2. $w(y,1)$	2. $w(y,1)$	2. $w(y,1)$
	3. $r(x)$	3. $r(y)$

Equivalence classes:	
Mazurkiewicz [15]	98
reads-from [22]	9
value-centric [23]	7
this work	1

Fig. 1. Concurrent program and its underlying partitioning classes.

The above benefits have led to recent attempts in performing SMC using a value-based equivalence [23,24]. However, as the realizability problem is NP-hard in general [25], both approaches suffer significant drawbacks. In particular, the work of [23] combines the value-centric approach with the Mazurkiewicz partitioning, which creates a refinement with exponentially many more classes than potentially necessary. The example program in Fig. 1 illustrates this, where while both read events can only observe one possible value, the work of [23] further enumerates all Mazurkiewicz orderings of all-but-one threads, resulting in 7 partitioning classes. Separately, the work of [24] relies on SMT solvers, thus spending exponential time to solve the realizability problem. Hence, each approach suffers an exponential blow-up a-priori, which motivates the following question: is there an efficient *parameterized* algorithm for the consistency problem? That is, we are interested in an algorithm that is exponential-time in the worst case (as the problem is NP-hard in general), but efficient when certain natural parameters of the input are small, and thus only becomes slow in extreme cases.

Another disadvantage of these works is that each of the exploration algorithms can end up to the same class of the partitioning many times, further hindering performance. To see an example, consider the program in Fig. 1 again. The work of [23] assigns values to reads one by one, and in this example, it needs to consider as separate cases both permutations of the two reads as the orders for assigning the values. This is to ensure completeness in cases where there are write events causally dependent on some read events (e.g., a write event appearing only if its thread-predecessor reads a certain value). However, no causally dependent write events are present in this program, and our work uses a principled approach to detect this and avoid the redundant exploration. While an example to demonstrate [24] revisiting partitioning classes is a bit more involved one, this property follows from the lack of information sharing between spawned subroutines, enabling the approach to be massively parallelized, which has been discussed already in prior works [21,23,26].

1.2 Our Contributions

In this work we tackle the two challenges illustrated in the motivating example in a principled, algorithmic way. In particular, our contributions are as follows.

(1) We study the problem of verifying the sequentially consistent executions. The problem is known to be NP-hard [25] in general, already for 3 threads. We show that the problem can be solved in $O(k^{d+1} \cdot n^{k+1})$ time for an input of n events, k threads and d variables. Thus, although the problem NP-hard in general, it can be solved in polynomial time when the number of threads and number of variables is bounded. Moreover, our bound reduces to $O(n^{k+1})$ in the class of programs where every variable is written by only one thread (while read by many threads). Hence, in this case the bound is polynomial for a fixed number of threads and without any dependence on the number of variables.

(2) We define a new equivalence between concurrent traces, called the *reads-value-from (RVF)* equivalence. Intuitively, two traces are RVF-equivalent if they agree on the value obtained in each read event, and read events induce consistent causal orderings between them. We show that RVF induces a coarser partitioning than the partitionings explored by recent well-studied SMC algorithms [15,21,23], and thus reduces the search space of the model checker.

(3) We develop a novel SMC algorithm called RVF-SMC, and show that it is sound and complete for local safety properties such as assertion violations. Moreover, RVF-SMC has complexity $k^d \cdot n^{O(k)} \cdot \beta$, where β is the size of the underlying RVF partitioning. Under the hood, RVF-SMC uses our consistency-checking algorithm of Item 1 to visit each RVF class during the exploration. Moreover, RVF-SMC uses a novel heuristic to significantly reduce the number of revisits in any given RVF class, compared to the value-based explorations of [23,24].

(4) We implement RVF-SMC in the stateless model checker Nidhugg [27]. Our experimental evaluation reveals that RVF is quite often a very effective equivalence, as the underlying partitioning is exponentially coarser than other approaches. Moreover, RVF-SMC generates representatives very efficiently, as the reduction in the partitioning is often met with significant speed-ups in the model checking task.

2 Preliminaries

General Notation. Given a natural number $i \geq 1$, we let $[i]$ be the set $\{1, 2, \dots, i\}$. Given a map $f \colon X \to Y$, we let $\mathsf{dom}(f) = X$ denote the domain of f. We represent maps f as sets of tuples $\{(x, f(x))\}_x$. Given two maps f_1, f_2 over the same domain X, we write $f_1 = f_2$ if for every $x \in X$ we have $f_1(x) = f_2(x)$. Given a set $X' \subset X$, we denote by $f|X'$ the restriction of f to X'. A binary relation $\sim$ on a set X is an *equivalence* iff $\sim$ is reflexive, symmetric and transitive.

2.1 Concurrent Model

Here we describe the computational model of concurrent programs with shared memory under the Sequential Consistency (SC) memory model. We follow a standard exposition of stateless model checking, similarly to [14,15,21–23,28],

Concurrent Program. We consider a concurrent program $\mathcal{H} = \{\mathsf{thr}_i\}_{i=1}^{k}$ of k deterministic threads. The threads communicate over a shared memory $\mathcal{G}$ of global variables with a finite value domain $\mathcal{D}$. Threads execute *events* of the following types.

(1) A *write event* w writes a value $v \in \mathcal{D}$ to a global variable $x \in \mathcal{G}$.
(2) A *read event* r reads the value $v \in \mathcal{D}$ of a global variable $x \in \mathcal{G}$.

Additionally, threads can execute local events which do not access global variables and thus are not modeled explicitly.

Given an event e, we denote by $\mathsf{thr}(e)$ its thread and by $\mathsf{var}(e)$ its global variable. We denote by $\mathcal{E}$ the set of all events, and by $\mathcal{R}$ ($\mathcal{W}$) the set of read (write) events. Given two events $e_1, e_2 \in \mathcal{E}$, we say that they *conflict*, denoted $e_1 \bowtie e_2$, if they access the same global variable and at least one of them is a write event.

Concurrent Program Semantics. The semantics of $\mathcal{H}$ are defined by means of a transition system over a state space of global states. A global state consists of (i) a memory function that maps every global variable to a value, and (ii) a local state for each thread, which contains the values of the local variables and the program counter of the thread. We consider the standard setting of Sequential Consistency (SC), and refer to [14] for formal details. As usual, $\mathcal{H}$ is execution-bounded, which means that the state space is finite and acyclic.

Event Sets. Given a set of events $X \subseteq \mathcal{E}$, we write $\mathcal{R}(X) = X \cap \mathcal{R}$ for the set of read events of X, and $\mathcal{W}(X) = X \cap \mathcal{W}$ for the set of write events of X. Given a set of events $X \subseteq \mathcal{E}$ and a thread thr, we denote by X_{thr} and $X_{\neq \mathsf{thr}}$ the events of thr, and the events of all other threads in X, respectively.

Sequences and Traces. Given a sequence of events $\tau = e_1, \dots, e_j$, we denote by $\mathcal{E}(\tau)$ the set of events that appear in τ. We further denote $\mathcal{R}(\tau) = \mathcal{R}(\mathcal{E}(\tau))$ and $\mathcal{W}(\tau) = \mathcal{W}(\mathcal{E}(\tau))$.

Given a sequence τ and two events $e_1, e_2 \in \mathcal{E}(\tau)$, we write $e_1 <_\tau e_2$ when e_1 appears before e_2 in τ, and $e_1 \leq_\tau e_2$ to denote that $e_1 <_\tau e_2$ or $e_1 = e_2$. Given a sequence τ and a set of events A, we denote by $\tau|A$ the *projection* of τ on A, which is the unique subsequence of τ that contains all events of $A \cap \mathcal{E}(\tau)$, and only those events. Given a sequence τ and a thread thr, let τ_{thr} be the subsequence of τ with events of thr, i.e., $\tau|\mathcal{E}(\tau)_{\mathsf{thr}}$. Given two sequences τ_1 and τ_2, we denote by $\tau_1 \circ \tau_2$ the sequence that results in appending τ_2 after τ_1.

A (concrete, concurrent) *trace* is a sequence of events σ that corresponds to a concrete valid execution of $\mathcal{H}$. We let $\mathsf{enabled}(\sigma)$ be the set of enabled events after σ is executed, and call σ *maximal* if $\mathsf{enabled}(\sigma) = \emptyset$. As $\mathcal{H}$ is bounded, all executions of $\mathcal{H}$ are finite and the length of the longest execution in $\mathcal{H}$ is a parameter of the input.

Reads-From and Value Functions. Given a sequence of events τ, we define the *reads-from function* of τ, denoted $\mathsf{RF}_\tau\colon \mathcal{R}(\tau) \to \mathcal{W}(\tau)$, as follows. Given a read event $r \in \mathcal{R}(\tau)$, we have that $\mathsf{RF}_\tau(r)$ is the latest write (of any thread) conflicting with r and occurring before r in τ, i.e., (i) $\mathsf{RF}_\tau(r) \bowtie r$, (ii) $\mathsf{RF}_\tau(r) <_\tau r$, and (iii) for each $\overline{w} \in \mathcal{W}(\tau)$ such that $\overline{w} \bowtie r$ and $\overline{w} <_\tau r$, we have $\overline{w} \leq_\tau \mathsf{RF}_\tau(r)$. We say that r reads-from $\mathsf{RF}_\tau(r)$ in τ. For simplicity, we assume that $\mathcal{H}$ has an initial salient write event on each variable.

Further, given a trace σ, we define the *value function* of σ, denoted $\mathsf{val}_\sigma\colon \mathcal{E}(\sigma) \to \mathcal{D}$, such that $\mathsf{val}_\sigma(e)$ is the value of the global variable $\mathsf{var}(e)$ after the prefix of σ up to and including e has been executed. Intuitively, $\mathsf{val}_\sigma(e)$ captures the value that a read (resp. write) event e shall read (resp. write) in σ. The value function val_σ is well-defined as σ is a valid trace and the threads of $\mathcal{H}$ are deterministic.

2.2 Partial Orders

In this section we present relevant notation around partial orders, which are a central object in this work.

Partial Orders. Given a set of events $X \subseteq \mathcal{E}$, a *(strict) partial order* P over X is an irreflexive, antisymmetric and transitive relation over X (i.e., $<_P \subseteq X \times X$). Given two events $e_1, e_2 \in X$, we write $e_1 \leq_P e_2$ to denote that $e_1 <_P e_2$ or $e_1 = e_2$. Two distinct events $e_1, e_2 \in X$ are *unordered* by P, denoted $e_1 \parallel_P e_2$, if neither $e_1 <_P e_2$ nor $e_2 <_P e_1$, and *ordered* (denoted $e_1 \not\parallel_P e_2$) otherwise. Given a set $Y \subseteq X$, we denote by $P|Y$ the *projection* of P on the set Y, where for every pair of events $e_1, e_2 \in Y$, we have that $e_1 <_{P|Y} e_2$ iff $e_1 <_P e_2$. Given two partial orders P and Q over a common set X, we say that Q *refines* P, denoted by $Q \sqsubseteq P$, if for every pair of events $e_1, e_2 \in X$, if $e_1 <_P e_2$ then $e_1 <_Q e_2$. A *linearization* of P is a total order that refines P.

Lower Sets. Given a pair (X, P), where X is a set of events and P is a partial order over X, a *lower set* of (X, P) is a set $Y \subseteq X$ such that for every event $e_1 \in Y$ and event $e_2 \in X$ with $e_2 \leq_P e_1$, we have $e_2 \in Y$.

Visible Writes. Given a partial order P over a set X, and a read event $r \in \mathcal{R}(X)$, the set of *visible writes* of r is defined as

$$\begin{aligned}\mathsf{VisibleW}_P(r) = \{\, & w \in \mathcal{W}(X) : \text{ (i) } r \bowtie w \text{ and (ii) } r \not<_P w \text{ and (iii) for each} \\ & w' \in \mathcal{W}(X) \text{ with } r \bowtie w', \text{ if } w <_P w' \text{ then } w' \not<_P r \,\}\end{aligned}$$

i.e., the set of write events w conflicting with r that are not “hidden” to r by P.

The Program Order PO**.** The *program order* PO of $\mathcal{H}$ is a partial order $<_{\mathsf{PO}} \subseteq \mathcal{E} \times \mathcal{E}$ that defines a fixed order between some pairs of events of the same thread, reflecting the semantics of $\mathcal{H}$.

A set of events $X \subseteq \mathcal{E}$ is *proper* if (i) it is a lower set of $(\mathcal{E}, \mathsf{PO})$, and (ii) for each thread thr, the events X_{thr} are totally ordered in PO (i.e., for each distinct $e_1, e_2 \in X_{\mathsf{thr}}$ we have $e_1 \not\parallel_{\mathsf{PO}} e_2$). A sequence τ is *well-formed* if (i) its set of events $\mathcal{E}(\tau)$ is proper, and (ii) τ respects the program order (formally, $\tau \sqsubseteq \mathsf{PO}|\mathcal{E}(\tau)$).

Every trace σ of $\mathcal{H}$ is well-formed, as it corresponds to a concrete valid execution of $\mathcal{H}$. Each event of $\mathcal{H}$ is then uniquely identified by its PO predecessors, and by the values its PO predecessor reads have read.

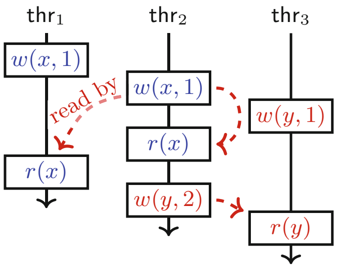

Fig. 2. A trace σ, the displayed events $\mathcal{E}(\sigma)$ are vertically ordered as they appear in σ. The solid black edges represent the program order PO. The dashed red edges represent the reads-from function RF_σ. The transitive closure of all the edges then gives us the causally-happens-before partial order $\hookrightarrow_\sigma$.

Causally-Happens-Before Partial Orders. A trace σ induces a *causally-happens-before* partial order $\hookrightarrow_\sigma \subseteq \mathcal{E}(\sigma) \times \mathcal{E}(\sigma)$, which is the weakest partial order such that (i) it refines the program order (i.e., $\hookrightarrow_\sigma \sqsubseteq \mathsf{PO}|\mathcal{E}(\sigma)$), and (ii) for every read event $r \in \mathcal{R}(\sigma)$, its reads-from $\mathsf{RF}_\sigma(r)$ is ordered before it (i.e., $\mathsf{RF}_\sigma(r) \hookrightarrow_\sigma r$). Intuitively, $\hookrightarrow_\sigma$ contains the causal orderings in σ, i.e., it captures the flow of write events into read events in σ together with the program order. Figure 2 presents an example of a trace and its causal orderings.

3 Reads-Value-From Equivalence

In this section we present our new equivalence on traces, called the *reads-value-from* equivalence (RVF equivalence, or $\sim_{\mathsf{RVF}}$, for short). Then we illustrate that $\sim_{\mathsf{RVF}}$ has some desirable properties for stateless model checking.

Reads-Value-From Equivalence. Given two traces σ_1 and σ_2, we say that they are *reads-value-from-equivalent*, written $\sigma_1 \sim_{\mathsf{RVF}} \sigma_2$, if the following hold.

(1) $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$, i.e., they consist of the same set of events.
(2) $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$, i.e., each event reads resp. writes the same value in both.
(3) $\hookrightarrow_{\sigma_1}|\mathcal{R} = \hookrightarrow_{\sigma_2}|\mathcal{R}$, i.e., their causal orderings agree on the read events.

Figure 3 presents an intuitive example of RVF-(in)equivalent traces.

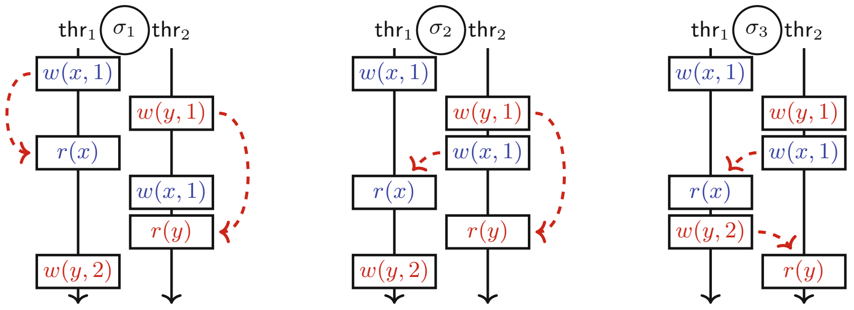

Fig. 3. Three traces σ_1, σ_2, σ_3, events of each trace are vertically ordered as they appear in the trace. Traces σ_1 and σ_2 are RVF-equivalent ($\sigma_1 \sim_{\mathsf{RVF}} \sigma_2$), as they have the same events, same value function, and the two read events are causally unordered in both. Trace σ_3 is not RVF-equivalent with either of σ_1 and σ_2. Compared to σ_1 resp. σ_2, the value function of σ_3 differs ($r(y)$ reads a different value), and the causal orderings of the reads differ ($r(x) \mapsto_{\sigma_3} r(y)$).

Soundness. The RVF equivalence induces a partitioning on the maximal traces of $\mathcal{H}$. Any algorithm that explores each class of this partitioning provably discovers every reachable local state of every thread, and thus RVF is a sound equivalence for local safety properties, such as assertion violations, in the same spirit as in other recent works [21–24]. This follows from the fact that for any two traces σ_1 and σ_2 with $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$ and $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$, the local states of each thread are equal after executing σ_1 and σ_2.

$$\begin{tikzcd}
 & \textbf{reads-from[22,28]} \arrow[dl, dashed] & & \\
\textbf{reads-value-from} & & \textbf{data-centric[21]} \arrow[ul, dashed] \arrow[dl, dashed] & \textbf{Mazurkiewicz[14,15,29]} \arrow[l] \\
 & \textbf{value-centric[23]} \arrow[ul, dashed] & &
\end{tikzcd}$$

Fig. 4. SMC trace equivalences. An edge from X to Y signifies that Y is always at least as coarse, and sometimes coarser, than X.

Coarseness. Here we describe the coarseness properties of the RVF equivalence, as compared to other equivalences used by state-of-the-art approaches in stateless model checking. Figure 4 summarizes the comparison.

The SMC algorithms of [22] and [28] operate on a *reads-from equivalence*, which deems two traces σ_1 and σ_2 equivalent if

(1) they consist of the same events ($\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$), and
(2) their reads-from functions coincide ($\mathsf{RF}_{\sigma_1} = \mathsf{RF}_{\sigma_2}$).

The above two conditions imply that the induced causally-happens-before partial orders are equal, i.e., $\mapsto_{\sigma_1} = \mapsto_{\sigma_2}$, and thus trivially also $\mapsto_{\sigma_1}|\mathcal{R} = \mapsto_{\sigma_2}|\mathcal{R}$.

Further, by a simple inductive argument the value functions of the two traces are also equal, i.e., $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$. Hence any two reads-from-equivalent traces are also RVF-equivalent, which makes the RVF equivalence always at least as coarse as the reads-from equivalence.

The work of [23] utilizes a *value-centric equivalence*, which deems two traces equivalent if they satisfy all the conditions of our RVF equivalence, and also some further conditions (note that these conditions are necessary for correctness of the SMC algorithm in [23]). Thus the RVF equivalence is trivially always at least as coarse. The value-centric equivalence preselects a single thread thr, and then requires two extra conditions for the traces to be equivalent, namely:

(1) For each read of thr, either the read reads-from a write of thr in both traces, or it does not read-from a write of thr in either of the two traces.
(2) For each conflicting pair of events not belonging to thr, the ordering of the pair is equal in the two traces.

Both the reads-from equivalence and the value-centric equivalence are in turn as coarse as the *data-centric equivalence* of [21]. Given two traces, the data-centric equivalence has the equivalence conditions of the reads-from equivalence, and additionally, it preselects a single thread thr (just like the value-centric equivalence) and requires the second extra condition of the value-centric equivalence, i.e., equality of orderings for each conflicting pair of events outside of thr.

Finally, the data-centric equivalence is as coarse as the classical *Mazurkiewicz equivalence* [13], the baseline equivalence for stateless model checking [14,15,29]. Mazurkiewicz equivalence deems two traces equivalent if they consist of the same set of events and they agree on their ordering of conflicting events.

While RVF is always at least as coarse, it can be (even exponentially) coarser, than each of the other above-mentioned equivalences. We illustrate this in Appendix B of [30]. We summarize these observations in the following proposition.

Proposition 1. RVF *is at least as coarse as each of the Mazurkiewicz equivalence* [15]*, the data-centric equivalence* [21]*, the reads-from equivalence* [22]*, and the value-centric equivalence* [23]*. Moreover,* RVF *can be exponentially coarser than each of these equivalences.*

In this work we develop our SMC algorithm RVF-SMC around the RVF equivalence, with the guarantee that the algorithm explores at most one maximal trace per class of the RVF partitioning, and thus can perform significantly fewer steps than algorithms based on the above equivalences. To utilize RVF, the algorithm in each step solves an instance of the verification of sequential consistency problem, which we tackle in the next section. Afterwards, we present RVF-SMC.

4 Verifying Sequential Consistency

In this section we present our contributions towards the problem of verifying sequential consistency (VSC). We present an algorithm VerifySC for VSC, and we show how it can be efficiently used in stateless model checking.

The VSC **Problem.** Consider an input pair (X, GoodW) where

(1) $X \subseteq \mathcal{E}$ is a proper set of events, and
(2) $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$ is a good-writes function such that $w \in \mathsf{GoodW}(r)$ only if $r \bowtie w$.

A *witness* of (X, GoodW) is a linearization τ of X (i.e., $\mathcal{E}(\tau) = X$) respecting the program order (i.e., $\tau \sqsubseteq \mathsf{PO}|X$), such that each read $r \in \mathcal{R}(\tau)$ reads-from one of its good-writes in τ, formally $\mathsf{RF}_\tau(r) \in \mathsf{GoodW}(r)$ (we then say that τ *satisfies* the good-writes function GoodW). The task is to decide whether (X, GoodW) has a witness, and to construct one in case it exists.

VSC **in Stateless Model Checking.** The VSC problem naturally ties in with our SMC approach enumerating the equivalence classes of the RVF trace partitioning. In our approach, we shall generate instances (X, GoodW) such that (i) each witness σ of (X, GoodW) is a valid program trace, and (ii) all witnesses σ_1, σ_2 of (X, GoodW) are pairwise RVF-equivalent ($\sigma_1 \sim_{\mathsf{RVF}} \sigma_2$).

Hardness of VSC**.** Given an input (X, GoodW) to the VSC problem, let $n = |X|$, let k be the number of threads appearing in X, and let d be the number of variables accessed in X. The classic work of [25] establishes two important lower bounds on the complexity of VSC:

(1) VSC is NP-hard even when restricted only to inputs with $k = 3$.
(2) VSC is NP-hard even when restricted only to inputs with $d = 2$.

The first bound eliminates the possibility of any algorithm with time complexity $O(n^{f(k)})$, where f is an arbitrary computable function. Similarly, the second bound eliminates algorithms with complexity $O(n^{f(d)})$ for any computable f.

In this work we show that the problem is parameterizable in $k + d$, and thus admits efficient (polynomial-time) solutions when both variables are bounded.

4.1 Algorithm for VSC

In this section we present our algorithm VerifySC for the problem VSC. First we define some relevant notation. In our definitions we consider a fixed input pair (X, GoodW) to the VSC problem, and a fixed sequence τ with $\mathcal{E}(\tau) \subseteq X$.

Active Writes. A write $w \in \mathcal{W}(\tau)$ is *active* in τ if it is the last write of its variable in τ. Formally, for each $w' \in \mathcal{W}(\tau)$ with $\mathsf{var}(w') = \mathsf{var}(w)$ we have $w' \leq_\tau w$. We can then say that w is the active write of the variable $\mathsf{var}(w)$ in τ.

Held Variables. A variable $x \in \mathcal{G}$ is *held* in τ if there exists a read $r \in \mathcal{R}(X) \setminus \mathcal{E}(\tau)$ with $\mathsf{var}(r) = x$ such that for each its good-write $w \in \mathsf{GoodW}(r)$ we have $w \in \tau$. In such a case we say that r *holds* x in τ. Note that several distinct reads may hold a single variable in τ.

Executable Events. An event $e \in \mathcal{E}(X) \setminus \mathcal{E}(\tau)$ is *executable* in τ if $\mathcal{E}(\tau) \cup \{e\}$ is a lower set of (X, PO) and the following hold.

(1) If e is a read, it has an active good-write $w \in \mathsf{GoodW}(e)$ in τ.
(2) If e is a write, its variable $\mathsf{var}(e)$ is not held in τ.

Memory Maps. A *memory map* of τ is a function from global variables to thread indices $\text{MMap}_\tau \colon \mathcal{G} \to [k]$ where for each variable $x \in \mathcal{G}$, the map $\text{MMap}_\tau(x)$ captures the thread of the active write of x in τ.

Witness States. The sequence τ is a *witness prefix* if the following hold.

(1) τ is a witness of $(\mathcal{E}(\tau), \mathsf{GoodW}|\mathcal{R}(\tau))$.
(2) For each $r \in X \setminus \mathcal{R}(\tau)$ that holds its variable $\mathsf{var}(r)$ in τ, one of its good-writes $w \in \mathsf{GoodW}(r)$ is active in τ.

Intuitively, τ is a witness prefix if it satisfies all VSC requirements modulo its events, and if each read not in τ has at least one good-write still available to read-from in potential extensions of τ. For a witness prefix τ we call its corresponding event set and memory map a *witness state.*

Figure 5 provides an example illustrating the above concepts, where for brevity of presentation, the variables are subscripted and the values are not displayed.

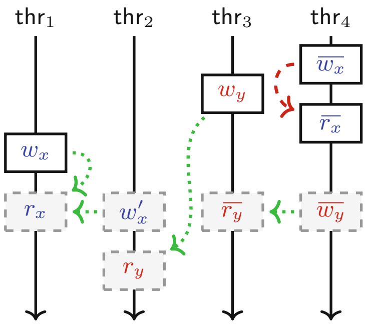

Fig. 5. Event set X, and the good-writes function GoodW denoted by the green dotted edges. The solid nodes are ordered vertically as they appear in τ. The grey dashed nodes are in $X \setminus \mathcal{E}(\tau)$. Events r_x and w'_x are executable in τ. Event $\overline{r_y}$ is not, its good-write is not active in τ. Event $\overline{w_y}$ is also not executable, as its variable y is held by r_y. The memory map of τ is $\text{MMap}_\tau(x) = 1$ and $\text{MMap}_\tau(y) = 3$. τ is a witness prefix, and $\mathcal{E}(\tau)$ with MMap_τ together form its witness state.

Algorithm 1: VerifySC

```
Input: Proper event set $X$ and good-writes function $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$
Output: A witness $\tau$ of $(X, \mathsf{GoodW})$ if $(X, \mathsf{GoodW})$ has a witness, else $\tau = \bot$
1 $\mathcal{S} \leftarrow \{\epsilon\}$; $\mathsf{Done} \leftarrow \{\epsilon\}$
2 while $\mathcal{S} \neq \emptyset$ do
3     Extract a sequence $\tau$ from $\mathcal{S}$
4     if $\mathcal{E}(\tau) = X$ then return $\tau$ ;          // All events executed, witness found
5     foreach event $e$ executable in $\tau$ do
6         Let $\tau_e \leftarrow \tau \circ e$                    // Execute $e$
7         if $\nexists \tau' \in \mathsf{Done}$ s.t. $\mathcal{E}(\tau_e) = \mathcal{E}(\tau')$ and $\mathrm{MMap}_{\tau_e} = \mathrm{MMap}_{\tau'}$ then
8             Insert $\tau_e$ in $\mathcal{S}$ and in $\mathsf{Done}$      // New witness state reached
9 return $\bot$                                               // No witness exists
```

Algorithm. We are now ready to describe our algorithm VerifySC, in Algorithm 1 we present the pseudocode. We attempt to construct a witness of (X, GoodW) by enumerating the witness states reachable by the following process. We start (Line 1) with an empty sequence ϵ as the first witness prefix (and state). We maintain a worklist $\mathcal{S}$ of so-far unprocessed witness prefixes, and a set Done of reached witness states. Then we iteratively obtain new witness prefixes (and states) by considering an already obtained prefix (Line 3) and extending it with each possible executable event (Line 6). Crucially, when we arrive at a sequence τ_e, we include it only if no sequence τ' with equal corresponding witness state has been reached yet (Line 7). We stop when we successfully create a witness (Line 4) or when we process all reachable witness states (Line 9).

Correctness and Complexity. We now highlight the correctness and complexity properties of VerifySC, while we refer to Appendix C of [30] for the proofs. The soundness follows straightforwardly by the fact that each sequence in $\mathcal{S}$ is a witness prefix. This follows from a simple inductive argument that extending a witness prefix with an executable event yields another witness prefix. The completeness follows from the fact that given two witness prefixes τ_1 and τ_2 with equal induced witness state, these prefixes are "equi-extendable" to a witness. Indeed, if a suffix τ^* exists such that $\tau_1 \circ \tau^*$ is a witness of (X, GoodW), then $\tau_2 \circ \tau^*$ is also a witness of (X, GoodW). The time complexity of VerifySC is bounded by $O(n^{k+1} \cdot k^{d+1})$, for n events, k threads and d variables. The bound follows from the fact that there are at most $n^k \cdot k^d$ pairwise distinct witness states. We thus have the following theorem.

Theorem 1. VSC *for n events, k threads and d variables is solvable in $O(n^{k+1} \cdot k^{d+1})$ time. Moreover, if each variable is written by only one thread,* VSC *is solvable in $O(n^{k+1})$ time.*

Implications. We now highlight some important implications of Theorem 1. Although VSC is NP-hard [25], the theorem shows that the problem is parameterizable in $k + d$, and thus in polynomial time when both parameters are bounded. Moreover, even when only k is bounded, the problem is fixed-parameter tractable in d, meaning that d only exponentiates a constant as opposed to n (e.g., we have a polynomial bound even when $d = \log n$). Finally, the algorithm is polynomial for a fixed number of threads regardless of d, when every memory location is written by only one thread (e.g., in producer-consumer settings, or in the concurrent-read-exclusive-write (CREW) concurrency model). These important facts brought forward by Theorem 1 indicate that VSC is likely to be efficiently solvable in many practical settings, which in turn makes RVF a good equivalence for SMC.

4.2 Practical Heuristics for VerifySC in SMC

We now turn our attention to some practical heuristics that are expected to further improve the performance of VerifySC in the context of SMC.

1. Limiting the Search Space. We employ two straightforward improvements to VerifySC that significantly reduce the search space in practice. Consider the for-loop in Line 5 of Algorithm 1 enumerating the possible extensions of τ. This enumeration can be sidestepped by the following two greedy approaches.

(1) If there is a read r executable in τ, then extend τ with r and do not enumerate other options.
(2) Let $\overline{w}$ be an active write in τ such that $\overline{w}$ is not a good-write of any $r \in \mathcal{R}(X) \setminus \mathcal{E}(\tau)$. Let $w \in \mathcal{W}(X) \setminus \mathcal{E}(\tau)$ be a write of the same variable ($\mathsf{var}(w) = \mathsf{var}(\overline{w})$), note that w is executable in τ. If w is also not a good-write of any $r \in \mathcal{R}(X) \setminus \mathcal{E}(\tau)$, then extend τ with w and do not enumerate other options.

The enumeration of Line 5 then proceeds only if neither of the above two techniques can be applied for τ. This extension of VerifySC preserves completeness (not only when used during SMC, but in general), and it can be significantly faster in practice. For clarity of presentation we do not fully formalize this extended version, as its worst-case complexity remains the same.

2. Closure. We introduce *closure*, a low-cost filter for early detection of VSC instances (X, GoodW) with no witness. The notion of closure, its beneficial properties and construction algorithms are well-studied for the *reads-from consistency verification* problems [21,22,31], i.e., problems where a desired reads-from function is provided as input instead of a desired good-writes function GoodW. Further, the work of [23] studies closure with respect to a good-writes function, but only for partial orders of Mazurkiewicz width 2 (i.e., for partial orders with no triplet of pairwise conflicting and pairwise unordered events). Here we define closure for all good-writes instances (X, GoodW), with the underlying partial order (in our case, the program order PO) of arbitrary Mazurkiewicz width.

Given a VSC instance (X, GoodW), its closure $P(X)$ is the weakest partial order that refines the program order ($P \sqsubseteq \mathsf{PO}|X$) and further satisfies the following conditions. Given a read $r \in \mathcal{R}(X)$, let $Cl(r) = \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_P(r)$. The following must hold.

(1) $Cl(r) \neq \emptyset$.
(2) If $(Cl(r), P|Cl(r))$ has a least element w, then $w <_P r$.
(3) If $(Cl(r), P|Cl(r))$ has a greatest element w, then for each $\overline{w} \in \mathcal{W}(X) \setminus \mathsf{GoodW}(r)$ with $r \bowtie \overline{w}$, if $\overline{w} <_P r$ then $\overline{w} <_P w$.
(4) For each $\overline{w} \in \mathcal{W}(X) \setminus \mathsf{GoodW}(r)$ with $r \bowtie \overline{w}$, if each $w \in Cl(r)$ satisfies $w <_P \overline{w}$, then we have $r <_P \overline{w}$.

If (X, GoodW) has no closure (i.e., there is no P with the above conditions), then (X, GoodW) provably has no witness. If (X, GoodW) has closure P, then each witness τ of VSC(X, GoodW) provably refines P (i.e., $\tau \sqsubseteq P$).

Finally, we explain how closure can be used by VerifySC. Given an input (X, GoodW), the closure procedure is carried out before VerifySC is called. Once the closure P of (X, GoodW) is constructed, since each solution of VSC(X, GoodW) has to refine P, we restrict VerifySC to only consider sequences refining P. This is ensured by an extra condition in Line 5 of Algorithm 1, where we proceed with an event e only if it is minimal in P restricted to events not yet in the sequence. This preserves completeness, while further reducing the search space to consider for VerifySC.

3. VerifySC Guided by Auxiliary Trace. In our SMC approach, each time we generate a VSC instance (X, GoodW), we further have available an auxiliary trace $\widetilde{\sigma}$. In $\widetilde{\sigma}$, either all-but-one, or all, good-writes conditions of GoodW are satisfied. If all good writes in GoodW are satisfied, we already have $\widetilde{\sigma}$ as a witness of (X, GoodW) and hence we do not need to run VerifySC at all. On the other hand, if case all-but-one are satisfied, we use $\widetilde{\sigma}$ to guide the search of VerifySC, as described below.

We guide the search by deciding the order in which we process the sequences of the worklist $\mathcal{S}$ in Algorithm 1. We use the auxiliary trace $\widetilde{\sigma}$ with $\mathcal{E}(\widetilde{\sigma}) = X$. We use $\mathcal{S}$ as a last-in-first-out stack, that way we search for a witness in a depth-first fashion. Then, in Line 5 of Algorithm 1 we enumerate the extension events in the reverse order of how they appear in $\widetilde{\sigma}$. We enumerate in reverse order, as each resulting extension is pushed into our worklist $\mathcal{S}$, which is a stack (last-in-first-out). As a result, in Line 3 of the subsequent iterations of the main while loop, we pop extensions from $\mathcal{S}$ in order induced by $\widetilde{\sigma}$.

5 Stateless Model Checking

We are now ready to present our SMC algorithm RVF-SMC that uses RVF to model check a concurrent program. RVF-SMC is a sound and complete algorithm for local safety properties, i.e., it is guaranteed to discover all local states that each thread visits.

RVF-SMC is a recursive algorithm. Each recursive call of RVF-SMC is argumented by a tuple $(X, \mathsf{GoodW}, \sigma, \mathcal{C})$ where:

(1) X is a proper set of events.
(2) $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$ is a desired good-writes function.
(3) σ is a valid trace that is a witness of (X, GoodW).

(4) $\mathcal{C}: \mathcal{R} \to \text{Threads} \to \mathbb{N}$ is a partial function called *causal map* that tracks implicitly, for each read r, the writes that have already been considered as reads-from sources of r.

Further, we maintain a function $\mathsf{ancestors}: \mathcal{R}(X) \to \{\mathsf{true}, \mathsf{false}\}$, where for each read $r \in \mathcal{R}(X)$, $\mathsf{ancestors}(r)$ stores a boolean *backtrack signal* for r. We now provide details on the notions of causal maps and backtrack signals.

Causal Maps. The causal map $\mathcal{C}$ serves to ensure that no more than one maximal trace is explored per RVF partitioning class. Given a read $r \in \mathsf{enabled}(\sigma)$ enabled in a trace σ, we define $\mathsf{forbids}_\sigma^{\mathcal{C}}(r)$ as the set of writes in σ such that $\mathcal{C}$ forbids r to read-from them. Formally, $\mathsf{forbids}_\sigma^{\mathcal{C}}(r) = \emptyset$ if $r \notin \mathsf{dom}(\mathcal{C})$, otherwise $\mathsf{forbids}_\sigma^{\mathcal{C}}(r) = \{w \in \mathcal{W}(\sigma) \mid w \text{ is within first } \mathcal{C}(r)(\mathsf{thr}(w)) \text{ events of } \sigma_{\mathsf{thr}}\}$. We say that a trace σ *satisfies* $\mathcal{C}$ if for each $r \in \mathcal{R}(\sigma)$ we have $\mathsf{RF}_\sigma(r) \notin \mathsf{forbids}_\sigma^{\mathcal{C}}(r)$.

Backtrack Signals. Each call of RVF-SMC (with its GoodW) operates with a trace $\widetilde{\sigma}$ satisfying GoodW that has only reads as enabled events. Consider one of those enabled reads $r \in \mathsf{enabled}(\widetilde{\sigma})$. Each maximal trace satisfying GoodW shall contain r, and further, one of the following two cases is true:

(1) In all maximal traces σ' satisfying GoodW, we have that r reads-from some write of $\mathcal{W}(\widetilde{\sigma})$ in σ'.
(2) There exists a maximal trace σ' satisfying GoodW, such that r reads-from a write not in $\mathcal{W}(\widetilde{\sigma})$ in σ'.

Whenever we can prove that the first above case is true for r, we can use this fact to prune away some recursive calls of RVF-SMC while maintaining completeness. Specifically, we leverage the following crucial lemma, and present the proof in Appendix D of [30].

Lemma 1. *Consider a call* RVF-SMC$(X, \mathsf{GoodW}, \sigma, \mathcal{C})$ *and a trace* $\widetilde{\sigma}$ *extending* σ *maximally such that no event of the extension is a read. Let* $r \in \mathsf{enabled}(\widetilde{\sigma})$ *such that* $r \notin \mathsf{dom}(\mathcal{C})$*. If there exists a trace* σ' *that (i) satisfies* GoodW *and* $\mathcal{C}$*, and (ii) contains* r *with* $\mathsf{RF}_{\sigma'}(r) \notin \mathcal{W}(\widetilde{\sigma})$*, then there exists a trace* $\bar{\sigma}$ *that (i) satisfies* GoodW *and* $\mathcal{C}$*, (ii) contains* r *with* $\mathsf{RF}_{\bar{\sigma}}(r) \in \mathcal{W}(\widetilde{\sigma})$*, and (iii) contains a write* $w \notin \mathcal{W}(\widetilde{\sigma})$ *with* $r \bowtie w$ *and* $\mathsf{thr}(r) \neq \mathsf{thr}(w)$*.*

We then compute a boolean *backtrack signal* for a given RVF-SMC call and read $r \in \mathsf{enabled}(\widetilde{\sigma})$ to capture satisfaction of the consequent of Lemma 1. If the computed backtrack signal is false, we can safely stop the RVF-SMC exploration of this specific call and backtrack to its recursion parent.

Algorithm. We are now ready to describe our algorithm RVF-SMC in detail, Algorithm 2 captures the pseudocode of RVF-SMC$(X, \mathsf{GoodW}, \sigma, \mathcal{C})$. First, in Line 1 we extend σ to $\widetilde{\sigma}$ maximally such that no event of the extension is a read. Then in Lines 2–5 we update the backtrack signals for ancestors of our current recursion call. After this, in Lines 6–11 we construct a sequence of reads enabled in $\widetilde{\sigma}$. Finally, we proceed with the main while-loop in Line 13. In each while-loop iteration we process an enabled read r (Line 14), and we perform no more while-loop iterations in case we receive a false backtrack signal for r.

Algorithm 2: RVF-SMC(X, GoodW, σ, $\mathcal{C}$)

```
Input: Proper set of events $X$, good-writes function GoodW, valid trace $\sigma$ that
       is a witness of $(X, \mathsf{GoodW})$, causal map $\mathcal{C}$.
1  $\widetilde{\sigma} \leftarrow \sigma \circ \widehat{\sigma}$ where $\widehat{\sigma}$ extends $\sigma$ maximally such that no event of $\widehat{\sigma}$ is a read
2  foreach $w \in \mathcal{E}(\widehat{\sigma})$ do                       // All extension events are writes
3      foreach $r \in \mathsf{dom}(\mathsf{ancestors})$ do                 // All ancestor mutations are reads
4          if $r \bowtie w$ and $\mathsf{thr}(r) \neq \mathsf{thr}(w)$ then   // Potential new source for $r$ to read-from
5              $\mathsf{ancestors}(r) \leftarrow \mathsf{true}$               // Set backtrack signal to true
6  $\mathsf{mutate} \leftarrow \epsilon$                                  // Construct a sequence of enabled reads
7  foreach $r \in \mathsf{enabled}(\widetilde{\sigma})$ do                  // Enabled events in $\widetilde{\sigma}$ are reads
8      if $r \in \mathsf{dom}(\mathcal{C})$ then                          // Causal map $\mathcal{C}$ is defined for $r$
9          $\mathsf{mutate} \leftarrow \mathsf{mutate} \circ r$                // Insert $r$ to the end of mutate
10     else                                                 // Causal map $\mathcal{C}$ is undefined for $r$
11         $\mathsf{mutate} \leftarrow r \circ \mathsf{mutate}$                // Insert $r$ to the beginning of mutate
12 $\mathsf{backtrack} \leftarrow \mathsf{true}$
13 while $\mathsf{backtrack} = \mathsf{true}$ and $\mathsf{mutate} \neq \epsilon$ do
14     $r \leftarrow$ pop front of mutate                          // Process next read of mutate
15     if $r \notin \mathsf{dom}(\mathcal{C})$ then
16         $\mathsf{backtrack} \leftarrow \mathsf{false}$
17     $\mathrm{F}_r \leftarrow \mathsf{VisibleW}_{\mathsf{PO}|\mathcal{E}(\widetilde{\sigma})}(r) \setminus \mathsf{forbids}^{\mathcal{C}}_{\widetilde{\sigma}}(r)$   // Visible writes not forbidden by $\mathcal{C}$
18     $\mathcal{D}_r \leftarrow \{\mathsf{val}_{\widetilde{\sigma}}(w) : w \in \mathrm{F}_r\}$            // The set of values that $r$ may read
19     foreach $v \in \mathcal{D}_r$ do                               // Process each value
20         $X' \leftarrow X \cup \mathcal{E}(\widetilde{\sigma}) \cup \{r\}$                     // New event set
21         $\mathsf{GoodW}' \leftarrow \mathsf{GoodW} \cup \{(r, \{ w \in \mathrm{F}_r \mid \mathsf{val}_{\widetilde{\sigma}}(w) = v \})\}$   // New good-writes
22         $\sigma' \leftarrow \mathrm{VerifySC}(X', \mathsf{GoodW}')$              // VerifySC guided by $\widetilde{\sigma} \circ r$
23         if $\sigma' \neq \bot$ then                          // $(X', \mathsf{GoodW}')$ has a witness
24             $\mathcal{C}' \leftarrow \mathcal{C}$
25             $\mathsf{ancestors}(r) \leftarrow \mathsf{backtrack}$                   // Record ancestor
26             RVF-SMC$(X', \mathsf{GoodW}', \sigma', \mathcal{C}')$
27             $\mathsf{backtrack} \leftarrow \mathsf{ancestors}(r)$             // Retrieve backtrack signal
28             delete $r$ from ancestors                        // Unrecord ancestor
29     foreach thr $\in$ Threads do                  // Update causal map $\mathcal{C}(r)$ for each thread
30         $\mathcal{C}(r)(\mathsf{thr}) \leftarrow |\mathcal{E}(\widetilde{\sigma})_{\mathsf{thr}}|$                // Number of events of thr in $\widetilde{\sigma}$
```

When processing r, first we collect its viable reads-from sources in Line 17, then we group the sources by value they write in Line 18, and then in iterations of the for-loop in Line 19 we consider each value-group. In Line 20 we form the event set, and in Line 21 we form the good-write function that designates the value-group as the good-writes of r. In Line 22 we use VerifySC to generate a witness, and in case it exists, we recursively call RVF-SMC in Line 26 with the newly obtained events, good-write constraint for r, and witness.

To preserve completeness of RVF-SMC, the backtrack-signals technique can be utilized only for reads r with undefined causal map $r \notin \mathsf{dom}(\mathcal{C})$ (cf. Lemma 1). The order of the enabled reads imposed by Lines 6–11 ensures that subsequently, in iterations of the loop in Line 13 we first consider all the reads where we can utilize the backtrack signals. This is an insightful heuristic that often helps in practice, though it does not improve the worst-case complexity.

Thread thr_1

1. $w_1(x, 1)$
2. $r_1(x)$
3. $w_2(y, 1)$

Thread thr_2

1. $w_3(x, 1)$
2. $w_4(y, 1)$
3. $r_2(y)$
4. $w_5(x, 2)$

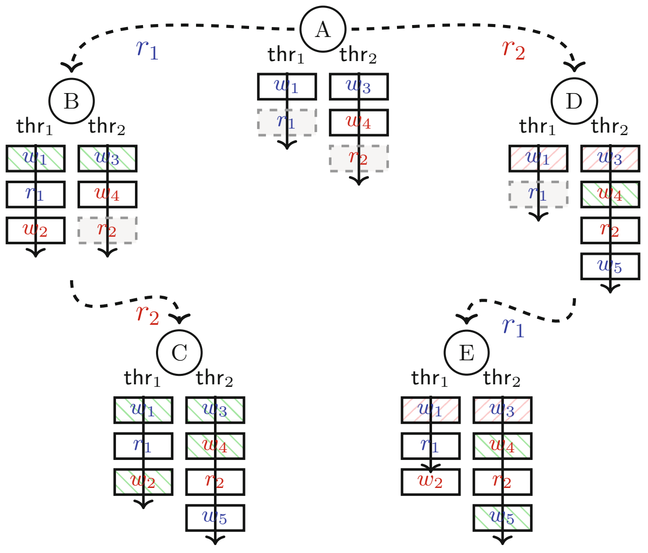

Fig. 6. RVF-SMC (Algorithm 2). Circles represent nodes of the recursion tree. Below each circle is its corresponding event set $\mathcal{E}(\widetilde{\sigma})$ and the enabled reads (dashed grey). Writes with green background are good-writes (GoodW) of its corresponding-variable read. Writes with red background are forbidden by $\mathcal{C}$ for its corresponding-variable read. Dashed arrows represent recursive calls. (Color figure online)

Example. Figure 6 displays a simple concurrent program on the left, and its corresponding RVF-SMC (Algorithm 2) run on the right. We start with RVF-SMC$(\emptyset, \emptyset, \epsilon, \emptyset)$ (A). By performing the extension (Line 1) we obtain the events and enabled reads as shown below (A). First we process read r_1 (Line 14). The read can read-from w_1 and w_3, both write the same value so they are grouped together as good-writes of r_1. A witness is found and a recursive call to (B) is performed. In (B), the only enabled event is r_2. It can read-from w_2 and w_4, both write the same value so they are grouped for r_2. A witness is found, a recursive call to (C) is performed, and (C) concludes with a maximal trace. Crucially, in (C) the event w_5 is discovered, and since it is a potential new reads-from source for r_1, a backtrack signal is sent to (A). Hence after RVF-SMC backtracks to (A), in (A) it needs to perform another iteration of Line 13 while-loop. In (A), first the causal map $\mathcal{C}$ is updated to forbid w_1 and w_3 for r_1. Then, read r_2 is processed from (A), creating (D). In (D), r_1 is the only enabled event, and w_5 is its only $\mathcal{C}$-allowed write. This results in (E) which reports a maximal trace. The algorithm backtracks and concludes, reporting two maximal traces in total.

Theorem 2. *Consider a concurrent program $\mathcal{H}$ of k threads and d variables, with n the length of the longest trace in $\mathcal{H}$.* RVF-SMC *is a sound and complete algorithm for local safety properties in $\mathcal{H}$. The time complexity of* RVF-SMC *is $k^d \cdot n^{O(k)} \cdot \beta$, where β is the size of the* RVF *trace partitioning of $\mathcal{H}$.*

Novelties of the Exploration. Here we highlight some key aspects of RVF-SMC. First, we note that RVF-SMC constructs the traces incrementally with each recursion step, as opposed to other approaches such as [15,22] that always work with maximal traces. The reason of incremental traces is technical and has to do with the value-based treatment of the RVF partitioning. We note that the other two value-based approaches [23,24] also operate with incremental traces. However, RVF-SMC brings certain novelties compared to these two methods. First, the exploration algorithm of [24] can visit the same class of the partitioning (and even the same trace) an exponential number of times by different recursion branches, leading to significant performance degradation. The exploration algorithm of [23] alleviates this issue using the causal map data structure, similar to our algorithm. The causal map data structure can provably limit the number of revisits to polynomial (for a fixed number of threads), and although it offers an improvement over the exponential revisits, it can still affect performance. To further improve performance in this work, our algorithm combines causal maps with a new technique, which is the backtrack signals. Causal maps and backtrack signals together are very effective in avoiding having different branches of the recursion visit the same RVF class.

Beyond RVF Partitioning. While RVF-SMC explores the RVF partitioning in the worst case, in practice it often operates on a partitioning coarser than the one induced by the RVF equivalence. Specifically, RVF-SMC may treat two traces σ_1 and σ_2 with same events ($\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$) and value function ($\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$) as equivalent even when they differ in some causal orderings ($\mapsto_{\sigma_1}|\mathcal{R} \neq \mapsto_{\sigma_2}|\mathcal{R}$). To see an example of this, consider the program and the RVF-SMC run in Fig. 6. The recursion node (C) spans all traces where (i) r_1 reads-from either w_1 or w_3, and (ii) r_2 reads-from either w_2 or w_4. Consider two such traces σ_1 and σ_2, with $\mathsf{RF}_{\sigma_1}(r_2) = w_2$ and $\mathsf{RF}_{\sigma_2}(r_2) = w_4$. We have $r_1 \mapsto_{\sigma_1} r_2$ and $r_1 \not\mapsto_{\sigma_2} r_2$, and yet σ_1 and σ_2 are (soundly) considered equivalent by RVF-SMC. Hence the RVF partitioning is used to upper-bound the time complexity of RVF-SMC. We remark that the algorithm is always sound, i.e., it is guaranteed to discover all thread states even when it does not explore the RVF partitioning in full.

6 Experiments

In this section we describe the experimental evaluation of our SMC approach RVF-SMC. We have implemented RVF-SMC as an extension in Nidhugg [27], a state-of-the-art stateless model checker for multithreaded C/C++ programs that operates on LLVM Intermediate Representation. First we assess the advantages of utilizing the RVF equivalence in SMC as compared to other trace equivalences.

Then we perform ablation studies to demonstrate the impact of the backtrack signals technique (cf. Sect. 5) and the VerifySC heuristics (cf. Sect. 4.2).

In our experiments we compare RVF-SMC with several state-of-the-art SMC tools utilizing different trace equivalences. First we consider VC-DPOR [23], the SMC approach operating on the value-centric equivalence. Then we consider Nidhugg/rfsc [22], the SMC algorithm utilizing the reads-from equivalence. Further we consider DC-DPOR [21] that operates on the data-centric equivalence, and finally we compare with Nidhugg/source [15] utilizing the Mazurkiewicz equivalence.[1] The works of [22] and [32] in turn compare the Nidhugg/rfsc algorithm with additional SMC tools, namely GenMC [28] (with reads-from equivalence), RCMC [29] (with Mazurkiewicz equivalence), and CDSChecker [33] (with Mazurkiewicz equivalence), and thus we omit those tools from our evaluation.

There are two main objectives to our evaluation. First, from Sect. 3 we know that the RVF equivalence can be up to exponentially coarser than the other equivalences, and we want to discover how often this happens in practice. Second, in cases where RVF does provide reduction in the trace-partitioning size, we aim to see whether this reduction is accompanied by the reduction in the runtime of RVF-SMC operating on RVF equivalence.

Setup. We consider 119 benchmarks in total in our evaluation. Each benchmark comes with a scaling parameter, called the *unroll* bound. The parameter controls the bound on the number of iterations in all loops of the benchmark. For each benchmark and unroll bound, we capture the number of explored maximal traces, and the total running time, subject to a timeout of one hour. In Appendix E of [30] we provide further details on our setup.

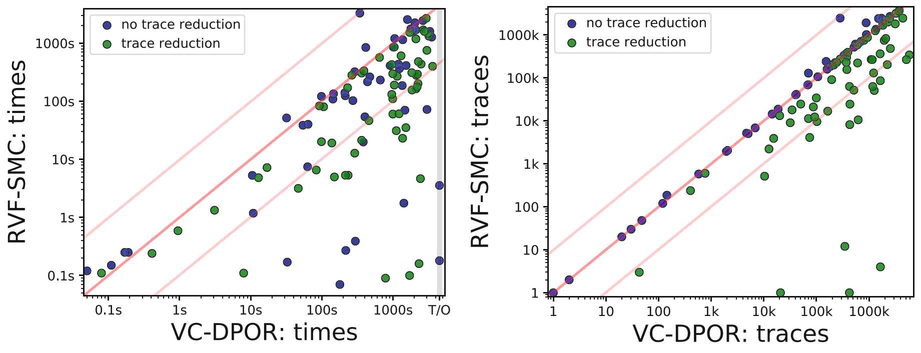

Fig. 7. Runtime and traces comparison of RVF-SMC with VC-DPOR.

[1] The MCR algorithm [24] is beyond the experimental scope of this work, as that tool handles Java programs and uses heavyweight SMT solvers that require fine-tuning.

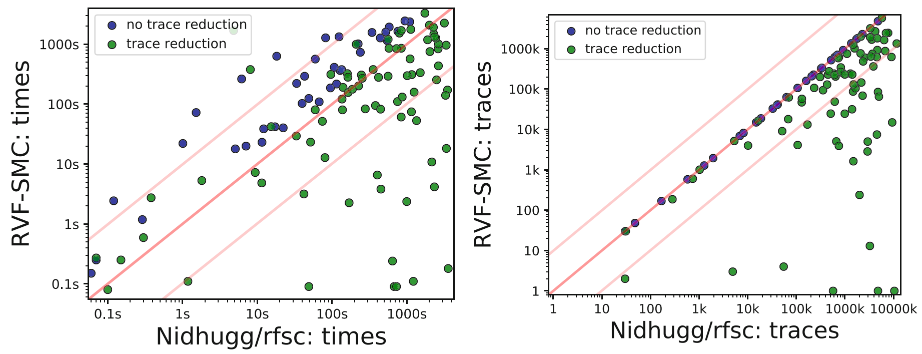

Fig. 8. Runtime and traces comparison of RVF-SMC with Nidhugg/rfsc.

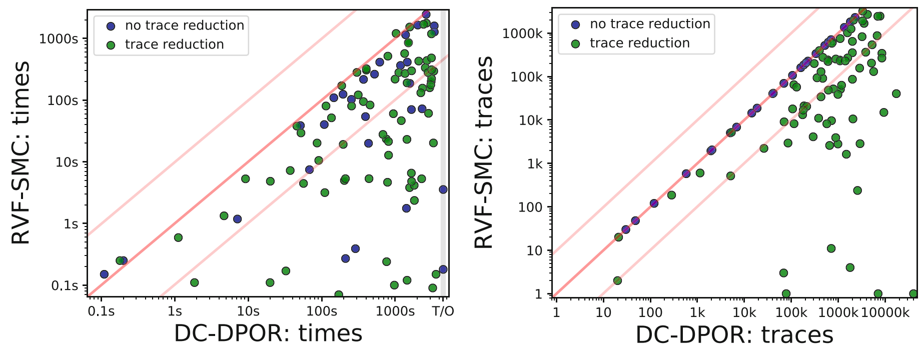

Fig. 9. Runtime and traces comparison of RVF-SMC with DC-DPOR.

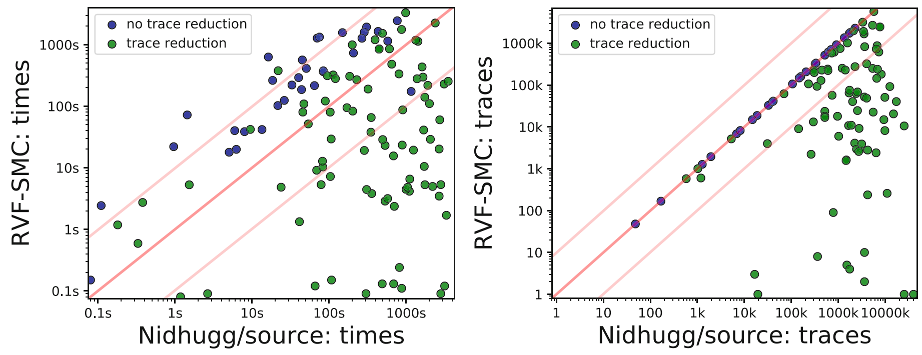

Fig. 10. Runtime and traces comparison of RVF-SMC with Nidhugg/source.

Results. We provide a number of scatter plots summarizing the comparison of RVF-SMC with other state-of-the-art tools. In Fig. 7, Fig. 8, Fig. 9 and Fig. 10 we provide comparison both in runtimes and explored traces, for VC-DPOR, Nidhugg/rfsc, DC-DPOR, and Nidhugg/source, respectively. In each scatter plot, both its axes are log-scaled, the opaque red line represents equality, and the two semi-transparent lines represent an order-of-magnitude difference. The points are colored green when RVF-SMC achieves trace reduction in the underlying benchmark, and blue otherwise.

Discussion: Significant Trace Reduction. In Table 1 we provide the results for several benchmarks where RVF achieves significant reduction in the trace-partitioning size. This is typically accompanied by significant runtime reduction, allowing is to scale the benchmarks to unroll bounds that other tools cannot handle. Examples of this are 27_Boop4 and scull_loop, two toy Linux kernel drivers.

In several benchmarks the number of explored traces remains the same for RVF-SMC even when scaling up the unroll bound, see 45_monabsex1, reorder_5 and singleton in Table 1. The singleton example is further interesting, in that while VC-DPOR and DC-DPOR also explore few traces, they still suffer in runtime due to additional redundant exploration, as described in Sects. 1 and 5.

Table 1. Benchmarks with trace reduction achieved by RVF-SMC. The unroll bound is shown in the column **U**. Symbol "–" indicates one-hour timeout. Bold-font entries indicate the smallest numbers for respective benchmark and unroll.

Benchmark		U	RVF-SMC	VC-DPOR	Nidh/rfsc	DC-DPOR	Nidh/source
27_Boop4 threads: 4	Traces	10	**1337215**	1574287	11610040	–	–
		12	**2893039**	–	–	–	–
	Times	10	**837 s**	1946 s	2616 s	–	–
		12	**2017 s**	–	–	–	–
45_monabsex1 threads: U	Traces	7	**1**	423360	262144	7073803	25401600
		8	**1**	–	4782969	–	–
	Times	7	**0.09 s**	784 s	33 s	3239 s	2819 s
		8	**0.09 s**	–	677 s	–	–
reorder_5 threads: U+1	Traces	9	**4**	1644716	1540	1792290	–
		30	**4**	–	54901	–	–
	Times	9	**0.10 s**	1711 s	0.44 s	974 s	–
		30	**0.09 s**	–	49 s	–	–
scull_loop threads: 3	Traces	2	**3908**	15394	749811	884443	3157281
		3	**115032**	–	–	–	–
	Times	2	**6.55 s**	83 s	403 s	1659 s	1116 s
		3	**266 s**	–	–	–	–
singleton threads: U+1	Traces	20	**2**	**2**	20	20	–
		30	**2**	–	30	–	–
	Times	20	**0.07 s**	179 s	0.08 s	171 s	–
		30	**0.08 s**	–	0.10 s	–	–

Table 2. Benchmarks with little-to-no trace reduction by RVF-SMC. Symbol † indicates that a particular benchmark operation is not handled by the tool.

Benchmark		U	RVF-SMC	VC-DPOR	Nidh/rfsc	DC-DPOR	Nidh/source
13_unverif threads: U	Traces	5	**14400**	**14400**	**14400**	**14400**	**14400**
		6	**518400**	–	**518400**	–	**518400**
	Times	5	7.45 s	63 s	3.33 s	68 s	**2.72 s**
		6	376 s	–	134 s	–	**84 s**
approxds_append threads: U	Traces	6	**50897**	1256381	198936	1114746	9847080
		7	**923526**	–	4645207	–	–
	Times	6	**60 s**	995 s	67 s	944 s	2733 s
		7	2078 s	–	**2003 s**	–	–
chase-lev-dq threads: 3	Traces	4	**87807**	†	175331	†	175331
		5	**227654**	†	448905	†	448905
	Times	4	289 s	†	**71 s**	†	**71 s**
		5	995 s	†	210 s	†	**200 s**
linuxrwlocks threads: U+1	Traces	1	**56**	†	59	†	59
		2	**62018**	†	70026	†	70026
	Times	1	0.12 s	†	**0.09 s**	†	0.13 s
		2	42 s	†	15 s	†	**9.50 s**
pgsql threads: 2	Traces	3	**3906**	**3906**	**3906**	**3906**	**3906**
		4	**335923**	**335923**	**335923**	**335923**	**335923**
	Times	3	3.30 s	5.98 s	1.01 s	4.00 s	**0.51 s**
		4	412 s	911 s	107 s	616 s	**51 s**

Discussion: Little-to-no Trace Reduction. Table 2 presents several benchmarks where the RVF partitioning achieves little-to-no reduction. In these cases the well-engineered Nidhugg/rfsc and Nidhugg/source dominate the runtime.

RVF-SMC Ablation Studies. Here we demonstrate the effect that follows from our RVF-SMC algorithm utilizing the approach of backtrack signals (see Sect. 5) and the heuristics of VerifySC (see Sect. 4.2). These techniques have no effect on the number of the explored traces, thus we focus on the runtime. The left plot of Fig. 11 compares RVF-SMC as is with a RVF-SMC version that does not utilize the backtrack signals (achieved by simply keeping the backtrack flag in Algorithm 2 always true). The right plot of Fig. 11 compares RVF-SMC as is with a RVF-SMC version that employs VerifySC without the closure and auxiliary-trace heuristics. We can see that the techniques almost always result in improved runtime. The improvement is mostly within an order of magnitude, and in a few cases there is several-orders-of-magnitude improvement.

Finally, in Fig. 12 we illustrate how much time during RVF-SMC is typically spent on VerifySC (i.e., on solving VSC instances generated during RVF-SMC).

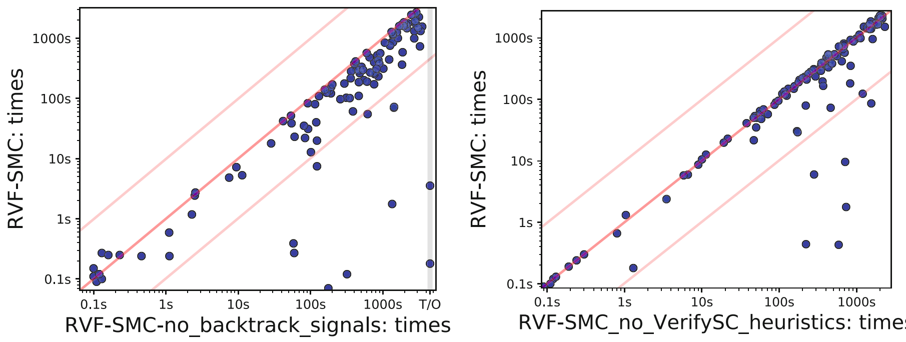

Fig. 11. Ablation studies of RVF-SMC. The left plot compares RVF-SMC with and without backtrack signals. The right plots compares RVF-SMC with and without the closure and auxiliary-trace heuristics of Sect. 4.2.

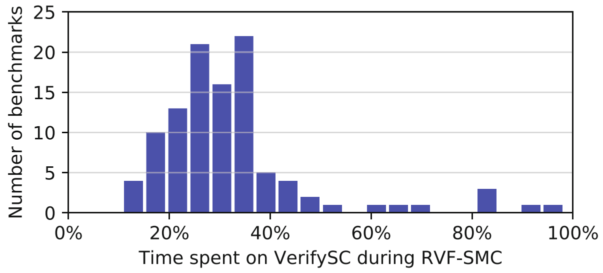

Fig. 12. Histogram that illustrates the percentage of time spent solving VSC instances during RVF-SMC.

7 Conclusions

In this work we developed RVF-SMC, a new SMC algorithm for the verification of concurrent programs using a novel equivalence called *reads-value-from (RVF)*. On our way to RVF-SMC, we have revisited the famous VSC problem [25]. Despite its NP-hardness, we have shown that the problem is parameterizable in $k+d$ (for k threads and d variables), and becomes even fixed-parameter tractable in d when k is constant. Moreover we have developed practical heuristics that solve the problem efficiently in many practical settings.

Our RVF-SMC algorithm couples our solution for VSC to a novel exploration of the underlying RVF partitioning, and is able to model check many concurrent programs where previous approaches time-out. Our experimental evaluation reveals that RVF is very often the most effective equivalence, as the underlying partitioning is exponentially coarser than other approaches. Moreover, RVF-SMC generates representatives very efficiently, as the reduction in the partitioning is often met with significant speed-ups in the model checking task. Interesting future work includes further improvements over the VSC, as well as extensions of RVF-SMC to relaxed memory models.

Acknowledgments. The research was partially funded by the ERC CoG 863818 (ForM-SMArt) and the Vienna Science and Technology Fund (WWTF) through project ICT15-003.

References

1. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI (2008)
2. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
3. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL (1997)
4. Godefroid, P.: Software model checking: the VeriSoft approach. FMSD **26**(2), 77–101 (2005)
5. Ball, T., Musuvathi, M., Qadeer, S.: Chess: a systematic testing tool for concurrent software. Technical report (2007)
6. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Secaucus (1996)
7. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. SIGPLAN Not. **42**(6), 446–455 (2007)
8. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. FMSD **35**(1), 73–97 (2009)
9. Chini, P., Kolberg, J., Krebs, A., Meyer, R., Saivasan, P.: On the complexity of bounded context switching. In: Pruhs, K., Sohler, C. (eds.) 25th Annual European Symposium on Algorithms (ESA 2017), Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, vol. 87, pp. 27:1–27:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. (2017)
10. Baumann, P., Majumdar, R., Thinniyam, R.S., Zetzsche, G.: Context-bounded verification of liveness properties for multithreaded shared-memory programs. In: Proceedings of ACM Programming Language, vol. 5, no. POPL, pp. 1–31 (2021)
11. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. STTT **2**(3), 279–287 (1999)
12. Peled, D.: All from one, one for all: on model checking using representatives. In: CAV (1993)
13. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) ACPN 1986. LNCS, vol. 255, pp. 278–324. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-17906-2_30
14. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL (2005)
15. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL (2014)
16. Nguyen, H.T.T., Rodríguez, C., Sousa, M., Coti, C., Petrucci, L.: Quasi-optimal partial order reduction. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, 14–17 July, 2018, Proceedings, Part II, pp. 354–371 (2018)
17. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 229–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_14

18. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods (extended abstract). In: CAV (1993)
19. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 526–543. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_26
20. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. In: Proceedings of ACM Programming Language, vol. 3, no. OOPSLA, pp. 1–26 (2019)
21. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. In: Proceedings of ACM Programming Language, vol. 2, no. POPL, pp. 31:1–31:30 (2017)
22. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. In: Proceedings of ACM Programming Language, vol. 3, no. OOPSLA, pp. 1–29 (2019)
23. Chatterjee, K., Pavlogiannis, A., Toman, V.: Value-centric dynamic partial order reduction. In: Proceedings of ACM Programming Language, vol. 3, no. OOPSLA, pp. 1–29 (2019)
24. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. In: PLDI (2015)
25. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997)
26. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. In: Proceedings of ACM Programming Language, vol. 2, no. OOPSLA, pp. 135:1–135:29 (2018)
27. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. Acta Informatica **54**(8), 789–818 (2016). https://doi.org/10.1007/s00236-016-0275-0
28. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, New York, NY, USA, pp. 96–110. ACM (2019)
29. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for c/c++ concurrency. In: Proceedings of ACM Programming Language, 2, no. POPL, pp. 17:1–17:32 (2017)
30. Agarwal, P., Chatterjee, K., Pathak, S., Pavlogiannis, A., Toman, V.: Stateless model checking under a reads-value-from equivalence. CoRR/arXiv abs/2105.06424 (2021)
31. Pavlogiannis, A.: Fast, sound, and effectively complete dynamic race prediction. In: Proceedings ACM Programming Language, vol. 4, no. POPL, pp. 1–29 (2019)
32. Lång, M., Sagonas, K.: Parallel graph-based stateless model checking. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 377–393. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_21
33. Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. ACM Trans. Program. Lang. Syst. **38**(3), 10:1–10:51 (2016)