Value-Centric Dynamic Partial Order Reduction

KRISHNENDU CHATTERJEE, IST Austria, Austria

ANDREAS PAVLOGIANNIS, EPFL, Switzerland

VIKTOR TOMAN, IST Austria, Austria

The verification of concurrent programs remains an open challenge, as thread interaction has to be accounted for, which leads to state-space explosion. Stateless model checking battles this problem by exploring traces rather than states of the program. As there are exponentially many traces, dynamic partial-order reduction (DPOR) techniques are used to partition the trace space into equivalence classes, and explore a few representatives from each class. The standard equivalence that underlies most DPOR techniques is the *happens-before* equivalence, however recent works have spawned a vivid interest towards coarser equivalences. The efficiency of such approaches is a product of two parameters: (i) the size of the partitioning induced by the equivalence, and (ii) the time spent by the exploration algorithm in each class of the partitioning.

In this work, we present a new equivalence, called *value-happens-before* and show that it has two appealing features. First, value-happens-before is always at least *as coarse as* the happens-before equivalence, and can be even exponentially coarser. Second, the value-happens-before partitioning is efficiently explorable when the number of threads is bounded. We present an algorithm called *value-centric* DPOR (VC-DPOR), which explores the underlying partitioning using polynomial time per class. Finally, we perform an experimental evaluation of VC-DPOR on various benchmarks, and compare it against other state-of-the-art approaches. Our results show that value-happens-before typically induces a significant reduction in the size of the underlying partitioning, which leads to a considerable reduction in the running time for exploring the whole partitioning.

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

Additional Key Words and Phrases: concurrency, stateless model checking, partial-order reduction

ACM Reference Format:

Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 124 (October 2019), 29 pages. https://doi.org/10.1145/3360550

1 INTRODUCTION

Model checking of concurrent programs. The formal analysis of concurrent programs is a key problem in program analysis and verification. Concurrency incurs a combinatorial explosion in

Authors' addresses: Krishnendu Chatterjee, IST Austria, Am Campus 1, Klosterneuburg, 3400, Austria, krishnendu.chatterjee@ist.ac.at; Andreas Pavlogiannis, EPFL, Route Cantonale, Lausanne, 1015, Switzerland, pavlogiannis@cs.au.dk; Viktor Toman, IST Austria, Am Campus 1, Klosterneuburg, 3400, Austria, viktor.toman@ist.ac.at.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART124
https://doi.org/10.1145/3360550

the behavior of the program, which makes errors hard to reproduce by testing (often identified as *Heisenbugs* [Musuvathi et al. 2008]). Thus, the formal analysis of concurrent program requires a systematic exploration of the state space, which is addressed by *model checking* [Clarke et al. 1999a]. However there are two key issues related to model-checking of concurrent programs: first, is related to the state-space explosion, and second, is related to the number of interleavings. Below we describe the main techniques to address these problems.

Stateless model checking. Model checkers typically store a large number of global states, and cannot handle realistic concurrent programs. The standard solution that is adopted to battle this problem on concurrent programs is *stateless model checking* [Godefroid 1996]. Stateless model-checking methods typically explore traces rather than states of the analyzed program, and only have to store a small number of traces. In such techniques, model checking is achieved by a scheduler, which drives the program execution based on the current interaction between the threads. The depth-first nature of the search enables it to be both systematic and memory-efficient. Stateless model-checking techniques have been employed successfully in several well-established tools, e.g., VeriSoft [Godefroid 1997, 2005] and CHESS [Madan Musuvathi 2007].

Partial-order Reduction (POR). While stateless model checking deals with the state-space issue, one key challenge that remains is exploring efficiently the exponential number of interleavings, which results from non-deterministic interprocess communication. There exist various techniques for reducing the number of explored interleavings, such as depth bounding and context bounding [Lal and Reps 2009; Musuvathi and Qadeer 2007]. One of the most well-studied techniques is *partial-order reduction (POR)* [Clarke et al. 1999b; Godefroid 1996; Peled 1993]. The main principle of POR is that two interleavings can be regarded as equal if they agree on the order of conflicting (dependent) events. In other words, POR considers certain pairs of traces to be equivalent, and the theoretical foundation of POR is the equivalence relation induced on the trace space, known as the *happens-before* (or *Mazurkiewicz)* equivalence $\mathcal{HB}$ [Mazurkiewicz 1987]. POR algorithms explore at least one trace from each equivalence class and guarantee a complete coverage of all behaviors that can occur in any interleaving, while exploring only a subset of the trace space. For the most interesting properties that arise in formal verification, such as safety, race freedom, absence of global deadlocks, and absence of assertion violations, POR-based algorithms make sound reports of correctness [Godefroid 1996].

Dynamic Partial-order Reduction (DPOR). Dynamic partial-order reduction (DPOR) is an on-the-fly version of POR [Flanagan and Godefroid 2005]. DPOR records conflicts that actually occur during the execution of traces, and thus is able to infer independence more frequently than static POR, which typically relies on over-approximations of conflicting events. Similar to POR, DPOR-based algorithms guarantee the exploration of at least one trace in each class of the happens-before partitioning. Recently, an optimal method for DPOR was developed [Abdulla et al. 2014] that explores exactly one trace from each happens-before equivalence class.

Efficiency of DPOR techniques. The efficiency of DPOR algorithms typically depends on two parameters, namely (i) the size of the trace-space partitioning and (ii) the time required to explore each class of the partitioning. The overall efficiency of the algorithm is a product of the two above, and there is usually a trade-off between the two, as coarser partitionings typically make the problem of moving between different classes of the partitioning computationally harder.

Beyond the Mazurkiewicz equivalence. Lately, there has been a considerable effort into going beyond Mazurkiewicz equivalence, by developing algorithms that explore partitionings of the trace space induced by equivalence relations that are coarser than Mazurkiewicz [Albert et al. 2017; Aronis

Thread p_1 :	Thread p_2 :
1. $w(x,1)$	1. $w(x,2)$
	2. $w(x,1)$
	3. $r(x)$

Fig. 1. A toy program with two threads.

et al. 2018; Chalupa et al. 2017; Huang 2015]. Such approaches can be broadly classified as *oracle-based* methods that rely on NP-hard oracles such as SMT-solvers to guide the exploration, and *explicit* methods which avoid such computationally expensive oracles. Explicit methods include DC-DPOR [Chalupa et al. 2017] and Optimal DPOR with Observers [Aronis et al. 2018], which rely on equivalences provably coarser than the happens-before equivalence, as well as Context-sensitive DPOR [Albert et al. 2017] which sometimes might be coarser, but not always. On the other hand, oracle-based methods include MCR [Huang 2015] and SATCheck [Demsky and Lam 2015].

Value-centric DPOR. The happens-before equivalence and most coarser equivalences which admit an efficient exploration are insensitive to the values that variables take during the execution of a trace. On the other hand, it is well-understood that equivalences which are sensitive to such values can be very coarse, thereby reducing the size of the trace-space partitioning. An interesting approach with a value-centric partitioning was recently explored in [Huang 2015]. However, that approach is implicit and relies on expensive NP-oracles repeatedly for guiding the exploration of the partitioning. This NP bottleneck was identified in that work, and was subsequently only partially improved with static-analysis-based heuristics [Huang and Huang 2017]. Hence, the challenge of constructing value-centric equivalences that also admit efficient explorations has remained open. In the next section, we illustrate the benefits of such equivalences on a small example.

1.1 A Small Motivating Example

Consider the simple program given in Figure 1, which consists of two threads communicating over a global variable x. We have two types of events: p_1 writes to x the value 1, whereas p_2 first writes to x the value 2, then writes to x the value 1, and finally it reads the value of x to its local variable. When we analyze this program, it becomes apparent that a model-checking algorithm can benefit if it takes into account the values written by the write events. Indeed, denote by w_i^j the j-th write event of thread i, and by r the unique read event. There exist 4 Mazurkiewicz orderings.

$$t_1 : \; w_1^1 w_2^1 w_2^2 r \qquad t_2 : \; w_2^1 w_1^1 w_2^2 r \qquad t_3 : \; w_2^1 w_2^2 w_1^1 r \qquad t_4 : \; w_2^1 w_2^2 r w_1^1$$

Hence, any algorithm that uses the Mazurkiewicz equivalence for exploring the trace space of the above program will have to explore at least 4 traces. Moreover, any sound algorithm that is insensitive to values will, in general, explore at least two traces (e.g., t_1, and t_3), since the value read by r can, in principle, be different in both cases. This is true, for example, for DC-DPOR [Chalupa et al. 2017], which is based on the recently introduced Observation equivalence, as well as the Optimal DPOR with observers [Aronis et al. 2018] (which explores 3). On the other hand, it is clear that examining a single trace suffices for visiting all the local states of all threads. Although minimal, the above example illustrates the advantage that stateless model checking algorithms can gain by being sensitive to the values used by the events during an execution.

1.2 Challenges and Our Contributions

Challenges. The above example illustrates that a value-centric partitioning can be coarse. To our knowledge, value-centric equivalences have been used systematically (i.e., with provable guarantees) only by implicit methods which rely on NP oracles (e.g., SMT-solvers [Huang 2015; Huang and Huang 2017]), which makes the exploration of the (reduced) partitionings computationally expensive. The challenge that arises naturally is to produce a partitioning that (a) is provably *always* coarser than Mazurkiewicz trace equivalence; and (b) is *efficiently* explorable (i.e., the time required in each step of the search is small/polynomial). In this work we address this challenge.

Our contributions. The main contribution of this work is a new value-centric equivalence, called *value-happens-before* ($\mathcal{VHB}$). Intuitively, $\mathcal{VHB}$ distinguishes (arbitrarily) a thread of the program, called the *root*, from the other threads, called the *leaves*. The coarsening of the partitioning is achieved by $\mathcal{VHB}$ by relaxing the happens-before orderings between events that belong to the root and leaf threads. Given two traces t_1 and t_2 which have the same happens-before ordering on the events of leaf threads, $\mathcal{VHB}$ deems t_1 and t_2 equivalent by using a combination of (i) the *values* and (ii) the *causally-happens-before* orderings on pairs of events between the root and the leaves.

Properties of $\mathcal{VHB}$. We discuss two key properties of $\mathcal{VHB}$.

(1) *Soundness.* The $\mathcal{VHB}$ equivalence is sound for reporting correctness of local-state properties. In particular, if $t_1 \sim_{\mathcal{VHB}} t_2$, then every trace is guaranteed to visit the same local states in both executions. Thus, in order to report local-state-specific properties (e.g., absence of assertion violations), it is sound to explore a single representative from each class of the underlying partitioning. Global-state properties can be encoded as local properties by using a thread to monitor the global state. Due to this fact, many other recent works on DPOR focus on local-state properties only [Aronis et al. 2018; Chalupa et al. 2017; Huang 2015; Huang and Huang 2017].
(2) *Exponentially coarser than happens-before.* The $\mathcal{VHB}$ is always at least as coarse as the happens-before (or Mazurkiewicz) equivalence, i.e., if two traces are $\mathcal{HB}$-equivalent, then they are also $\mathcal{VHB}$-equivalent. This implies that the underlying $\mathcal{VHB}$ partitioning is never larger than the $\mathcal{HB}$ partitioning. In addition, we show that there exist programs for which the $\mathcal{VHB}$ partitioning is exponentially smaller, thereby getting a significant reduction in one of the two factors that affect the efficiency of DPOR algorithms. Interestingly, this reduction is achieved even if there are *no* concurrent writes in the program.

Value-centric DPOR. We develop an efficient DPOR algorithm that explores the $\mathcal{VHB}$ partitioning, called VC-DPOR. This algorithm is guaranteed to visit every class of the $\mathcal{VHB}$ partitioning, and for a constant number of threads, the time spent in each class is polynomial. Hence, VC-DPOR explores efficiently a value-centric partitioning without relying on NP oracles. For example, in the program of Figure 1, VC-DPOR explores only one trace.

Experimental results. Finally, we make a prototype implementation of VC-DPOR and evaluate it on various classes of concurrency benchmarks. We use our implementation to assess (i) the coarseness of the $\mathcal{VHB}$ partitioning in practice, and (ii) the efficiency of VC-DPOR to explore such partitionings. To this end, we compare these two metrics with existing state-of-the-art explicit DPOR algorithms, namely, the Source-DPOR [Abdulla et al. 2014], Optimal-DPOR [Abdulla et al. 2014], Optimal-DPOR with observers [Aronis et al. 2018], as well as DC-DPOR [Chalupa et al. 2017]. Our results show a significant reduction in the size of the partitioning compared to the partitionings explored by existing techniques, which also typically leads to smaller running times.

2 PRELIMINARIES

2.1 Concurrent Computation Model

In this section we define the model of concurrent programs and introduce general notation. We follow a standard exposition found in the literature (e.g., [Abdulla et al. 2014; Chalupa et al. 2017]). For simplicity of presentation we do not consider locks in our model. Later, we remark how locks can be handled naturally by our approach (see Remark 4).

General notation. Given a natural number $i \geq 1$, we denote by $[i]$ the set $\{1, 2, \ldots, i\}$. Given a map $f : X \to Y$, we let $\text{dom}(f) = X$ and $\text{img}(f) = Y$ denote the domain and image sets of f, respectively. We represent maps f as sets of tuples $\{(x, f(x))\}_x$. Given two maps f_1, f_2, we write $f_1 = f_2$ to denote that $\text{dom}(f_1) = \text{dom}(f_2)$ and for every $x \in \text{dom}(f_1)$ we have $f_1(x) = f_2(x)$, and we write $f_1 \neq f_2$ otherwise. A binary relation $\sim$ on a set X is an *equivalence* relation iff $\sim$ is reflexive, symmetric and transitive. Given an equivalence $\sim_E$ and some $x \in X$, we denote by $[x]_E$ the equivalence class of x under $\sim_E$, i.e., $[x]_E = \{y \in X : x \sim_E y\}$. The *quotient set* $X/E = \{[x]_E \mid x \in X\}$ of X under $\sim_E$ is the set of all equivalence classes of X under $\sim_E$.

Concurrent program. We consider a concurrent program $\mathcal{H} = \{p_i\}_{i=1}^k$ of k threads communicating over shared memory, where k is some arbitrary constant. For simplicity of presentation, we neglect dynamic thread creation. We distinguish p_1 as the *root thread* of $\mathcal{H}$, and refer to the remaining threads $p_2, \ldots, p_k$ as *leaf threads*. The shared memory consists of a finite set $\mathcal{G}$ of global variables, where each variable receives values from a finite value domain $\mathcal{D}$. Every thread executes instructions, which we call *events*, and are 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 $x \in \mathcal{G}$.
(3) A *local (invisible) event* is an event that does not access any global variable.

Although typically threads contain local events to guide the control-flow, such events are not relevant in our setting, and will thus be ignored. For simplicity of exposition, we consider that every thread is represented as an unrolled tree, which captures its unrolled control-flow, and every event is a node in this tree. In practice, each event is sufficiently identified by its thread identifier and an integer that counts how many preceding events of the same thread have been executed already. Given an event e, we denote by $p(e)$ the thread of e and by $\text{loc}(e)$ the unique global variable that e accesses. We denote by $\mathcal{E}$ the set of all events, by $\mathcal{W}$ the set of write events, and by $\mathcal{R}$ the set of read events of $\mathcal{H}$. Given a thread p, we denote by $\mathcal{E}_p$, $\mathcal{W}_p$ and $\mathcal{R}_p$ the set of events, read events and write events of p, respectively. In addition, we let $\mathcal{E}_{\neq p} = \bigcup_{p' \neq p} \mathcal{E}_{p'}$ and similarly for $\mathcal{W}_{\neq p}$ and $\mathcal{R}_{\neq p}$, i.e., $\mathcal{E}_{\neq p}$ denote the set of events of threads other than thread p, and similarly, for $\mathcal{W}_{\neq p}$ and $\mathcal{R}_{\neq p}$. Finally, given a set $X \subseteq \mathcal{E}$, we let $\mathcal{W}(X) = X \cap \mathcal{W}$ and $\mathcal{R}(X) = X \cap \mathcal{R}$ for the set of write and read events of X, respectively.

Concurrent program semantics. The semantics of $\mathcal{H}$ are defined by means of a transition system over a state space of global states $s = (\text{val}, \mathcal{L}_1, \ldots, \mathcal{L}_k)$, where $\text{val} : \mathcal{G} \to \mathcal{D}$ is a *value function* that maps every global variable to a value, and $\mathcal{L}_i$ is a *local state* of thread p_i, which contains the values of the local variables of each thread. The memory model considered here is sequentially consistent. Since the setting is standard, we omit here the formal setup and refer the reader to [Godefroid 2005] for details. As usual in stateless model checking, we focus our attention on state spaces $\mathcal{S}_{\mathcal{H}}$ that are acyclic (hence our focus is on bounded model checking).

Traces. A (concurrent) *trace* is a sequence of events $t = e_1, \dots, e_j$ that corresponds to a valid execution of $\mathcal{H}$. Given a trace t, we denote by $\mathcal{E}(t)$ the set of events that appear in t, and by $\mathcal{R}(t) = \mathcal{E}(t) \cap \mathcal{R}$ (resp., $\mathcal{W}(t) = \mathcal{E}(t) \cap \mathcal{W}$) the read (resp., write) events in t. We let $\mathsf{enabled}(t)$ denote the set of enabled events in the state reached after t is executed, and call t *maximal* if $\mathsf{enabled}(t) = \emptyset$. We write $\mathcal{T}_{\mathcal{H}}$ and $\mathcal{T}_{\mathcal{H}}^{\max}$ for the set of all traces and maximal traces, respectively, of $\mathcal{H}$. Given a set of events A, we denote by $t|A$ the *projection* of t on A, which is the unique subsequence of t that contains all events of $A \cap \mathcal{E}(t)$, and only those.

Observation, side and value functions. Given a trace t and a read event $r \in \mathcal{R}(t)$, the *observation* of r in t is the last write event w that appears before r in t such that $\mathsf{loc}(r) = \mathsf{loc}(w)$. The *observation function* of t is a function $O_t \colon \mathcal{R}(t) \to \mathcal{W}(t)$ such that $O_t(r)$ is the observation of r in t. The *side function* of t is a function $S_t \colon \mathcal{R}(t) \cap \mathcal{R}_{p_1} \to [2]$ such that $S_t(r) = 1$ if $p(O_t(r)) = p_1$ and $S_t(r) = 2$ otherwise. In other words, a side function is defined for the read events of the root thread, and assigns 1 (resp., 2) to each read event if it observes a local (resp., remote) write event in the trace [1]. The *value function* of t is a function $\mathsf{val}_t \colon \mathcal{E}(t) \to \mathcal{D}$ such that $\mathsf{val}_t(e)$ is the value of the global variable $\mathsf{loc}(e)$ after the prefix of t up to e has been executed. Note that since each thread is deterministic, this value is always unique and thus val_t is well-defined.

2.2 Problem and Complexity Parameters

The local-state reachability problem. The problem we address in this work is detecting erroneous local states of threads, e.g., whether a thread ever encounters an assertion violation. The underlying algorithmic problem is that of discovering every possible local state of every thread of $\mathcal{H}$, and checking whether a bug occurs. In stateless model checking, the focal object for this task is the trace, and algorithms solve the problem by exploring different maximal traces of the trace space $\mathcal{T}_{\mathcal{H}}^{\max}$. DPOR techniques use an equivalence E to partition the trace space into equivalence classes, and explore the partitioning $\mathcal{T}_{\mathcal{H}}^{\max}/E$ instead of the whole space $\mathcal{T}_{\mathcal{H}}^{\max}$.

Complexity parameters. Given an equivalence E over $\mathcal{T}_{\mathcal{H}}^{\max}$, the efficiency of an algorithm that explores the partitioning $\mathcal{T}_{\mathcal{H}}^{\max}/E$ is typically a product of two factors $O(\alpha \cdot \beta)$. The first factor α is the size of the partitioning itself, i.e., $\alpha = |\mathcal{T}_{\mathcal{H}}^{\max}/E|$, which is typically exponentially large. As we construct coarser equivalences E, α decreases. The second factor β captures the amortized time on each explored class, and can be either polynomial (i.e., efficient) or exponential. There is a tradeoff between α and β: typically, for coarser equivalences E the algorithms spend more time to explore each class, and hence α is decreased at the cost of increasing β. Hence, the challenge is to make α as small as possible without increasing β much.

This work. In this work, we introduce the value-happens before equivalence $\mathcal{VHB}$ and show that the $\mathcal{VHB}$-partitioning is efficiently explorable. For a constant number of threads, which is typically the case, $\beta = \mathsf{poly}(n)$, i.e., β is polynomial in the length of the longest trace in $\mathcal{T}_{\mathcal{H}}^{\max}$. Since, on the other hand, α is usually exponentially large in n, we will not focus on establishing the exact dependency of β on n. This helps to keep the exposition of the main message clear and focused.

Due to space restrictions, proofs and some experimental details appear in a technical report [Chatterjee et al. 2019].

[1] Although the definition of side functions might appear arbitrary, we rely on this definition later for computing the $\mathcal{VHB}$ abstraction.

2.3 Partial Orders

Here we introduce some useful notation around partial orders, which are the central objects of our algorithms in later sections.

Partial orders. Given a trace t and a set $X \subseteq \mathcal{E}(t)$, a *(strict) partial order* $P(X)$ over X is an irreflexive, antisymmetric and transitive relation over X (i.e., $<_{P(X)} \subseteq X \times X$). When X is clear from the context, we will simply write P for the partial order $P(X)$. Given two events $e_1, e_2 \in X$, we write $e_1 \leq_P e_2$ to denote that $e_1 < e_2$ or $e_1 = e_2$ Given two distinct events $e_1, e_2 \in X$, we say that e_1 and e_2 are *unordered* by P, denoted by $e_1 \parallel_P e_2$, if neither $e_1 <_P e_2$ nor $e_2 <_P e_1$. Given a set $Y \subseteq X$, we denote by $P|Y$ the *projection* of P on the set Y, i.e., $<_{P|Y} \subseteq Y \times Y$, and 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 $e_1 <_Q e_2$. We write $Q \sqsubset P$ to denote that $Q \sqsubseteq P$ and $P \not\sqsubseteq Q$. A *linearization* of P is a total order that refines P. Note that a trace t is a partial (and, in fact, total) order over the set $\mathcal{E}(t)$.

Conflicting events, width and Mazurkiewicz width. Two events e_1, e_2 are called *conflicting*, written $e_1 \bowtie e_2$, if they access the same global variable and at least one writes to the variable. Let P be a partial order over a set X. The *width* $\mathsf{width}(P)$ of P is the length of its longest antichain, i.e., it is the smallest integer i such that for every set $Y \subseteq X$ of size $i+1$ $e_1, e_2 \in Y$ such that $e_1 \not\parallel_P e_2$. A set $Y \subseteq X$ is called *pairwise conflicting* if for every pair of distinct events $e_1, e_2 \in Y$, we have that $e_1 \bowtie e_2$. We define the *Mazurkiewicz width* $\mathsf{Mwidth}(P)$ of P as the smallest integer i such that for every pairwise conflicting set $Y \subseteq X$ of size $i+1$ there exists a pair $e_1, e_2 \in Y$ such that $e_1 \not\parallel_P e_2$. Intuitively, $\mathsf{Mwidth}(P)$ is similar to $\mathsf{width}(P)$, with the difference that, in the first case, we focus on events that are conflicting as opposed to any events.

The thread order TO. The *thread order* TO of $\mathcal{H}$ is a partial order $<_{\mathsf{TO}} \subseteq \mathcal{E} \times \mathcal{E}$ that defines a fixed order between pairs of events of the same thread. For every trace $t \in \mathcal{T}_{\mathcal{H}}$, we have that $t \sqsubseteq \mathsf{TO}|\mathcal{E}(t)$. Every partial order P used in this work respects the thread order.

Visible, maximal and minimal writes. Consider a partial order P over a set X. Given a read event $r \in \mathcal{R}(X)$ we define the set of *visible writes* of r as

$$\begin{aligned}\mathsf{VisibleW}_P(r) = \{ & w \in \mathcal{W}(X) \colon\ r \bowtie w \text{ and } r \not<_P w \text{ and for each } w' \in \mathcal{W}(X) \\ & \text{s.t. } r \bowtie w', \text{ if } w <_P w' \text{ then } w' \not<_P r\}\end{aligned}$$

In words, $\mathsf{VisibleW}_P(r)$ contains the write events w that conflict with r and are not "hidden" to r by P, i.e., there exist linearizations t of P such that $O_t(r) = w$ (note that here t is not necessarily an actual trace of $\mathcal{H}$). The set of *minimal writes* $\mathsf{MinW}_P(r)$ (resp., *maximal writes* $\mathsf{MaxW}_P(r)$) of r contains the write events that are minimal (resp., maximal) elements in $P|\mathsf{VisibleW}_P(r)$.

The happens-before partial order. A trace t induces a *happens-before* partial order $\rightarrow_t \subseteq \mathcal{E}(t) \times \mathcal{E}(t)$, which is the smallest transitive relation on $\mathcal{E}(t)$ such that (i) $\rightarrow_t \sqsubseteq \mathsf{TO}|\mathcal{E}(t)$ and (ii) $e_1 \rightarrow_t e_2$ if $e_1 <_t e_2$ and $e_1 \bowtie e_2$.

The causally-happens-before partial order. A trace t induces a *causally-happens-before* partial order $\mapsto_t \subseteq \mathcal{E}(t) \times \mathcal{E}(t)$, which is the smallest transitive relation on $\mathcal{E}(t)$ such that (i) $\mapsto_t \sqsubseteq \mathsf{TO}|\mathcal{E}(t)$ and (ii) for every read event $r \in \mathcal{R}(t)$, we have $O_t(r) \mapsto_t r$. In words, $\mapsto$ captures the flow of write events into read events, and is closed under composition with the thread order. Intuitively, for an event e, the set of events e' that causally-happen-before e are the events that need to be present so

that e is enabled. Note that $\rightarrow_t \sqsubseteq \mapsto_t$, i.e., the happens-before partial order refines the causally-happens-before partial order.

We refer to Figure 2 for an illustration of the $\rightarrow_t$ and $\mapsto_t$ partial orders.

	τ_1	τ_2	τ_3
1	$w(x,1)$		
2			$w(x,1)$
3		$w(y,1)$	
4		$r(y,1)$	
5		$w(x,1)$	
6			$w(y,2)$
7	$w(y,1)$		
8	$r(x,1)$		

(a) A trace t of three threads.

$$\begin{aligned}\rightarrow_t = &\mathsf{TO}|\mathcal{E}(t) \cup && \text{(thread order)}\\ &\{e_1 \rightarrow_t e_2 \rightarrow_t e_5 \rightarrow_t e_8\} \cup && \text{(on } x)\\ &\{e_3 \rightarrow_t e_4 \rightarrow_t e_6 \rightarrow_t e_7\} && \text{(on } y)\end{aligned}$$

$$\begin{aligned}\mapsto_t = &\mathsf{TO}|\mathcal{E}(t) \cup && \text{(thread order)}\\ &\{e_5 \mapsto_t e_8\} && \text{(on } x)\end{aligned}$$

(b) The happens-before $\rightarrow_t$ and causally-happens-before $\mapsto_t$ partial orders.

Fig. 2. A trace (a) and the induced happens-before and causally-happens-before partial orders (b). We use the notation e_i to refer to the i-th event of t.

3 THE VALUE-HAPPENS-BEFORE EQUIVALENCE

In this section we introduce our new equivalence between traces, called *value-happens-before*, and prove some of its properties. We start with the happens-before equivalence, which has been used by DPOR algorithms in the literature.

The happens-before equivalence. Two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{H}}$ are called *happens-before-equivalent* (commonly referred to as *Mazurkiewicz equivalent*), written $t_1 \sim_{\mathcal{HB}} t_2$, if the following hold.

(1) $\mathcal{E}(t_1) = \mathcal{E}(t_2)$, i.e., they consist of the same set of events.
(2) $\rightarrow_{t_1} = \rightarrow_{t_2}$, i.e., their happens-before partial orders are equal.

The value-happens-before equivalence. Two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{H}}$ are called *value-happens-before-equivalent*, written $t_1 \sim_{\mathcal{VHB}} t_2$, if the following hold.

(1) $\mathcal{E}(t_1) = \mathcal{E}(t_2)$, $\mathsf{val}_{t_1} = \mathsf{val}_{t_2}$ and $S_{t_1} = S_{t_2}$, i.e., they consist of the same set of events, and their value functions and side functions are equal.
(2) $\mapsto_{t_1}|\mathcal{R} = \mapsto_{t_2}|\mathcal{R}$, i.e., $\mapsto_{t_i}$ agree on the read events.
(3) $\rightarrow_{t_1}|\mathcal{E}_{\neq p_1} = \rightarrow_{t_2}|\mathcal{E}_{\neq p_1}$, i.e., $\rightarrow_{t_i}$ agree on the events of the leaf threads.

Remark 1 (Soundness). *Since every thread of $\mathcal{H}$ is deterministic, for any two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{H}}$ such that $\mathcal{E}(t_1) = \mathcal{E}(t_2)$ and $\mathsf{val}_{t_1} = \mathsf{val}_{t_2}$, the local states of each thread after executing t_1 and t_2 agree. It follows that any algorithm that explores every class of $\mathcal{T}_{\mathcal{H}}^{\max}/\mathcal{VHB}$ discovers every local state of every thread, and thus $\mathcal{VHB}$ is a sound equivalence for local-state reachability.*

Exponential coarseness. Here we provide two toy examples which illustrate different cases where $\mathcal{VHB}$ can be exponentially coarser than $\mathcal{HB}$, i.e., $\mathcal{T}_{\mathcal{H}}/\mathcal{HB}$ can have exponentially more classes than $\mathcal{T}_{\mathcal{H}}/\mathcal{VHB}$.

Many operations on one variable. First, consider the program shown in Figure 3a which consists of two threads p_1 and p_2, with p_1 being the root thread. This program has a single global variable x, and the threads perform operations on x repeatedly. We assume a salient write event $w(x,0)$ that writes

Thread p_1 :	Thread p_2 :
1. $w(x, 0)$	1. $r(x)$
2. $w(x, 0)$	2. $r(x)$
... ...	... ...
n. $w(x, 0)$	n. $r(x)$

(a) Many operations on one variable.

Thread p_1 :	Thread p_2 :
1. $w(x_1, 0)$	1. $r(x_1)$
2. $w(x_1, 0)$	2. $r(x_2)$
... ...	... ...
$2 \cdot n - 1$. $w(x_n, 0)$	n. $r(x_n)$
$2 \cdot n$. $w(x_n, 0)$	

(b) Few operations on many variables.

Fig. 3. Toy programs where $\mathcal{VHB}$ is exponentially coarser than $\mathcal{HB}$.

the initial value of x. Consider any two traces t_1, t_2 that consist of the $i \geq 0$ first $w(x)$ events of p_1 and $j \geq 0$ first $r(x)$ events of p_2 (hence $\mathcal{E}(t_1) = \mathcal{E}(t_2)$). Since each $w(x)$ writes the same value, we have $\mathsf{val}_{t_1}(r) = \mathsf{val}_{t_2}(r)$ for every read event r in p_2. Moreover, since the root thread p_1 has no read events, we trivially have $S_{t_1} = S_{t_2}$. Since all read events are on thread p_2, we have $\mapsto_{t_1}|\mathcal{R} = \mapsto_{t_2}|\mathcal{R} = \mathsf{TO}|\mathcal{R}(t_1)$. Finally, since we only have one leaf thread, $\rightarrow_{t_1}|\mathcal{E}_{\neq p_1} = \rightarrow_{t_2}|\mathcal{E}_{\neq p_1} = \mathsf{TO}|\mathcal{E}_{\neq p_1}(t_1)$. We conclude that $t_1 \sim_{\mathcal{VHB}} t_2$, and thus given $i \geq 0$ and $j \geq 0$ there exists a single class of $\sim_{\mathcal{VHB}}$ that contains the first i and first j events of p_1 and p_2, respectively. Thus $|\mathcal{T}_{\mathcal{H}}/\mathcal{VHB}| = O(n^2)$. On the other hand, given the first $i \geq 0$ and $j \geq 0$ events of threads p_1 and p_2, respectively, there exist $\frac{(i+j)!}{i! \cdot j!} = \binom{i+j}{i}$ different ways to order them without violating the thread order. Observe that every such reordering induces a different happens-before relation. Using Stirling's approximation, we obtain

$$|\mathcal{T}_{\mathcal{H}}/\mathcal{HB}| \geq \frac{(2 \cdot n)!}{(n!)^2} \simeq \frac{\sqrt{2 \cdot \pi \cdot 2 \cdot n} \cdot (2 \cdot n/e)^{2 \cdot n}}{\left(\sqrt{2 \cdot \pi \cdot n} \cdot (n/e)^n\right)^2} = \Omega\left(\frac{4^n}{\sqrt{n}}\right)$$

Few operations on many variables. Now consider the example program shown in Figure 3b which consists of two threads p_1 and p_2, with p_1 being the root thread. We assume a salient write event $w(x_i, 0)$ that writes the initial value of x_i. Consider any two traces t_1, t_2 that consist of the $i \geq 0$ first $w(x)$ events of p_1 and $j \geq 0$ first $r(x)$ events of p_2 (hence $\mathcal{E}(t_1) = \mathcal{E}(t_2)$). Since each $w(x_i, 0)$ writes the same value, we have $\mathsf{val}_{t_1}(r) = \mathsf{val}_{t_2}(r)$ for every read event r in p_2. Moreover, since the root thread p_1 has no read events, we trivially have $S_{t_1} = S_{t_2}$. Since all read events are on thread p_2, we have $\mapsto_{t_1}|\mathcal{R} = \mapsto_{t_2}|\mathcal{R} = \mathsf{TO}|\mathcal{R}(t_1)$. Finally, since we only have one leaf thread, $\rightarrow_{t_1}|\mathcal{E}_{\neq p_1} = \rightarrow_{t_2}|\mathcal{E}_{\neq p_1} = \mathsf{TO}|\mathcal{E}_{\neq p_1}(t_1)$. We conclude that $t_1 \sim_{\mathcal{VHB}} t_2$, and thus given $i \geq 0$ and $j \geq 0$ there exists a single class of $\sim_{\mathcal{VHB}}$ that contains the first i and first j events of p_1 and p_2, respectively. Thus $|\mathcal{T}_{\mathcal{H}}/\mathcal{VHB}| = O(n^2)$. On the other hand, given the first i read events of p_2 and $2 \cdot i$ write events of p_1, there exist at least 2^i different observation functions that map each read event r to one of the two write events that r observes. Hence $|\mathcal{T}_{\mathcal{H}}/\mathcal{HB}| = \Omega(2^n)$.

THEOREM 3.1. *$\mathcal{VHB}$ is sound for local-state reachability. Also, $\mathcal{VHB}$ is at least as coarse as $\mathcal{HB}$, and there exist programs where $\mathcal{VHB}$ is exponentially coarser.*

4 CLOSED ANNOTATED PARTIAL ORDERS

In this section we develop the core algorithmic concepts that will be used in the enumerative exploration of the $\mathcal{VHB}$. We introduce *annotated partial orders*, which are traditional partial orders over events, with additional constraints. We formulate the question of the realizability of an annotated partial order $\mathcal{P}$, which asks for a witness trace t that linearizes $\mathcal{P}$ and satisfies the

constraints. We develop the notion of *closure* of annotated partial orders, and show that (i) an annotated partial order is realizable if and only if its closure exists, and (ii) deciding whether the closure exists can be done efficiently. This leads to an efficient procedure for deciding realizability.

4.1 Annotated Partial Orders

Here we introduce the notion of annotated partial orders, which is a central concept of our work. We build some definitions and notation, and provide some intuition around them.

Annotated Partial Orders. An *annotated partial order* is a tuple $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ where the following hold.

(1) X_1, X_2 are sets of events such that $X_1 \cap X_2 = \emptyset$.
(2) P is a partial order over the set $X = X_1 \cup X_2$.
(3) $\mathsf{val}\colon X \to \mathcal{D}$ is a value function.
(4) $S\colon \mathcal{R}(X_1) \to [2]$ is a side function.
(5) $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$ is a good-writes function such that $w \in \mathsf{GoodW}(r)$ only if $r \bowtie w$ and $\mathsf{val}(r) = \mathsf{val}(w)$ and, if $r \in X_1$ then $w \in X_{S(r)}$.
(6) $\mathsf{width}(P|X_1) = \mathsf{Mwidth}(P|X_2) = 1$.

We let the bad-writes function be $\mathsf{BadW}(r) = \{w \in \mathcal{W}(X) \setminus \mathsf{GoodW}(r) : r \bowtie w\}$. We call $\mathcal{P}$ *consistent* if for every thread p, we have that $\tau_p = \mathsf{TO}|(X \cap \mathcal{E}_p)$ is a local trace of thread p that occurs if every event e of τ_p reads/writes the value $\mathsf{val}(e)$. Hereinafter we only consider consistent annotated partial orders.

The realizability problem for annotated partial orders. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$. A trace t is a *linearization* of $\mathcal{P}$ if (i) $t \sqsubseteq P$ and (ii) for every read event $r \in \mathcal{R}(X_1 \cup X_2)$ we have that $O_t(r) \in \mathsf{GoodW}(r)$. In words, t must be a linearization of the partial order P with the additional constraint that the observation function of t must agree with the good-writes function GoodW of $\mathcal{P}$. We call $\mathcal{P}$ *realizable* if it has a linearization. The associated realizability problem takes as input an annotated partial order $\mathcal{P}$ and asks whether $\mathcal{P}$ is realizable.

Remark 2 (Realizability to valid traces.). *If t is a linearization of some consistent annotated partial order $\mathcal{P}$ then t is a valid (i.e., actual) trace of $\mathcal{H}$. This holds because of the following observations.*

(1) Since t is a linearization of $\mathcal{P}$, we have $O_t(r) \in \mathsf{GoodW}(r)$ for every read event $r \in \mathcal{R}(t)$.
(2) Due to the previous item and the consistency of $\mathcal{P}$, for every thread p we have that $\tau_p = \mathsf{TO}|(X \cap \mathcal{E}_p)$ is a valid local trace of p.

Intuition. An annotated partial order $\mathcal{P}$ contains a partial order P over a set $X = X_1 \cup X_2$ of events and the value of each event of X. Intuitively, the consistency of $\mathcal{P}$ states that we obtain the set of events X if we execute each thread and force every read event in this execution to observe the value of a write event according to the good-writes function. In the next section, our VC-DPOR algorithm uses annotated partial orders to represent different classes of the $\mathcal{VHB}$ equivalence in order to guide the trace-space exploration. The set X_1 (resp., X_2) will contain the events of the root thread (resp., leaf threads). We will see that if VC-DPOR constructs two annotated partial orders $\mathcal{P}$ and $\mathcal{Q}$ during the exploration, then any two linearizations t_1 and t_2 of $\mathcal{P}$ and $\mathcal{Q}$, respectively, will satisfy that $t_1 \not\sim_{\mathcal{VHB}} t_2$, and hence $\mathcal{P}$ and $\mathcal{Q}$ represent different classes of the $\mathcal{VHB}$ partitioning.

Closed annotated partial orders. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ and let $X = X_1 \cup X_2$. We say that $\mathcal{P}$ is *closed* if the following conditions hold for every read event $r \in \mathcal{R}(X)$.

(1) There exists a write event $w \in \mathsf{GoodW}(r) \cap \mathsf{MinW}_P(r)$ such that $w <_P r$.
(2) $\mathsf{MaxW}_P(r) \cap \mathsf{GoodW}(r) \neq \emptyset$.
(3) For every write event $w' \in \mathsf{BadW}(r) \cap \mathsf{MinW}_P(r)$ such that $w' <_P r$ there exists a write event $w \in \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_P(r)$ such that $w' <_P w$.

Our motivation behind this definition becomes clear from the following lemma, which states that closed annotated partial orders are realizable.

Lemma 4.1. *If $\mathcal{P}$ is closed then it is realizable and a witness can be constructed in $O(\mathsf{poly}(n))$ time.*

In particular, the witness trace of $\mathcal{P}$ is constructed by the following process.

(1) Create a partial order Q as follows.
   (a) For every pair of events e_1, e_2 with $e_1 <_P e_2$, we have $e_1 <_Q e_2$.
   (b) For every pair of events e_1, e_2 with $e_i \in X_i$ for each $i \in [2]$, if $e_2 \not<_P e_1$ then $e_1 <_Q e_2$.
(2) Create t by linearizing Q arbitrarily.

The above construction is guaranteed to produce a valid witness trace for $\mathcal{P}$. The consistency of annotated partial orders guarantees that t is a valid trace of the concurrent program $\mathcal{H}$ (see Remark 2). We provide an illustration of this construction later in Figure 5.

We now introduce the notion of *closures*. Intuitively, the closure of an annotated partial order $\mathcal{P}$ strengthens $\mathcal{P}$ by introducing the smallest set of event orderings such that the resulting annotated partial order $\mathcal{Q}$ is closed. The intuition behind the closure is the following: whenever a rule forces some ordering, any trace that witnesses the realizability of $\mathcal{P}$ also linearizes $\mathcal{Q}$. In some cases this operation results to cyclic orderings, and thus the closure does not exist. We also show that obtaining the closure or deciding that it does not exist can be done in polynomial time. Thus, in combination with Lemma 4.1, we obtain an efficient algorithm for deciding whether $\mathcal{P}$ is realizable, by deciding whether it has a closure.

Closure of annotated partial orders. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$. We say that an annotated partial order $\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ is a *closure* of $\mathcal{P}$ if (i) $Q \sqsubseteq P$, (ii) $\mathcal{Q}$ is closed, and (iii) for any partial order K with $Q \sqsubset K \sqsubseteq P$, we have that the annotated partial order $(X_1, X_2, K, \mathsf{val}, S, \mathsf{GoodW})$ is not closed. As the following lemma states, $\mathcal{P}$ can have at most one closure.

Lemma 4.2. *There exists at most one weakest partial order Q such that $Q \sqsubseteq P$ and $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ is closed.*

Feasible annotated partial orders. In light of Lemma 4.2, we define the *closure* of $\mathcal{P}$ as the unique annotated partial order $\mathcal{Q}$ that is a closure of $\mathcal{P}$, if such $\mathcal{Q}$ exists, and $\perp$ otherwise. We call $\mathcal{P}$ *feasible* if its closure is not $\perp$. We have the following lemma.

Lemma 4.3. *$\mathcal{P}$ is realizable if and only if it is feasible.*

Intuitively, Lemma 4.3 states that the closure rules give the weakest strengthening of $\mathcal{P}$ that is met by any linearization of $\mathcal{P}$. If that strengthening can be made (i.e., $\mathcal{P}$ is feasible), then $\mathcal{P}$ has a linearization. Hence, to decide whether $\mathcal{P}$ is realizable, it suffices to decide whether it is feasible, by computing its closure. In the next section we show that this computation can be done efficiently.

4.2 Computing the Closure

We now turn our attention to computing the closure of annotated partial orders, which will provide us with a way of solving the realizability problem.

Algorithm Closure**.** Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ and let $X = X_1 \cup X_2$. The algorithm Closure either computes the closure of $\mathcal{P}$, or concludes that $\mathcal{P}$ is not feasible, and returns $\perp$. Intuitively, the algorithm maintains a partial order Q, initially identical to P. The algorithm iterates over every read event r and tests whether r violates Item 1, Item 2 or Item 3 of the definition of closed annotated partial orders. When it discovers that r violates one such closure rule, Closure calls one of the *closure methods* Rule1(r), Rule2(r), Rule3(r), for violation of Item 1, Item 2 and Item 3 of the definition, respectively. In turn, each of these methods inserts a new ordering $e_1 \rightarrow e_2$ in Q, with the guarantee that if $\mathcal{P}$ has a closure $\mathcal{F} = (X_1, X_2, F, \mathsf{val}, S, \mathsf{GoodW})$, then $e_1 <_F e_2$. Hence, $e_1 \rightarrow e_2$ is a *necessary ordering* in the closure of $\mathcal{P}$. Finally, when the algorithm discovers that all closure rules are satisfied by every read event in Q, it returns the annotated partial order $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$, which, due to Lemma 4.2, is guaranteed to be the closure of $\mathcal{P}$. We refer to Algorithm 1 for a formal description.

We now provide some intuition behind each of the closure methods. Given an event $e \in X$, we let $\mathcal{I}_{\mathcal{P}}(e) = i$ such that $e \in X_i$. Given two events $e_1, e_2 \in X$, we say that e_2 is *local* to e_1 if $\mathcal{I}_{\mathcal{P}}(e_1) = \mathcal{I}_{\mathcal{P}}(e_2)$, i.e., e_1 and e_2 belong to the same set X_i. If e_2 is not local to e_1, then it is *remote* to e_1.

(1) Rule1(r). This rule is called when Item 1 of closure is violated, i.e., there exists no write event $w \in \mathsf{GoodW}(r) \cap \mathsf{MinW}_Q(r)$ such that $w <_Q r$. Observe that in this case there is no write event that is (i) local to r, (ii) good for r and (iii) visible to r. To make r respect this rule, the algorithm finds the first write event w that is (i) good for r and (ii) visible to r, and orders $w \rightarrow r$ in Q. See Figure 4a provides an illustration.
(2) Rule2(r). This rule is violated when $\mathsf{MaxW}_Q(r) \cap \mathsf{GoodW}(r) = \emptyset$, i.e., every maximal write event is bad for r. To make r respect this rule, the algorithm finds the unique maximal write event w that is remote to r and orders $r \rightarrow w$ in Q. Rule2(r) is called only if r does not violate Item 1 of closure, which guarantees that w exists. Figure 4b provides an illustration.
(3) Rule3(r). This rule is violated when there exists a write event $\overline{w} \in \mathsf{BadW}(r) \cap \mathsf{MinW}_Q(r)$ such that (i) $\overline{w} <_Q r$, and (ii) there exists no write event $w' \in \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_Q(r)$ such that $\overline{w} <_Q w'$. To make r respect this rule, the algorithm determines a maximal write event w that is (i) remote to $\overline{w}$ and (ii) a good write for r, and orders $\overline{w} \rightarrow w$ in Q. Rule3(r) is called only if r does not violate either Item 1 or Item 2 of closure, which guarantees that w exists. Figure 4c provides an illustration, depending on whether $\overline{w}$ is local or remote to r.

We have the following lemma regarding the correctness and complexity of Closure.

Lemma 4.4. Closure *correctly computes the closure of* $\mathcal{P}$ *and requires* $O(\mathsf{poly}(n))$ *time.*

4.3 Realizing Annotated Partial Orders

Finally, we address the question of realizability of annotated partial orders. Lemma 4.3 implies that in order to decide whether an annotated partial order is realizable, it suffices to compute its closure,

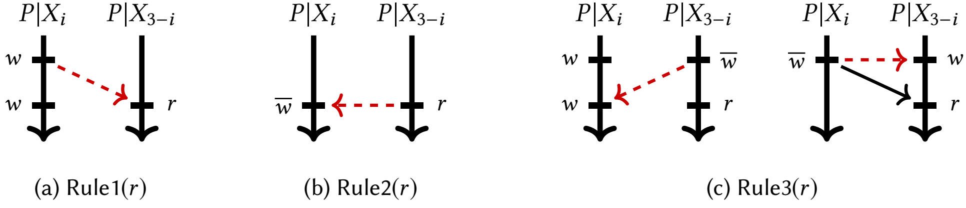


Fig. 4. Illustration of the three closure operations Rule1(r) (a), Rule2(r) (b) and Rule3(r) (c). We follow the convention that barred and unbarred write events ($\overline{w}$ and w) are bad writes and good writes for r, respectively. In each case, the dashed edge shows the new order introduced by the algorithm in Q.

Algorithm 1: Closure($\mathcal{P}$)

Input: An annotated partial order $\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW})$.
Output: The closure of $\mathcal{P}$ if it exists, else $\perp$.

```
1  Q ← P                                      // We will strengthen Q during the closure computation
2  Flag ← True
3  while Flag do
4  |  Flag ← False
5  |  foreach r ∈ R(X1 ∪ X2) do                                          // Iterate over the reads
6  |  |  if r violates Item 1 of closure then
7  |  |  |  Call Rule1(r)                                     // Strengthen Q to remove violation
8  |  |  |  Flag ← True                          // Repeat as new violations might have appeared
9  |  |  if r violates Item 2 of closure then
10 |  |  |  Call Rule2(r)                                     // Strengthen Q to remove violation
11 |  |  |  Flag ← True                          // Repeat as new violations might have appeared
12 |  |  if r violates Item 3 of closure then
13 |  |  |  Call Rule3(r)                                     // Strengthen Q to remove violation
14 |  |  |  Flag ← True                          // Repeat as new violations might have appeared
15 |  end
16 end
17 return (X1, X2, Q, val, S, GoodW)                                         // The closure of P
```

Algorithm 2: Rule1(r)

```
1  Y ← GoodW(r) ∩ VisibleW_Q(r)
2  if Y = ∅ then return ⊥
3  w ← min_Q(Y)                            // Since Rule 1 is violated, min_Q(Y) is unique
4  Insert w → r in Q
```

Algorithm 3: Rule2(r)

```
1  w ← the unique event in MaxW_Q(r) ∩ X_{3−I_P(r)}         // w exists since Item 1 of closure holds
2  Insert r → w in Q
```

and Lemma 4.4 states that the closure can be computed efficiently. Together, these two lemmas yield a simple algorithm for solving the realizability problem.

Algorithm 4: Rule3(r)

1 $\overline{w} \leftarrow$ the unique event in $\mathsf{MinW}_Q(r) \cap \mathsf{BadW}(r)$ // $\overline{w}$ exists since Items 1 and 2 of closure hold
2 $w \leftarrow$ the unique event in $\mathsf{MaxW}_Q(r) \cap X_{3-\mathcal{I}_{\mathcal{P}}(\overline{w})}$ // w exists since Items 1 and 2 of closure hold
3 Insert $\overline{w} \rightarrow w$ in Q

Algorithm Realize**.** We describe a simple algorithm Realize that decides whether an annotated partial order $\mathcal{P}$ is realizable. The algorithms runs in two steps.

(1) Use Lemma 4.4 to compute the closure of $\mathcal{P}$. If the closure is $\perp$, report that $\mathcal{P}$ is not realizable. Otherwise, the closure is an annotated partial order $\mathcal{Q}$.
(2) Use Lemma 4.1 to obtain a witness trace t that linearizes $\mathcal{Q}$. Report that $\mathcal{P}$ is linearizable, and t is the witness trace.

We conclude the results of this section with the following theorem.

THEOREM 4.5. *Let $\mathcal{P}$ be an annotated partial order of n events. Deciding whether $\mathcal{P}$ is realizable requires $O(\mathrm{poly}(n))$ time. If $\mathcal{P}$ is realizable, a witness trace can be produced in $O(\mathrm{poly}(n))$ time.*

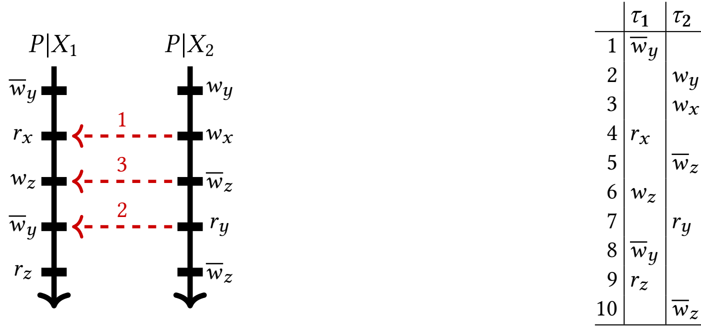
Example on the realizability of annotated partial orders. We illustrate Realize on a simple example in Figure 5 with an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$, which we assume to be consistent. We have a concurrent program $\mathcal{H}$ of two threads. To represent $\mathcal{P}$, we make the following conventions. We have three global variables x, y, z, and a unique read event per variable. Event subscripts denote the variable accessed by the corresponding event. For each variable, we have a unique read event, and barred and unbarred events denote the good and bad write events, respectively, for that read event. Since we have specified the good-writes for each read event, the value function val is not important for this example. Note also that $S(r_x) = 2$ (resp., $S(r_z) = 1$) since the good writes of r_x (resp., r_z) are remote (resp., local) to the read event. The partial order P of $\mathcal{P}$ consists of the thread orders of each thread, shown in solid lines in Figure 5a. The dashed edges of Figure 5a show the strengthening of P performed by the algorithm Closure (Algorithm 1). The numbers above the dashed edges denote both the order in which these orderings are added and the closure rule that is responsible for the corresponding ordering. In particular, algorithm Closure performs the following steps.

(1) Initially there are no dashed edges, and r_x violates Item 1 of closure, as there is no good write event for r_x that is ordered before r_x. Rule1 inserts an ordering $w_x \rightarrow r_x$ (dashed edge 1).
(2) After the previous step, r_y violates Item 2 of closure, as at this point, r_y has only one maximal write event $\overline{w}_y$, which is bad for r_y. Rule2 inserts an ordering $r_y \rightarrow \overline{w}_y$ (dashed edge 2).
(3) After the previous step, r_z violates Item 3 of closure, as at this point, r_z has a bad minimal write event $\overline{w}_z$ that is ordered before r_z but not before any good write event. Rule3 inserts an ordering $\overline{w}_z \rightarrow w_z$ (dashed edge 3).

At this point no closure rule is violated, and Closure returns the closure $\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ of $\mathcal{P}$ where P has been strengthened to Q with the dashed edges. Observe that Q has Mazurkiewicz width 2 (and not 1), as there still exist pairs of conflicting events that are unordered, both on variable y and variable z. For example, there exist two write events on variable y that are unordered, and hence there exist some linearizations that are "bad" in the sense that the read event r_y does not observe the good write event w_y. Nevertheless, Lemma 4.1 guarantees

that the corresponding annotated partial order is linearizable to a valid trace, which is shown in Figure 5b We make two final remarks for this example.

(1) Not every linearization of Q produces a valid witness trace for the realizability of $\mathcal{Q}$, as some linearizations violate the additional constraints that every read event must observe a write event that is good for the read event. Hence, the challenge is to find a correct witness.
(2) $\mathcal{Q}$ has more than one witness of realizability. Figure 5b shows one such witness t, as constructed by Lemma 4.1. It is easy to verify that t is a valid witness. Due to Remark 2, the consistency of $\mathcal{P}$ guarantees that t is a valid trace of the program $\mathcal{H}$.



	τ_1	τ_2
1	$\overline{w}_y$	
2		w_y
3		w_x
4	r_x	
5		$\overline{w}_z$
6	w_z	
7		r_y
8	$\overline{w}_y$	
9	r_z	
10		$\overline{w}_z$

(a) An annotated partial order $\mathcal{P}$ and its closure (dashed edges). (b) A witness trace that linearizes $\mathcal{P}$.

Fig. 5. Figure 5a shows an annotated partial order $\mathcal{P}$ on a concurrent program of two threads. Subscripts denote the variable accessed by each event. For each variable, we have a unique read event, and barred and unbarred events denote the good and bad write events, respectively, for that read event. Dashed edges are added by Closure (Algorithm 1) during closure. Figure 5b shows a witness trace that linearizes $\mathcal{P}$.

5 VALUE-CENTRIC DYNAMIC PARTIAL ORDER REDUCTION

We now present our algorithm VC-DPOR for exploring the partitioning $\mathcal{T}_{\mathcal{H}}^{\max}/\mathcal{VHB}$. Intuitively, the algorithm manipulates annotated partial orders of the form $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$, where $X_1 \subseteq \mathcal{E}_{p_1}$ and $X_2 \subseteq \mathcal{E}_{\neq p_1}$, i.e., X_1 (resp., X_2) contains events of the root thread (resp., leaf threads). We first introduce some useful concepts and then proceed with the main algorithm.

Trace extensions and inevitable sets. Given a trace t, an *extension* of t is a trace t' such that t is a prefix of t'. We say that t' is a *maximal extension* of t if t' is an extension of t and t' is maximal. A set of events X is *inevitable* for t if for every maximal extension t' of t we have $X \in \mathcal{E}(t')$. A *write extension* of t, denoted by $\mathsf{WExtend}(t)$, is any arbitrary largest extension t' of t such that $\mathcal{E}(t') \setminus \mathcal{E}(t) \subseteq \mathcal{W}$. In words, we obtain each t' by extending t arbitrarily until (but not included) the next read event of each thread. Note that for every such write extension t' of t, for every thread p, the local trace $t'|\mathcal{E}(p)$ is unique, and the set $\mathcal{E}(t')$ is inevitable for t. Let $\mathcal{P}$ be a closed annotated partial order over a set X. A set of events Y is *inevitable* for $\mathcal{P}$ if for every linearization t of $\mathcal{P}$ and every maximal extension t' of t, we have that $Y \subseteq \mathcal{E}(t')$.

Leaf refinement and minimal annotated partial orders. Consider two partial orders P, Q over a set X. We say that Q *leaf-refines* P, denoted by $Q \preccurlyeq P$ if for every pair of events $e_1, e_2 \in X \cap \mathcal{E}_{\neq p_1}$, if $e_1 \bowtie e_2$ and $e_1 <_P e_2$ then $e_1 <_Q e_2$. In words, Q leaf-refines P if Q agrees with P on the order of every pair of conflicting events that belong to leaf threads. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$. We call $\mathcal{P}$ *minimal* if for every closed annotated partial order

$\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$, if $Q \preccurlyeq P$ then $Q \sqsubseteq P$. Intuitively, the minimality of $\mathcal{P}$ guarantees that P is the weakest partial order among all partial orders Q that

(1) agree with P on the order of conflicting pairs of events that belong to leaf threads, and
(2) make the resulting annotated partial order $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ closed.

Hence P does not contain any unnecessary orderings, given these two constraints. Observe that if $\mathcal{P}$ is minimal and $\mathcal{K}$ is the closure of $\mathcal{P}$ then $\mathcal{K}$ is also minimal. Afterwards, our algorithm VC-DPOR will use minimal annotated partial orders to represent different classes of the $\mathcal{VHB}$ partitioning.

Algorithm $\mathsf{Extend}(\mathcal{P}, X', \mathsf{val}', S', \mathsf{GoodW}')$**.** Let $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ be a minimal, closed annotated partial order, and $X = X_1 \cup X_2$. Consider

(1) a set X' with (i) $X' \setminus X \subseteq \mathcal{W}$ or $|X' \setminus X| = 1$ and (ii) X' is inevitable for $\mathcal{P}$,
(2) a value function val' over X' such that $\mathsf{val} \subseteq \mathsf{val}'$,
(3) a side function S' over X' such that $S \subseteq S'$, and
(4) a good-writes set GoodW' over X' such that $\mathsf{GoodW} \subseteq \mathsf{GoodW}'$.

We rely on an algorithm called Extend that constructs an *extension* of $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ to X', val', S' and GoodW' as a set of minimal closed annotated partial orders $\{\mathcal{K}_i = (X'_1, X'_2, K_i, \mathsf{val}', S', \mathsf{GoodW}')\}_i$, where $X'_1 \cup X'_2 = X'$. Intuitively, if t is a linearization of $\mathcal{P}$, then for every extension t' of t such that $\mathcal{E}(t') = X'$, $\mathsf{val}_{t'} = \mathsf{val}'$ and $S_{t'} = S'$, there exists some $\mathcal{K}_i$ that linearizes to t'. In VC-DPOR, we will use Extend to extend annotated partial orders with new events.

We describe Extend for the special case where $|X' \setminus X| = 1$. When $|X' \setminus X| = q > 1$, Extend calls itself recursively for every annotated partial order of its output set on a sequence of sets $Y_1, \dots, Y_q$ where $Y_q = X'$, $Y_0 = X$ and $|Y_{i+1} \setminus Y_i| = 1$. Let $X' \setminus X = \{e\}$.

(1) If $p(e) = p_1$ (i.e., e belongs to the root thread), the algorithm simply constructs a partial order K over the set X' such that $K|X = P$ and $e' <_K e$ for every event $e \in X'$ such that $e' <_{\mathsf{TO}} e$. Afterwards, the algorithm constructs the annotated partial order $\mathcal{K} = (X'_1, X'_2, K, \mathsf{val}', S', \mathsf{GoodW}')$ and returns the singleton set $\mathcal{A}_w = \{\mathsf{Closure}(\mathcal{K})\}$.
(2) If $p(e) \neq p_1$ (i.e., e belongs to the leaf threads), the algorithm first constructs a partial order K as in the previous item. Afterwards, it creates a new partial order K_i for every possible ordering of e with all events $e' \in X_2$ such that $e \bowtie e'$. Finally, the algorithm constructs the annotated partial orders $A = \mathcal{K}_i = (X'_1, X'_2, K_i, \mathsf{val}', S', \mathsf{GoodW}')$, and returns the set $\mathcal{A} = \{\mathsf{Closure}(\mathcal{K}_i) : \mathcal{K}_i \in A \text{ and } \mathsf{Closure}(\mathcal{K}_i) \neq \bot\}$.

Causally-happens-before maps, guarding reads and candidate writes. A *causally-happens-before (CHB) map* is a map $C : \mathcal{R} \to \mathcal{H} \to \mathcal{R} \cup \{\bot, \perp\!\!\!\perp\}$ such that for each read event $r \in \mathsf{dom}(C)$ and thread $p \in \mathcal{H}$ we have that $C(r)(p) \in \mathcal{R}_p \cup \{\bot, \perp\!\!\!\perp\}$. In words, C maps read events to functions that map every thread $p \in \mathcal{H}$ to a read event of p, or to some initial values $\{\bot, \perp\!\!\!\perp\}$. Given a trace t and an event $e \in \mathcal{E}(t)$, we define the *guarding read* $\mathsf{Guard}_t(e)$ of e in t as the last read event of $p(e)$ that happens before e in t, and $\mathsf{Guard}_t(e) = \bot$ if no such read event exists. Formally,

$$\mathsf{Guard}_t(e) = \max_t(\{r \in \mathcal{R}(t|p(e)) : r <_{\mathsf{TO}} e\})$$

where we take the maximum of the empty set to be $\bot$. Given a trace t, a CHB map C and a read event $r \in \text{enabled}(t)$, we define the *candidate write set* $M_t^C(r)$ of r in t given C as follows:

$$
\begin{aligned}
M_t^C(r) = \{ w \in \mathcal{W}(t) :\ & r \bowtie w \quad \text{and} \\
\text{either} \quad & \text{Guard}_t(w) = \bot \quad \text{and} \quad C(r)(p(w)) = \perp\!\!\perp \\
\text{or} \quad & \text{Guard}_t(w) \neq \bot \quad \text{and also} \quad C(r)(p(w)) \in \{\perp\!\!\perp, \bot\} \quad \text{or} \quad C(r)(p(w)) <_{\text{TO}} \text{Guard}_t(w)
\end{aligned}
$$

We refer to Figure 6 for an illustration of the above notation. Intuitively, $C(r)(p)$ encodes the prefix of the local trace of thread p that contains write events which have already been considered by the algorithm as good writes for r. Instead of the whole prefix, we store the last read of that prefix. The two special values $\perp\!\!\perp$ and $\bot$ encode the empty prefix, and the prefix before the first read. The guarding read of a write w is the last local read event the same thread that appears before w in the execution so far. Hence, if the guarding read of w appears before $C(r)(p)$, we know that w has been considered as a good write for r. The candidate write set for r contains writes that are considered as good writes for r in the current recursive step.

	τ_1	τ_2	τ_3
1	w_x		
2		w_y	
3		w_x	
4		r_y	
5		w_x	
6			r_y
7			w_x
8			r_x
9			w_x

(a) A trace t. Threads p_1 and p_3 have enabled events r_x^1 and r_x^3 (not shown), which access the variable x.

$$\text{enabled}(t) \cap \mathcal{E}_{p_1} = r_x^1$$
$$\text{enabled}(t) \cap \mathcal{E}_{p_3} = r_x^3$$

$$C(r_x^1) = \{(p_1, \bot), (p_2, e_4), (p_3, e_6)\}$$
$$C(r_x^3) = \{(p_1, \perp\!\!\perp), (p_2, \perp\!\!\perp), (p_3, \perp\!\!\perp)\}$$

$$M_t^C(r_x^1) = \{e_9\}$$
$$M_t^C(r_x^3) = \{e_1, e_3, e_5, e_7, e_9\}$$

(b) The candidate write sets of the read events r_x^1 and r_x^3 given the causally-happens-before map C.

Fig. 6. Example of a trace (Figure 6a) and candidate write sets of read events given their causally-happens-before maps (Figure 6b). We denote by e_i the i-th event of t.

Algorithm 5: VC-DPOR($\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW}), C$)

Input: A minimal closed annotated partial order $\mathcal{P}$, a CHB map C.

```
1 t′ ← Realize(𝒫)                                        // 𝒫 is closed hence realizable
2 t ← WExtend(t′)                    // Extend t′ until before the next read of each thread
3 foreach Q ∈ Extend(𝒫, ℰ(t), val_t, S, GoodW) do                  // Extensions of 𝒫 to ℰ(t)
4 │ C_Q ← C                                                      // Create a copy of the CHB C
5 │ ExtendRoot(Q, t, C_Q)                                           // Process the root thread
6 │ foreach p ∈ ℋ \ {p_1} do                                        // Process the leaf threads
7 │ │ ExtendLeaf(Q, t, C_Q, p)
8 │ end
9 end
```

Algorithm VC-DPOR. We are now ready to describe our main algorithm VC-DPOR for the enumerative exploration of the partitioning $\mathcal{T}_{\mathcal{H}}/\mathcal{VHB}$. The algorithm takes as input a minimal closed

annotated partial order $\mathcal{P}$ and a CHB map C. First, VC-DPOR calls Realize to obtain a linearization t' of $\mathcal{P}$ and constructs the write-extension t of t' which reveals new write events in t. Afterwards, the algorithm extends $\mathcal{P}$ to the set $\mathcal{E}(t)$ by calling Extend. Recall that Extend returns a set of minimal closed annotated partial orders. For every annotated partial order Q returned by Extend, the algorithm calls ExtendRoot to process the read event of the root thread p_1 that is enabled in t. Finally, the algorithm calls ExtendLeaf for every leaf thread $p \neq p_1$ to process the read event of p that is enabled in t. For the initial call, we construct an empty annotated partial order $\mathcal{P}$ and an initial CHB map C that for every read event $r \in \mathcal{R}$ and thread $p \in \mathcal{H}$ maps $C(r)(p) = \{\perp\!\!\!\perp\}$.

Algorithm 6: ExtendRoot($Q = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW}), t, C_Q$)

Input: A minimal closed annotated partial order Q, a trace t, a CHB map C_Q.

```
1  r ← enabled(t, p_1)                                   // The next enabled event in p_1 is a read
2  Y_1 ← M_t^{C_Q}(r) ∩ W_{p_1}                          // The set of local candidate writes of r
3  Y_2 ← M_t^{C_Q}(r) ∩ W_{≠p_1}                         // The set of remote candidate writes of r
4  foreach i ∈ [2] do                                    // i = 1 (i = 2) reads from local (remote) writes
5      S_r ← S ∪ {(r, i)}                                // The new side function
6      D_r ← {val_t(w) : w ∈ Y_i}                        // The set of values of candidate writes of r
7      foreach v ∈ D_r do                                // Every value v that r may read
8          val_r ← val_t ∪ {(r, v)}                      // The new value function
9          GoodW_r ← GoodW ∪ {(r, {w ∈ Y_i : val_t(w) = v})}   // The new good-writes function
10         K ← Extend(Q, X_1 ∪ X_2 ∪ {r}, val_r, S_r, GoodW_r)  // Returns one element
11         if K ≠ ⊥ then                                 // Extension is successful
12             Call VC-DPOR(K, C_Q)                      // Recurse
13     end
14 end
15 C_Q(r) ← {(p, max_t({R(t)|p})) : p ∈ H}               // The last read of each thread in t
```

Algorithm ExtendRoot**.** The algorithm takes as input a minimal closed annotated partial order Q, a trace t and a CHB map C_Q, and attempts all possible extensions of Q with the read event r of p_1 that is enabled in t to all possible values that are written in t. The algorithm first constructs two sets Y_1 and Y_2 which hold the local and remote, respectively, write events of t that are candidate writes for r according to the CHB map C_Q. Then, it iterates over the local ($i = 1$) and remote ($i = 2$) write choices for r in Y_i. Finally, the algorithm (i) collects all possible values that r may read from the set Y_i, (ii) constructs the appropriate new side function, value function and good-writes function, and (iii) calls Extend on these new parameters in order to establish the respective extension for r. For every such case, Extend returns a new minimal, closed annotated partial order $\mathcal{K}$ which is passed recursively to VC-DPOR.

Algorithm ExtendLeaf**.** The algorithm ExtendLeaf takes as input a minimal closed partial order Q, a trace t, a CHB map C_Q, and a thread $p \in \mathcal{H} \setminus \{p_1\}$. Similarly to ExtendRoot, ExtendLeaf attempts all possible extensions of Q with the read event r of p that is enabled in t to all possible values that are written in t. The main difference compared to ExtendRoot is that since r belongs to a leaf thread, Extend returns a set of minimal, closed annotated partial orders (as opposed to just one) which result from all possible orderings of r with the write events of X_2 that are conflicting with r. Then ExtendLeaf makes a recursive call to VC-DPOR for each such annotated partial order.

The following theorem states the main result of this paper.

Algorithm 7: ExtendLeaf($\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW}), t, C_{\mathcal{Q}}, p$)

Input: A minimal closed annotated partial order $\mathcal{Q}$, a trace t, a CHB map $C_{\mathcal{Q}}$, a thread p.

```
1  r ← enabled(t, p)                                          // The next enabled event in p is a read
2  D_r ← {val_t(w) : w ∈ M_t^{C_Q}(r)}                          // The set of values of candidate writes of r
3  foreach v ∈ D_r do                                          // Every value v that r may read
4  |  val_r ← val_t ∪ {(r, v)}                                  // The new value function
5  |  GoodW_r ← GoodW ∪ {(r, {w ∈ M_t^{C_Q}(r) : val_t(w) = v})}  // The new good-writes function
6  |  foreach K ∈ Extend(Q, X_1 ∪ X_2 ∪ {r}, val_r, S, GoodW_r) do   // Returns many elements
7  |  |  Call VC-DPOR(K, C_Q)                                   // Recurse
8  |  end
9  end
10 C_Q(r) ← {(p, max_t({R(t)|p})) : p ∈ H}                     // The last read of each thread in t
```

Theorem 5.1. *Consider a concurrent program $\mathcal{H}$ over a constant number of threads, and let $\mathcal{T}_{\mathcal{H}}^{\max}$ be the maximal trace space of $\mathcal{H}$.* VC-DPOR *solves the local-state reachability problem on $\mathcal{H}$ and requires $O\left(|\mathcal{T}_{\mathcal{H}}^{\max}/\mathcal{VHB}| \cdot \mathsf{poly}(n)\right)$ time, where n is the length of the longest trace in $\mathcal{T}_{\mathcal{H}}^{\max}$.*

We conclude with two remarks on space usage and the way lock events can be handled.

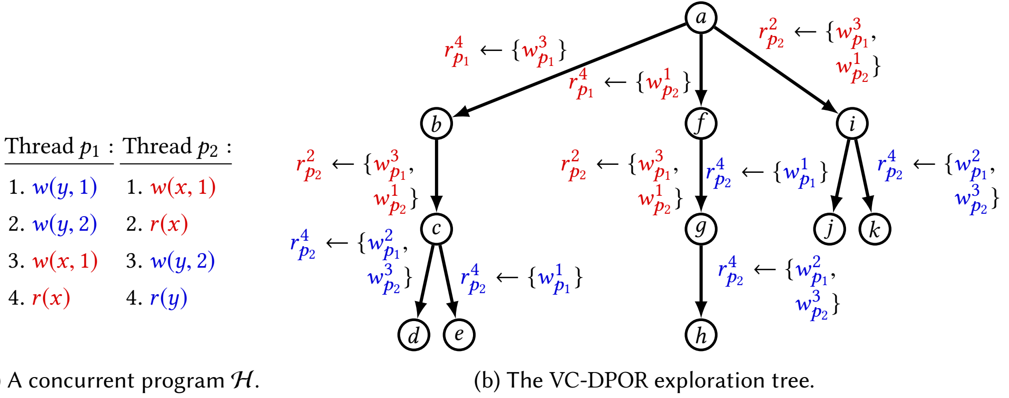
Remark 3 (Space complexity). *To make our presentation simpler so far,* VC-DPOR *and* ExtendLeaf *iterate over the set of annotated partial orders returned by* Extend*, which can be exponentially large. An efficient variant of* VC-DPOR *shall explore these sets recursively, instead of computing all elements of each set imperatively. This results in polynomial space complexity for* VC-DPOR.

Remark 4 (Handling locks). *For simplicity of presentation, so far we have neglected locks. However, lock events can be incorporated naturally, as follows.*

(1) Each lock-release event is a write event, writing an arbitrary value.
(2) Each lock-acquire event is a read event. Given two lock-acquire events r_1, r_2 the algorithm maintains that $\mathsf{GoodW}(r_1) \cap \mathsf{GoodW}(r_2) = \emptyset$

VC-DPOR **running example.** Figure 7 illustrates the main aspects of VC-DPOR (Algorithms 5, 6, and 7) on a small example. We start with an empty annotated partial order $\mathcal{P}$ and a CHB map C that is empty (i.e., $C(r)(p) = \{\perp\!\!\!\perp\}$ for every read event $r \in \mathcal{R}$ and thread $p \in \mathcal{H}$). The initial trace obtained in Line 1 of Algorithm 5 is $t' = \varepsilon$. Its write-extension t in Line 2 contains the three writes of p_1 and the first write of p_2. Next, Line 3 returns an annotated partial order $\mathcal{Q}_a$ that corresponds to the thread order $\mathsf{TO}|\mathcal{E}(t)$. In t, the root thread p_1 has an enabled event (which is always a read), so ExtendRoot (Algorithm 6) is called on $\mathcal{Q}_a$ and the (empty) CHB map $C_{\mathcal{Q}_a}$. (†)

The enabled read in Line 1 is $r_{p_1}^4$, its local candidate write (computed in Line 2) is $w_{p_1}^3$ and its remote candidate write (computed in Line 3) is $w_{p_2}^1$. This holds because $C_{\mathcal{Q}_a}(r)(p_1) = \{(p_1, \perp\!\!\!\perp), (p_2, \perp\!\!\!\perp)\}$, which allows any write event to be observed. For the local (Line 4, $i = 1$) candidate $w_{p_1}^3$, first the side function is updated with $\{(r_{p_1}^4, 1)\}$ in Line 5. Then in Line 6, the only considered value is 1. Thus, in Line 8 the value function is updated with $\{(r_{p_1}^4, 1)\}$, and in Line 9 the good-writes



(a) A concurrent program $\mathcal{H}$. (b) The VC-DPOR exploration tree.

Fig. 7. A program with two threads (Figure 7a) and the corresponding VC-DPOR exploration (Figure 7b).

function is updated with $\{(r_{p_1}^4, \{w_{p_1}^3\})\}$. Then, such an update is successfully realized in Line 10 by Extend, where the partial order is extended with $r_{p_1}^4$ and afterwards it is closed using algorithm Closure (Algorithm 1). Thus VC-DPOR (Algorithm 5) is recursively called on the corresponding annotated partial order $\mathcal{K}_a$ (and the empty CHB map C_{Q_a}), and we proceed to the child b of a.

In node b, no new event is added during the write-extension (Line 2), as $r_{p_1}^4$ is the last event of p_1, and in Line 3 we obtain Q_b. The only thread with an enabled read event is p_2, so ExtendLeaf (Algorithm 7) is called on Q_b and p_2 (and empty CHB map C_{Q_b}). The enabled read $r_{p_2}^2$ has candidate writes $w_{p_1}^3$ and $w_{p_2}^1$, both of which write the same value (c.f. Line 2), and hence the algorithm will allow $r_{p_2}^2$ to observe either. This is an example of the value-centric gains we obtain in this work. In Line 4 the value function is updated with $\{(r_{p_2}^2, 1)\}$, and in Line 5 the good-writes function is updated with $\{(r_{p_2}^2, \{w_{p_1}^3, w_{p_2}^1\})\}$. The realization of this update happens in Line 6 by Extend, where the partial order is extended with $r_{p_2}^2$ and then closed using algorithm Closure (Algorithm 1). One annotated partial order $\mathcal{K}_b$ is returned and it is the argument of the further VC-DPOR call (with an empty CHB map C_{Q_b}), we proceed to the child c of b. In node c, the write-extension adds the event $w_{p_2}^3$, which, in similar steps as before, will lead to nodes d and e.

Next, the recursion backtracks to the call of ExtendRoot in the node a (†). The second iteration ($i = 2$) of the loop in Line 4 proceeds, where the remote candidate write $w_{p_2}^1$ is considered for $r_{p_1}^4$. In a similar fashion, the descendants f, g, and h are created and h concludes with a maximal trace.

Finally, the recursion backtracks to the node a again, where ExtendRoot (†) concludes with updating the CHB map as follows: $C_{Q_a}(r_{p_1}^4) = \{(p_1, \bot),\ (p_2, \bot)\}$. The control-flow comes back to the initial VC-DPOR call (from Line 5), where the annotated partial order Q_a with the (now updated) CHB map C_{Q_a} is considered. The thread p_2 has an enabled read ($r_{p_2}^2$) in t, hence ExtendLeaf is called on Q_a, C_{Q_a}, and p_2. Eventually, the descendants i, j, and k are created and the exploration concludes. Note that in each of i, j, k, the thread p_1 has an enabled read $r_{p_1}^4$. However, note that $\text{Guard}_t(w_{p_1}^3) = \text{Guard}_t(w_{p_2}^1) = \bot$ and in all those nodes we have $C(r_{p_1}^4)(p_1) = \{(p_1, \bot), (p_2, \bot)\}$, and thus w_{pp1}^3 and $w_{p_2}^1$ are never considered as candidate writes for $r_{p_1}^4$. This illustrates how VC-DPOR never explores the same class of $\mathcal{VHB}$ twice.

6 EXPERIMENTS

We have seen in Theorem 3.1 that $\mathcal{VHB}$ is a coarse partitioning that can be explored efficiently by VC-DPOR. In this section we present an experimental evaluation of VC-DPOR on various classes of concurrent benchmarks, to assess

(1) the reduction of the trace-space partitioning achieved by $\mathcal{VHB}$, and
(2) the efficiency with which this partitioning is explored by VC-DPOR.

Implementation and experiments. To address the above questions, we have made a prototype implementation of VC-DPOR in the stateless model checker Nidhugg [Abdulla et al. 2015], which works on LLVM IR[2]. We have tested VC-DPOR on benchmarks coming in four classes:

(1) The TACAS Software Verification Competition (SV-COMP).
(2) Mutual-exclusion algorithms from the literature.
(3) Multi-threaded dynamic-programming algorithms that use memoization.
(4) Individual benchmarks that exercise various concurrency patterns.

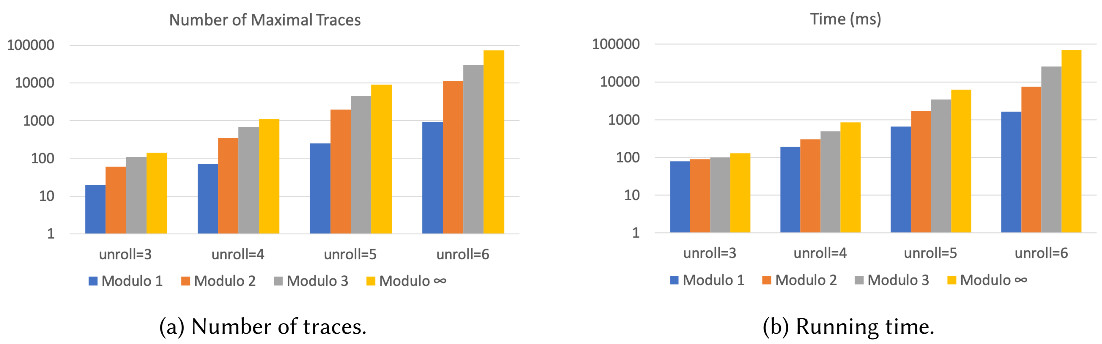
Each benchmark comes with a scaling parameter, which is either the number of threads, or an unroll bound on all loops of the benchmark (often the unroll bound also controls the number of threads that are spawned.) We have compared our algorithm with three other state-of-the-art DPOR algorithms that are implemented in Nidhugg, namely Source [Abdulla et al. 2014], Optimal [Abdulla et al. 2014] and Optimal* ("optimal with observers") [Aronis et al. 2018], as well as our own implementation of DC-DPOR [Chalupa et al. 2017]. For our experiments, we have used a Linux machine with Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz and 128GB of RAM. We have run Nidhugg with Clang and LLVM version 3.8. In all cases, we report the number of maximal traces and the total running time of each algorithm, subject to a timeout of 4 hours, indicated by "-".

Implementation details. Here we clarify some details regarding our implementation.

(1) In our theory so far, we have neglected dynamic thread creation for simplicity of presentation. In practice, all our benchmarks spawn threads dynamically. This situation is handled straightforwardly, by including in our partial orders the orderings that are naturally induced by spawn and join events.
(2) The root thread is chosen as the first thread that is spawned from the main thread. We make this choice instead of the main thread as in many benchmarks, the main thread mainly spawns worker threads and performs only a few concurrent operations.
(3) In our presentation of $\mathsf{Extend}(\mathcal{P}, X', \mathsf{val}', S', \mathsf{GoodW}')$, given $X' \setminus X = \{e\}$ such that e belongs to a leaf thread, we consider all possible orderings of e with conflicting events from all leaf threads. In our implementation, we relax this in two ways. Given a write event e_w, we say it is *never-good* if it does not belong to $\mathsf{GoodW}'(r)$ for any read event r. Further, given e_w and an annotated partial order $\mathcal{K}$, we say that e_w is *unobservable* in $\mathcal{K}$, if for every linearization of $\mathcal{K}$ no read event can observe e_w. Given two unordered conflicting write events from leaf threads, we do not order them if (i) both are never-good, or (ii) at least one is unobservable.

Value-centric gains. As a preliminary experimental step, we explore the gains of our value-centric technique on small variants of the simple benchmark `fib_bench` from SV-COMP. This benchmark consists of a main thread and two worker threads, and two global variables x and y. The first worker thread enters a loop in which it performs the update $x \leftarrow x + y$. Similarly, the second worker thread enters a loop in which it performs the update $y \leftarrow y + x$. To explore the sensitivity

[2]Code accessible at https://github.com/ViToSVK/nidhugg/tree/valuecentric_stable



(a) Number of traces.

(b) Running time.

Fig. 8. Number of traces (a) and running time (b) on variants of the fib_bench benchmark.

of our value-centric DPOR to values, we have created three variants fib_bench_1, fib_bench_2, fib_bench_3 of the main benchmark. In variant fib_bench_i each worker thread performs the addition modulo i. Hence, the first and the second worker performs the update $x \leftarrow (x + y) \mod i$ and $y \leftarrow (y + x) \mod i$, respectively. For smaller values of i, we expect more write events to write the same value, and thus VC-DPOR to benefit both in terms of the traces explored and the running time. Although simple, this experiment serves the purpose of quantifying the value-centric gains of VC-DPOR in a controlled benchmark. Figure 8 depicts the obtained results for the three variants of fib_bench, where Modulo $= \infty$ represents the original benchmark (i.e., without the modulo operation). We see that indeed, as i gets smaller, VC-DPOR benefits significantly in both number of traces and running time. Moreover, this benefit gets amplified with higher unroll bounds.

Benchmarks from SV-COMP. Here we present experiments on benchmarks from SV-COMP (along the industrial benchmark parker) (Table 1). We have replaced all assertions with simple read events. This way we ensure a fair comparison among all algorithms in exploring the trace-space of each benchmark, as an assertion violation would halt the search. We have verified that all assertion violations present in these benchmarks are detected by all algorithms before this modification. The scaling parameter in each case controls the size of the input benchmark in terms of loop unrolls.

Dynamic-programming benchmarks. Here we present experiments on various multi-threaded dynamic-programming algorithms (Table 2). For efficiency, these algorithms use memoization to avoid recomputing instances that correspond to the same sub-problem. The benchmarks consist of three or four threads. In each case, all-but-one threads are performing the dynamic programming computation, and one thread reads a flag signaling that the computation is finished, as well as the result of the computation. Each benchmark name contains either the substring "td" or the substring"bu", denoting that the dynamic programming table is computed top-down or bottom-up, respectively. The scaling parameter of each benchmark controls the different sizes of the input problem. The dynamic programming problems we use as benchmarks are the following.

- rod_cut computes, given one rod of a given length and prices for rods of shorter lengths, the maximum profit achievable by cutting the given rod.
- lis computes, given an array of non-repeating integers, the length of the longest increasing subsequence (not necessarily contiguous) in the array.
- coin_all computes, given an unlimited supply of coins of given denominations, the total number of distinct ways to get a desired change.

Table 1. Experimental comparison on SV-COMP benchmarks.

Benchmark	Maximal Traces					Time				
	VC-DPOR	Source	Optimal	Optimal*	DC-DPOR	VC-DPOR	Source	Optimal	Optimal*	DC-DPOR
parker(6)	**38670**	1100917	1100917	1023567	985807	**1m29s**	23m5s	24m29s	24m54s	46m41s
parker(7)	**52465**	1735432	1735432	1613807	1554237	**2m23s**	41m28s	44m41s	45m13s	1h27m
parker(8)	**68360**	2576147	2576147	2395947	2307467	**3m35s**	1h9m	1h15m	1h17m	2h29m
27_Boop(6)	**248212**	35079696	35079696	4750426	1468774	**3m26s**	2h54m	2h49m	26m22s	12m33s
27_Boop(7)	**420033**	-	-	10134616	2874202	**6m33s**	-	-	1h0m	27m21s
27_Boop(8)	**677870**	-	-	20003512	5268064	**11m54s**	-	-	2h7m	56m13s
30_Fun_Point(6)	**5040**	665280	665280	665280	665280	**5.52s**	4m2s	4m14s	4m36s	1m34s
30_Fun_Point(7)	**40320**	17297280	17297280	17297280	17297280	**57.50s**	2h7m	2h15m	2h29m	51m46s
30_Fun_Point(8)	**362880**	-	-	-	-	**10m51s**	-	-	-	-
45_monabsex(5)	**600**	14400	14400	9745	6197	**0.44s**	2.28s	2.36s	1.86s	1.50s
45_monabsex(6)	**13152**	518400	518400	291546	180126	**14.93s**	1m41s	1m41s	1m5s	1m0s
45_monabsex(7)	**423360**	25401600	25401600	11710405	7073803	**13m30s**	1h43m	1h40m	51m57s	56m16s
46_monabsex(5)	**1064**	14400	14400	5566	2653	**0.32s**	1.98s	2.02s	0.87s	0.51s
46_monabsex(6)	**21371**	518400	518400	157717	62864	**6.26s**	1m29s	1m23s	28.04s	10.33s
46_monabsex(7)	**621948**	25401600	25401600	6053748	2057588	**4m9s**	1h38m	1h23m	21m3s	7m24s
fk2012_true(3)	**12400**	42144	42144	42144	33886	**5.55s**	9.34s	10.59s	11.08s	13.13s
fk2012_true(4)	**252586**	1217826	1217826	1217826	888404	**2m3s**	5m6s	5m35s	6m11s	6m30s
fk2012_true(5)	**3757292**	24580886	24580886	24580886	16494444	**37m3s**	2h0m	2h12m	2h26m	2h28m
fkp2013_true(5)	**17751**	86400	86400	48591	25626	**3.75s**	16.40s	15.20s	9.70s	4.90s
fkp2013_true(6)	**513977**	3628800	3628800	1672915	786499	**2m18s**	14m27s	12m55s	6m34s	3m18s
fkp2013_true(7)	**20043857**	-	-	-	32244120	**2h16m**	-	-	-	3h11m
nondet-array(4)	**404**	2616	2616	688	592	**0.13s**	0.88s	0.80s	0.27s	0.20s
nondet-array(5)	**10804**	128760	128760	18665	15449	**3.11s**	46.23s	46.99s	8.66s	4.26s
nondet-array(6)	**430004**	9854640	9854640	711276	571476	**2m36s**	1h15m	1h14m	7m45s	3m30s
pthread-de(7)	**327782**	4027216	4027216	4027216	829168	**1m10s**	12m9s	13m32s	17m36s	2m12s
pthread-de(8)	**2457752**	43976774	43976774	43976774	6984234	**10m29s**	2h29m	2h46m	3h24m	22m1s
pthread-de(9)	**18568126**	-	-	-	59287740	**1h33m**	-	-	-	3h37m
reorder_5(5)	**1016**	1755360	1755360	68206	4978	**0.21s**	9m0s	9m22s	26.45s	0.34s
reorder_5(8)	**247684**	-	-	-	437725	1m47s	-	-	-	**1m29s**
reorder_5(9)	**1644716**	-	-	-	1792290	22m53s	-	-	-	**12m38s**
scull_true(3)	**3426**	617706	617706	436413	172931	**19.77s**	9m46s	10m22s	9m7s	4m46s
scull_true(4)	**8990**	2732933	2732933	1840022	656100	**1m7s**	51m37s	54m33s	46m12s	25m56s
scull_true(5)	**19881**	9488043	9488043	6070688	1988798	**3m8s**	3h29m	3h42m	2h54m	1h47m
sigma_false(7)	**12509**	135135	135135	30952	30952	**10.52s**	55.87s	1m0s	18.65s	17.87s
sigma_false(8)	**133736**	2027025	2027025	325488	325488	**2m4s**	16m21s	18m45s	4m12s	3m44s
sigma_false(9)	**1625040**	-	-	3845724	3845724	**31m53s**	-	-	1h6m	53m28s
check_bad_arr(5)	**4046**	12838	12838	10989	6689	2.74s	6.98s	6.83s	6.49s	**2.72s**
check_bad_arr(6)	**87473**	357368	357368	307097	187377	1m47s	5m21s	4m36s	4m24s	**1m33s**
check_bad_arr(7)	**1856332**	8245810	8245810	6943293	4069592	2h11m	3h9m	2h19m	2h12m	**1h7m**
32_pthread5(1)	**20**	24	24	24	**20**	**0.05s**	**0.04s**	**0.04s**	0.06s	0.06s
32_pthread5(2)	**1470**	1890	1890	1806	**1470**	0.67s	**0.38s**	0.45s	0.54s	0.67s
32_pthread5(3)	**226800**	302400	302400	280800	**226800**	2m30s	**1m14s**	1m17s	1m17s	2m21s
fkp2014_true(2)	**16**	**16**	**16**	**16**	**16**	**0.05s**	**0.05s**	**0.04s**	**0.04s**	**0.05s**
fkp2014_true(3)	**1098**	**1098**	**1098**	**1098**	**1098**	0.86s	**0.19s**	**0.20s**	0.21s	0.72s
fkp2014_true(4)	**207024**	**207024**	**207024**	**207024**	**207024**	3m40s	**39.84s**	41.70s	44.67s	3m15s
singleton(8)	**2**	40320	40320	8	8	0.06s	14.92s	15.24s	**0.04s**	0.09s
singleton(9)	**2**	362880	362880	9	9	0.09s	2m31s	2m32s	**0.05s**	0.15s
singleton(10)	**2**	3628800	3628800	10	10	0.16s	27m33s	28m9s	**0.05s**	0.19s
stack_true(9)	**48620**	**48620**	**48620**	**48620**	**48620**	2m24s	**37.55s**	38.47s	40.06s	2m23s
stack_true(10)	**184756**	**184756**	**184756**	**184756**	**184756**	11m58s	**2m31s**	2m40s	2m50s	11m1s
stack_true(11)	**705432**	**705432**	**705432**	**705432**	**705432**	58m34s	**10m32s**	11m8s	11m48s	54m42s
48_ticket_lock(2)	**6**	**6**	**6**	**6**	**6**	0.05s	**0.03s**	**0.04s**	**0.04s**	0.05s
48_ticket_lock(3)	**204**	**204**	**204**	**204**	**204**	0.25s	**0.08s**	0.10s	**0.09s**	0.34s
48_ticket_lock(4)	**41400**	**41400**	**41400**	**41400**	**41400**	55.67s	**13.88s**	15.27s	16.56s	52.57s

- `coin_min` computes, given an unlimited supply of coins of given denominations, the minimum number of coins required to get a desired change.
- `bin_nocon` computes the number of binary strings of a given length that do not contain the substring '11'.

Table 2. Experimental comparison on dynamic-programming benchmarks.

Benchmark	**Maximal Traces**					**Time**				
	VC-DPOR	**Source**	**Optimal**	**Optimal***	**DC-DPOR**	**VC-DPOR**	**Source**	**Optimal**	**Optimal***	**DC-DPOR**
rod_cut_td3(7)	**4324**	102128	102128	51974	23143	**33.23s**	4m14s	7m43s	3m47s	1m28s
rod_cut_td3(8)	**14744**	508646	508646	257707	114624	**3m4s**	27m32s	57m42s	28m2s	12m9s
rod_cut_td3(9)	**50320**	2574752	-	1300067	577682	**17m24s**	3h0m	-	3h27m	1h39m
rod_cut_td4(3)	**1478**	91592	91592	17451	4810	**0.97s**	1m29s	1m49s	21.79s	1.46s
rod_cut_td4(4)	**21358**	2459640	2459640	359609	85203	**28.55s**	1h6m	1h33m	14m2s	57.94s
rod_cut_td4(5)	**433371**	-	-	-	2551714	**20m57s**	-	-	-	1h22m
rod_cut_bu3(6)	**19933**	183516	183516	147746	71670	**56.15s**	2m23s	3m59s	3m26s	2m3s
rod_cut_bu3(7)	**99622**	1101084	1101084	886466	429494	**8m6s**	17m52s	33m33s	29m19s	21m40s
rod_cut_bu3(8)	**498061**	6606492	-	-	2574902	**1h6m**	2h12m	-	-	3h30m
rod_cut_bu4(2)	**1901**	33912	33912	14667	5377	**0.70s**	11.76s	13.36s	6.75s	1.15s
rod_cut_bu4(3)	**74541**	2246424	2246424	913299	292633	**46.95s**	18m50s	24m12s	11m37s	1m52s
rod_cut_bu4(4)	**3007476**	-	-	-	-	**1h17m**	-	-	-	-
lis_bu3(8)	**118812**	1744064	1744064	475986	358347	**4m24s**	33m22s	1h0m	18m27s	7m24s
lis_bu3(9)	**368400**	7001792	-	1439130	1092553	**15m49s**	2h38m	-	1h10m	27m6s
lis_bu3(10)	**3133740**	-	-	-	-	**3h59m**	-	-	-	-
lis_bu4(2)	**1137**	18522	18522	7936	2828	**0.45s**	8.45s	9.41s	4.42s	0.52s
lis_bu4(3)	**29931**	1024002	1024002	364560	101766	**12.70s**	10m36s	12m49s	5m0s	19.41s
lis_bu4(4)	**1222278**	-	-	-	5679067	**16m34s**	-	-	-	37m20s
coin_all_td3(9)	**4015**	566214	566214	23308	8071	22.23s	34m25s	1h20m	2m36s	**21.13s**
coin_all_td3(10)	**9052**	2444048	-	59168	19829	**1m2s**	2h56m	-	8m20s	1m3s
coin_all_td3(19)	**637859**	-	-	-	1528102	**2h43m**	-	-	-	3h5m
coin_all_td4(2)	**5938**	6406248	-	74153	20668	**4.86s**	3h46m	-	3m27s	6.47s
coin_all_td4(3)	**68966**	-	-	1549115	319142	**1m36s**	-	-	2h15m	2m34s
coin_all_td4(5)	**379086**	-	-	-	2857926	**16m12s**	-	-	-	36m32s
coin_min_td3(8)	**46535**	1902262	1902262	981936	382275	**3m0s**	1h13m	2h12m	1h12m	14m0s
coin_min_td3(9)	**154663**	-	-	-	1634899	**11m36s**	-	-	-	1h8m
coin_min_td3(11)	**1312252**	-	-	-	-	**2h4m**	-	-	-	-
coin_min_td4(4)	**9912**	1470312	1470312	208367	46634	**30.52s**	36m17s	51m36s	7m6s	47.93s
coin_min_td4(5)	**102154**	-	-	3534815	718883	**6m7s**	-	-	2h59m	14m59s
coin_min_td4(6)	**1490420**	-	-	-	-	**1h52m**	-	-	-	-
bin_nocon_td3(7)	**13202**	1664672	1664672	471151	121350	**29.57s**	48m34s	1h26m	26m11s	2m4s
bin_nocon_td3(8)	**44802**	-	-	2825725	603668	**1m54s**	-	-	3h17m	12m32s
bin_nocon_td3(11)	**922114**	-	-	-	-	**1h0m**	-	-	-	-
bin_nocon_bu3(6)	**52500**	773122	773122	115625	75000	1m15s	12m37s	19m44s	3m5s	**1m8s**
bin_nocon_bu3(7)	**262500**	5411854	5411854	578125	375000	7m27s	1h45m	2h52m	19m54s	**6m50s**
bin_nocon_bu3(8)	**1312500**	-	-	2890625	1875000	45m2s	-	-	2h1m	**41m9s**

Mutual-exclusion benchmarks. Here we present experiments on various mutual-exclusion algorithms from the literature (Table 3). In particular, we use the two-thread solutions of Dijkstra [Dijkstra 1983], Kessels [Kessels 1982], Tsay [Tsay 1998], Peterson [Peterson 1981], Peterson-Fischer [Peterson and Fischer 1977], Szymanski [Szymanski 1988], Dekker [Knuth 1966], as well as various solutions of Correia-Ramalhete [Correia and Ramalhete 2016]. In addition, we use the two-thread and three-thread versions of Burns's algorithm [Burns and Lynch 1980]. These protocols exercise a wide range of communication patterns, based, e.g., on the number of shared variables and the number of sequentially consistent stores/loads required to enter/leave the critical section. In all these benchmarks, each thread executes the corresponding protocol to enter a (empty) critical section a number of times, the latter controlled by the scaling parameter.

Individual benchmarks. Here we present experiments on individual benchmarks (Table 4): `eratosthenes` consists of two threads computing the sieve of Eratosthenes in parallel; `redundant_co` consists of three threads, two of which repeatedly write to a variable and one reads from it; `float_read` consists of several threads, each writing once to a variable, and one reading from it (adapted from [Aronis et al. 2018]); `opt_lock` consists of three threads in an optimistic-lock scheme. The scaling parameter controls the size in terms of loop unrolls.

Table 3. Experimental comparison on mutual-exclusion benchmarks.

Benchmark	Maximal Traces					Time				
	VC-DPOR	Source	Optimal	Optimal*	DC-DPOR	VC-DPOR	Source	Optimal	Optimal*	DC-DPOR
tsay(2)	**2488**	7469	7469	7469	7469	**0.81s**	2.46s	2.76s	2.99s	1.82s
tsay(3)	**241822**	1414576	1414576	1414576	1414576	**1m38s**	10m2s	10m54s	12m1s	7m42s
tsay(4)	**24609389**	-	-	-	-	**3h51m**	-	-	-	-
peter_fisch(2)	**1371**	4386	4386	4386	4386	**0.69s**	1.56s	1.61s	1.73s	1.16s
peter_fisch(3)	**70448**	430004	430004	430004	430004	**34.03s**	2m54s	3m10s	3m31s	2m20s
peter_fisch(4)	**3747718**	-	-	-	-	**41m31s**	-	-	-	-
peterson(5)	**86929**	268706	268706	268706	256457	**32.42s**	49.22s	54.60s	1m4s	1m32s
peterson(6)	**880069**	3462008	3462008	3462008	3303617	**7m10s**	11m50s	13m18s	15m51s	25m29s
peterson(7)	**9013381**	45046254	45046254	-	-	**1h30m**	2h56m	3h21m	-	-
lamport(2)	**958**	3940	3940	2454	1456	**0.39s**	0.75s	0.77s	0.59s	0.45s
lamport(3)	**57436**	741370	741370	328764	130024	**28.14s**	2m24s	2m43s	1m29s	52.24s
lamport(4)	**3723024**	-	-	-	13088038	**49m40s**	-	-	-	2h26m
dekker(5)	**89647**	435245	435245	435245	435245	**29.78s**	1m14s	1m23s	1m37s	2m14s
dekker(6)	**932559**	6745775	6745775	6745775	6745775	**6m44s**	21m36s	24m12s	28m22s	42m46s
dekker(7)	**9837974**	-	-	-	-	**1h28m**	-	-	-	-
X2Tv6(3)	**7859**	20371	20371	20371	20371	**3.89s**	5.35s	5.68s	6.58s	7.69s
X2Tv6(4)	**152999**	596354	596354	596354	596354	**1m38s**	3m6s	3m23s	3m47s	5m17s
X2Tv6(5)	**3058189**	17836411	17836411	17836411	17836411	**46m41s**	1h51m	2h3m	2h21m	3h36m
kessels(3)	**8900**	13856	13856	13856	13856	**2.80s**	5.07s	5.45s	5.98s	3.70s
kessels(4)	**194858**	323400	323400	323400	323400	**1m13s**	2m19s	2m30s	2m48s	1m41s
kessels(5)	**4379904**	7763704	7763704	7763704	7763704	**35m59s**	1h8m	1h13m	1h22m	53m50s
X2Tv7(9)	**452142**	2004774	2004774	2004774	2004774	**7m34s**	24m59s	27m10s	29m54s	13m36s
X2Tv7(10)	**1721564**	7708671	7708671	7708671	7708671	**35m19s**	1h47m	1h58m	2h10m	1h1m
X2Tv7(11)	**6584004**	-	-	-	-	**2h37m**	-	-	-	-
X2Tv2(2)	**894**	1293	1293	1293	1293	**0.32s**	0.46s	0.46s	0.51s	0.50s
X2Tv2(3)	**42141**	69316	69316	69316	69316	**17.73s**	29.21s	31.04s	34.65s	22.01s
X2Tv2(4)	**1827915**	3552837	3552837	3552837	3552837	**17m21s**	31m13s	33m46s	37m35s	25m52s
burns(4)	**381**	140380	140380	140380	140380	**0.31s**	1m24s	1m28s	1m37s	1m8s
burns(5)	**1415**	2916980	2916980	2916980	2916980	**0.98s**	35m29s	38m9s	41m55s	29m25s
burns(11)	**4114995**	-	-	-	-	**1h48m**	-	-	-	-
burns3(1)	**67**	849	849	849	849	**0.09s**	0.45s	0.40s	0.44s	0.49s
burns3(2)	**11297**	1490331	1490331	1490331	1490331	**16.27s**	16m49s	17m32s	20m4s	26m4s
burns3(3)	**1638338**	-	-	-	-	**1h0m**	-	-	-	-
X2Tv10(2)	**4130**	5079	5079	5079	5079	1.81s	1.94s	1.95s	2.18s	**1.71s**
X2Tv10(3)	**213381**	308433	308433	308433	308433	**1m47s**	2m15s	2m26s	2m39s	1m56s
X2Tv10(4)	**10274441**	17910500	17910500	17910500	17910500	**1h58m**	2h48m	3h2m	3h29m	2h35m
X2Tv5(4)	**38743**	46161	46161	46161	46161	**14.34s**	21.05s	22.57s	24.92s	15.35s
X2Tv5(5)	**595527**	730647	730647	730647	730647	**4m37s**	6m28s	6m57s	7m50s	5m2s
X2Tv5(6)	**9312813**	11755440	11755440	11755440	11755440	**1h26m**	2h2m	2h17m	2h33m	1h37m
X2Tv1(6)	**224803**	253042	253042	253042	253042	1m45s	2m19s	2m27s	2m46s	**1m42s**
X2Tv1(7)	**1880095**	2115302	2115302	2115302	2115302	18m4s	21m56s	23m59s	26m35s	**17m31s**
X2Tv1(8)	**15873308**	17857733	17857733	-	17857733	2h59m	3h29m	3h49m	-	**2h51m**
X2Tv8(3)	**6168**	9894	9894	8700	8434	2.79s	**2.56s**	2.63s	2.64s	3.15s
X2Tv8(4)	**122932**	228417	228417	194206	186040	1m8s	**1m7s**	1m13s	1m10s	1m30s
X2Tv8(5)	**2503292**	5391534	5391534	4428748	4192466	31m12s	**31m4s**	34m43s	32m37s	44m43s
X2Tv9(3)	**7234**	7304	7304	7304	7304	2.53s	**2.11s**	2.23s	2.49s	2.41s
X2Tv9(4)	**150535**	153725	153725	153725	153725	1m3s	**52.80s**	56.85s	1m3s	56.86s
X2Tv9(5)	**3261067**	3324991	3324991	3324991	3324991	29m53s	**22m17s**	24m10s	27m11s	27m10s
szymanski(3)	**27892**	27951	27951	27951	27951	12.06s	**5.06s**	5.66s	6.69s	9.81s
szymanski(4)	**395743**	396583	396583	396583	396583	4m0s	**1m26s**	1m39s	1m49s	3m14s
szymanski(5)	**5734528**	5746703	5746703	5746703	5746703	1h17m	**25m17s**	28m59s	32m36s	1h1m

Summary. For the sake of completeness, we refer to Table 5 for some statistics on our benchmark set. Entries marked with "U" denote that the corresponding parameter is controlled by the unroll bound of the respective benchmark. In a variety of cases, the $\mathcal{VHB}$ partitioning is significantly coarser than each of the partitionings constructed by the other algorithms. This coarseness makes VC-DPOR more efficient in its exploration than the alternatives. We note that in some cases, $\mathcal{VHB}$ offers little-to-no reduction, and then VC-DPOR becomes slower than the alternatives, due to the overhead incurred in constructing $\mathcal{VHB}$. For example, for the benchmark `reorder_5` of Table 1, the partitioning reduction achieved by VC-DPOR is large enough compared to Source, Optimal

Table 4. Experimental comparison on individual benchmarks.

Benchmark	Maximal Traces					Time				
	VC-DPOR	Source	Optimal	Optimal*	DC-DPOR	VC-DPOR	Source	Optimal	Optimal*	DC-DPOR
eratosthenes(5)	**3500**	1527736	1527736	27858	19991	**16.92s**	18m37s	20m39s	41.14s	1m29s
eratosthenes(7)	**29320**	-	-	253792	189653	**3m37s**	-	-	9m29s	19m41s
eratosthenes(8)	**110380**	-	-	938756	710551	**11m29s**	-	-	42m27s	1h4m
redundant_co(2)	**11**	1969110	1969110	5401	729	**0.06s**	7m16s	7m32s	1.51s	0.07s
redundant_co(8)	**35**	-	-	1118305	35937	**0.09s**	-	-	13m24s	0.97s
redundant_co(9)	**39**	-	-	1778221	50653	**0.07s**	-	-	23m49s	1.35s
float_read(9)	**9**	3628800	3628800	2305	10	**0.05s**	26m30s	26m38s	1.27s	**0.04s**
float_read(15)	**15**	-	-	245761	16	**0.65s**	-	-	3m52s	0.74s
float_read(16)	**16**	-	-	524289	17	**1.42s**	-	-	9m25s	1.44s
opt_lock(2)	**2497**	69252	69252	11982	6475	**1.50s**	15.10s	15.53s	3.25s	2.50s
opt_lock(3)	**80805**	15036174	15036174	416850	212877	**52.13s**	1h5m	1h9m	2m9s	1m29s
opt_lock(4)	**2543298**	-	-	14038926	6743831	**37m41s**	-	-	1h27m	1h2m

Table 5. Benchmark statistics.

Benchmark	LOC	Var	Locks	Threads	Benchmark	LOC	Var	Locks	Threads	Benchmark	LOC	Var	Locks	Threads
parker	134	4	0	2	48_ticket_lock	52	3	1	U	dekker	91	4	0	2
27_Boop	74	4	0	4	rod_cut_td3	50	51	0	3	X2Tv6	75	4	0	2
30_Fun_Point	67	1	1	U	rod_cut_td4	62	51	0	4	kessels	44	3	0	2
45_monabsex	24	1	0	U	rod_cut_bu3	36	51	0	3	X2Tv7	83	3	0	2
46_monabsex	22	2	0	U	rod_cut_bu4	37	51	0	4	X2Tv2	65	3	0	2
fk2012_true	100	1	2	3	lis_bu3	47	51	0	3	burns	70	3	0	2
fkp2013_true	26	1	0	U	lis_bu4	48	51	0	4	burns3	70	4	0	3
nondet-array	29	1	0	U	coin_all_td3	51	151	0	3	X2Tv10	91	3	0	2
pthread-de	67	1	1	U	coin_all_td4	53	151	0	4	X2Tv5	55	4	0	2
reorder_5	1227	4	0	U	coin_min_td3	46	51	0	3	X2Tv1	56	3	0	2
scull_true	389	7	1	3	coin_min_td4	52	51	0	4	X2Tv8	64	4	0	2
sigma_false	36	1	0	U	bin_nocon_td3	43	101	0	3	X2Tv9	61	3	0	2
check_bad_arr	33	1	0	U	bin_nocon_bu3	53	101	0	3	szymanski	93	3	0	2
32_pthread5	87	4	1	U	tsay	54	3	0	2	eratosthenes	25	U	0	2
fkp2014_true	36	2	1	U	peter_fisch	59	3	0	2	redundant_co	23	1	0	2
singleton	43	1	0	U	peterson	68	4	0	2	float_read	25	1	0	U
stack_true	104	U	1	2	lamport	83	5	0	2	opt_lock	31	2	0	3

and Optimal* that makes VC-DPOR significantly faster than each of these techniques. However, although the partitioning of VC-DPOR is smaller than DC-DPOR, the corresponding reduction is not large enough to make VC-DPOR faster than DC-DPOR in this benchmark (in general, VC-DPOR has a larger polynomial overhead than DC-DPOR.) Similarly, for the benchmark `X2Tv9` of Table 3, the reduction of the $\mathcal{VHB}$ partitioning is quite small, and although Source is the slowest algorithm in theory, its more lightweight nature makes it faster in practice for this benchmark. Finally, we also identify benchmarks such as `stack_true` and `48_ticket_lock` where there is no trace reduction at all, and are better handled by existing methods. We note that our approach is fairly different from the literature, and our implementation of VC-DPOR still largely unoptimized. We identify potential for improving the performance of VC-DPOR by improving the closure computation, as well as reducing (or eliminating) the number of non-maximal traces explored by the algorithm.

7 RELATED WORK AND CONCLUSIONS

The formal analysis of concurrent programs is a major challenge in verification, and has been a subject of extensive research [Cadiou and Lévy 1973; Clarke et al. 1986; Farzan and Kincaid 2012; Farzan and Madhusudan 2009; Lal and Reps 2009; Lipton 1975; Petri 1962]. Since it is hard to reproduce bugs by testing due to scheduling nondeterminism, systematic state space exploration by model checking is an important approach for the problem [Alglave et al. 2013; Andrews et al. 2004; Clarke et al. 1999a; Godefroid 2005; Musuvathi and Qadeer 2007]. In this direction, stateless

model checking has been employed to combat state-space explosion [Godefroid 1996, 1997, 2005; Madan Musuvathi 2007].

To deal with the exponential number of interleavings faced by the early model checking [Godefroid 1997], several reduction techniques have been proposed such as POR and context bounding [Musuvathi and Qadeer 2007; Peled 1993]. Several POR methods, based on persistent set [Clarke et al. 1999b; Godefroid 1996; Valmari 1991] and sleep set techniques [Godefroid 1997], have been studied. DPOR techniques were first proposed in [Flanagan and Godefroid 2005], and several variants and improvements have been made since [Lauterburg et al. 2010; Saarikivi et al. 2012; Sen and Agha 2006, 2007; Tasharofi et al. 2012]. In [Abdulla et al. 2014], source sets and wakeup trees were developed to make DPOR optimal, and the underlying computational problems were further studied in [Nguyen et al. 2018]. Besides the present work, further improvements over optimal DPOR have been made in [Aronis et al. 2018; Chalupa et al. 2017], as well as with maximal causal models [Huang 2015; Huang and Huang 2017]. Other techniques such as unfoldings have also been explored [Kähkönen et al. 2012; McMillan 1995; Rodríguez et al. 2015]. Techniques for POR have also been applied to relaxed memory models [Abdulla et al. 2015; Demsky and Lam 2015; Huang and Huang 2016; Kokologiannakis et al. 2017; Wang et al. 2008] and message passing programs [Godefroid 1996; Godefroid et al. 1995; Katz and Peled 1992].

In this work, we have introduced a new equivalence on traces, called the value-happens-before equivalence $\mathcal{VHB}$, which considers the values of trace events in order to determine whether two traces are equivalent. We have shown that $\mathcal{VHB}$ is coarser than the standard happens-before equivalence, which is the theoretical foundation of the majority of DPOR algorithms. In fact, this coarsening occurs even when there are no concurrent write events. In addition, we have developed an algorithm VC-DPOR that relies on $\mathcal{VHB}$ to partition the trace space into equivalence classes and explore each class efficiently. Our experiments show that, in a variety of benchmarks, $\mathcal{VHB}$ indeed produces smaller partitionings than those explored by alternative, state-of-the-art methods, which often leads to a large reduction in running times.

ACKNOWLEDGMENTS

The authors would also like to thank anonymous referees for their valuable comments and helpful suggestions. This work is supported by the Austrian Science Fund (FWF) NFN grants S11407-N23 (RiSE/SHiNE) and S11402-N23 (RiSE/SHiNE), by the Vienna Science and Technology Fund (WWTF) Project ICT15-003, and by the Austrian Science Fund (FWF) Schrodinger grant J-4220.

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction *(POPL)*.

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *TACAS*.

Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. 2017. Context-Sensitive Dynamic Partial Order Reduction. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 526–543.

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV*.

Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. 2004. Zing: A Model Checker for Concurrent Software. In *CAV*.

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal Dynamic Partial Order Reduction with Observers. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 229–248.

James Burns and Nancy A Lynch. 1980. Mutual exclusion using invisible reads and writes. In *In Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*. Citeseer.

Jean-Marie Cadiou and Jean-Jacques Lévy. 1973. Mechanizable proofs about parallel processes. In *SWAT*.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158119

Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-centric Dynamic Partial Order Reduction. arXiv:arXiv:1909.00989

E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. 1999b. State space reduction using partial order techniques. *STTT* 2, 3 (1999), 279–287.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (1986).

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999a. *Model Checking*. MIT Press, Cambridge, MA, USA.

Andreia Correia and Pedro Ramalhete. 2016. 2-thread software solutions for the mutual exclusion problem. https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/cr2t-2016.pdf.

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed Stateless Model Checking for SC and TSO *(OOPSLA)*. ACM, New York, NY, USA, 20–36. https://doi.org/10.1145/2814270.2814297

E. W. Dijkstra. 1983. Solution of a Problem in Concurrent Programming Control. *Commun. ACM* 26, 1 (Jan. 1983), 21–22. https://doi.org/10.1145/357980.357989

Azadeh Farzan and Zachary Kincaid. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. In *CAV*.

Azadeh Farzan and P. Madhusudan. 2009. The Complexity of Predicting Atomicity Violations. In *TACAS*.

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *POPL*.

P. Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag, Secaucus, NJ, USA.

Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *POPL*.

Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *FMSD* 26, 2 (2005), 77–101.

Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. 1995. State-space Caching Revisited. *FMSD* 7, 3 (1995), 227–241.

Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *PLDI*.

Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. *SIGPLAN Not.* 51, 10 (Oct. 2016), 447–461. https://doi.org/10.1145/3022671.2984025

Shiyou Huang and Jeff Huang. 2017. Speeding Up Maximal Causality Reduction with Static Dependency Analysis. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain.* 16:1–16:22. https://doi.org/10.4230/LIPIcs.ECOOP.2017.16

Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. 2012. Using Unfoldings in Automated Testing of Multithreaded Programs. In *ACSD*.

Shmuel Katz and Doron Peled. 1992. Defining Conditional Independence Using Collapses. *Theor. Comput. Sci.* 101, 2 (1992), 337–359.

J. L. W. Kessels. 1982. Arbitration without common modifiable variables. *Acta Informatica* 17, 2 (01 Jun 1982), 135–141. https://doi.org/10.1007/BF00288966

Donald E. Knuth. 1966. Additional Comments on a Problem in Concurrent Programming Control. *Commun. ACM* 9, 5 (May 1966), 321–322. https://doi.org/10.1145/355592.365595

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

Akash Lal and Thomas Reps. 2009. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. *FMSD* 35, 1 (2009), 73–97.

Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. 2010. Evaluating Ordering Heuristics for Dynamic Partial-order Reduction Techniques. In *FASE*.

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

Tom Ball Madan Musuvathi, Shaz Qadeer. 2007. *CHESS: A systematic testing tool for concurrent software.* Technical Report.

A Mazurkiewicz. 1987. Trace Theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency.* Springer-Verlag New York, Inc., 279–324.

K. L. McMillan. 1995. A Technique of State Space Search Based on Unfolding. *FMSD* 6, 1 (1995), 45–65.

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. *SIGPLAN Not.* 42, 6 (2007), 446–455.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*.

Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. 2018. 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, July 14-17, 2018, Proceedings, Part II*. 354–371. https://doi.org/10.1007/978-3-319-96142-2_22

Doron Peled. 1993. All from One, One for All: On Model Checking Using Representatives. In *CAV*.

Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.* 12 (1981), 115–116.

Gary L. Peterson and Michael J. Fischer. 1977. Economical Solutions for the Critical Section Problem in a Distributed System (Extended Abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC '77)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/800105.803398

Carl Adam Petri. 1962. *Kommunikation mit Automaten.* Ph.D. Dissertation. Universität Hamburg.

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

Olli Saarikivi, Kari Kahkonen, and Keijo Heljanko. 2012. Improving Dynamic Partial Order Reductions for Concolic Testing. In *ACSD*.

Koushik Sen and Gul Agha. 2006. Automated Systematic Testing of Open Distributed Programs. In *FASE*.

Koushik Sen and Gul Agha. 2007. A Race-detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs. In *HVC*.

B. K. Szymanski. 1988. A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait. In *Proceedings of the 2Nd International Conference on Supercomputing (ICS '88)*. ACM, New York, NY, USA, 621–626. https://doi.org/10.1145/55364.55425

Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A Novel Dynamic Partial-order Reduction Technique for Testing Actor Programs. In *FMOODS/FORTE*.

Yih-Kuen Tsay. 1998. Deriving a Scalable Algorithm for Mutual Exclusion. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*. Springer-Verlag, London, UK, UK, 393–407. http://dl.acm.org/citation.cfm?id=645955.675799

Antti Valmari. 1991. Stubborn Sets for Reduced State Space Generation. In *Petri Nets*.

Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole Partial Order Reduction. In *TACAS*.