Data-centric dynamic partial order reduction

1 Anonymous, 2 Anonymous, 3 Anonymous, 4 Anonymous

Deposited at: 12 Dec 2018 11:53 ISSN: 2664-1690

IST Austria (Institute of Science and Technology Austria)
Am Campus 1
A-3400 Klosterneuburg, Austria

Institute of Science and Technology

Data-centric Dynamic Partial Order Reduction

1 Anonymous and 2 Anonymous and 3 Anonymous and 4 Anonymous

Technical Report No. IST-2016-620-v1+1
Deposited at 15 Jul 2016 08:43
https://repository.ist.ac.at/620/1/main.pdf

IST Austria (Institute of Science and Technology Austria)
Am Campus 1
A-3400 Klosterneuburg, Austria

Data-centric Dynamic Partial Order Reduction

Abstract

We present a new dynamic partial-order reduction method for stateless model checking of concurrent programs. A common approach for exploring program behaviors relies on enumerating the traces of the program, without storing the visited states (aka *stateless* exploration). As the number of distinct traces grows exponentially, dynamic partial-order reduction (DPOR) techniques have been successfully used to partition the space of traces into equivalence classes (*Mazurkiewicz* partitioning), with the goal of exploring only few representative traces from each class.

We introduce a new equivalence on traces under sequential consistency semantics, which we call the *observation* equivalence. Two traces are observationally equivalent if every read event observes the same write event in both traces. While the traditional Mazurkiewicz equivalence is control-centric, our new definition is data-centric. We show that our observation equivalence is coarser than the Mazurkiewicz equivalence, and in many cases even exponentially coarser. We devise a DPOR exploration of the trace space, called *data-centric* DPOR, based on the observation equivalence.

1. For acyclic architectures, our algorithm is guaranteed to explore *exactly* one representative trace from each observation class, while spending polynomial time per class. Hence, our algorithm is *optimal* wrt the observation equivalence, and in several cases explores exponentially fewer traces than *any* enumerative method based on the Mazurkiewicz equivalence.
2. For cyclic architectures, we consider an equivalence between traces which is finer than the observation equivalence; but coarser than the Mazurkiewicz equivalence, and in some cases is exponentially coarser. Our data-centric DPOR algorithm remains optimal under this trace equivalence.

Finally, we perform a basic experimental comparison between the existing Mazurkiewicz-based DPOR and our data-centric DPOR on a set of academic benchmarks. Our results show a significant reduction in both running time and the number of explored equivalence classes.

1. Introduction

Stateless model-checking of concurrent programs. The verification of concurrent programs is one of the major challenges in formal methods. Due to the combinatorial explosion on the number of interleavings, errors found by testing are hard to reproduce (often called *Heisenbugs* [31]), and the problem needs to be addressed by a systematic exploration of the state space. *Model checking* [9] addresses this issue, however, since model checkers store a large number of global states, it cannot be applied to realistic programs. One solution that is adopted is *stateless model checking* [15], which avoids the above problem by exploring the state space without explicitly storing the global states. This is typically achieved by a a scheduler, which drives the program execution based on the current interaction between the processes. Well-known tools such as VeriSoft [16, 17] and CHESS [27] have successfully employed stateless model checking.

Process p_1 :	Process p_2 :
1. write x;	1. write x;
2. read x;	2. read x;

Figure 1: A system of two processes with two events each.

Partial-Order Reduction (POR). Even though stateless model-checking addresses the global state space issue, it still suffers from the combinatorial explosion of the number of interleavings, which grows exponentially. While there are many approaches to reduce the number of explored interleavings, such as, depth-bounding and context bounding [22, 30], the most well-known method is *partial order reduction (POR)* [7, 15, 32]. The principle of POR is that two interleavings can be regarded as equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (independent) execution steps. The theoretical foundation of POR is the equivalence class of traces induced by the *Mazurkiewicz trace equivalence* [28], and POR explores at least one trace from each equivalence class. POR provides a full coverage of all behaviors that can occur in any interleaving, even though it explores only a subset of traces. Moreover, POR is sufficient for checking most of the interesting verification properties such as safety properties, race freedom, absence of global deadlocks, and absence of assertion violations [15].

Dynamic Partial-order Reduction (DPOR). Dynamic partial-order reduction (DPOR) [12] improves the precision of POR by recording actually occurring conflicts during the exploration and using this information on-the-fly. DPOR guarantees the exploration of at least one trace in each Mazurkiewicz equivalence class when the explored state space is acyclic and finite, which holds for stateless model checking, as usually the length of executions is bounded [12, 17, 31]. Recently, an optimal method for DPOR was developed [1].

A fundamental limitation. All existing approaches for DPOR are based on the Mazurkiewicz equivalence, i.e., they explore at least one (and possibly more) trace from each equivalence class. A basic and fundamental question is whether coarser equivalence classes than the Mazurkiewicz equivalence can be applied to stateless model checking and whether some DPOR-like approach can be developed based on such coarser equivalences. We address this fundamental question in this work. We start with a motivating example.

1.1 A minimal motivating example

Consider a concurrent system that consists of two processes and a single global variable x shown in Figure 1. Denote by w_i and r_i the write and read events to x by process p_i, respectively. The system consists of four events which are all pairwise dependent, except for the pair r_1, r_2. Two traces t and t' are called Mazurkiewicz equivalent, denoted $t \sim_M t'$, if they agree on the order of dependent events. The traditional DPOR based on the Mazurkiewicz equivalence $\sim_M$ will explore at least one representative trace from every class induced on the trace space by the Mazurkiewicz equivalence. There exist $\frac{2^3}{2} = 4$ possible orderings of dependent events, as there

are 2^3 possible interleavings, but half of those reorder the independent events r_1, r_2, and thus will not be considered. The traditional DPOR will explore the following four traces.

$$t_1 : w_1, r_1, w_2, r_2 \qquad t_2 : w_1, w_2, r_1, r_2$$
$$t_3 : w_2, w_1, r_1, r_2 \qquad t_4 : w_2, r_2, w_1, r_1$$

Note however that t_1 and t_4 are state-equivalent, in the sense that the local states visited by p_1 and p_2 are identical in the two traces. This is because each read event *observes* the same write event in t_1 and t_4. In contrast, in every pair of traces among t_1, t_2, t_3, there is at least one read event that observes a different write event in that pair. This observation makes it natural to consider two traces equivalent if they contain the same read events, and every read event observes the same write event in both traces. This example illustrates that it is possible to have coarser equivalence than the traditional Mazurkiewicz equivalence.

1.2 Our contributions

In this work our contributions are as follows.

Observation equivalence. We introduce a new notion of *observation equivalence* (Section 3.1), which is intuitively as follows: An observation function of a trace maps every read event to the write event it observes under sequentially consistency semantics. In contrast to every possible ordering of dependent control locations of Mazurkiewicz equivalence, in observation equivalence two traces are equivalent if they have the same observation function. The observation equivalence has the following properties.

1. *Soundness.* The observation equivalence is sufficient for exploring all local states of each process, and is thus sufficient for model checking wrt to local properties (similar to Mazurkieqicz equivalence).
2. *Coarser.* Second, we show that observation equivalence is coarser than Mazurkiewicz equivalence, i.e., if two traces are Mazurkiewicz equivalent, then they are also observation equivalent (Section 3.1).
3. *Exponentially coarser.* Third, we show that observation equivalence can be exponentially more succinct than Mazurkiewicz equivalence, i.e., we present examples where the ratio of the number of equivalence classes between observation and Mazurkiewicz equivalence is exponentially small (Section 3.2).

In summary, observation equivalence is a sound method which is always coarser, and in cases, strictly coarser than the fundamental Mazurkiewicz equivalence.

Principal difference. The principal difference between the Mazurkiewicz and our new observation equivalence is that while the Mazurkiewicz equivalence is *control-centric*, observation equivalence is *data-centric*. The data-centric approach takes into read-write and memory consistency restrictions as opposed to only event-dependency relation of Mazurkiewicz equivalence. Moreover, the data-centric approach allows analysis to be more 'white-box' and thus potentially more efficient.

Data-centric DPOR. We devise a DPOR exploration of the trace space, called *data-centric* DPOR, based on the observation equivalence. Our DPOR algorithm is based on a notion of *annotations*, which are intended observation functions (see Section 4). The basic computational problem is, given an annotation, decide whether there exists a trace which realizes the annotation. We show the computational problem is NP-complete in general, but for the important special case of *acyclic* architectures we present a polynomial-time (cubic-time) algorithm based on reduction to 2-SAT (details in Section 4). Our algorithm has the following implications.

1. For acyclic architectures, our algorithm is guaranteed to explore *exactly one* representative trace from each observation equivalence class, while spending polynomial time per class. Hence, our algorithm is *optimal* wrt the observation equivalence, and in several cases explores exponentially fewer traces than *any* enumerative method based on the Mazurkiewicz equivalence (details in Section 5).
2. For cyclic architectures, we consider an equivalence between traces which is finer than the observation equivalence; but coarser than the Mazurkiewicz equivalence, and in many cases is exponentially coarser. For this equivalence on traces, we again present an algorithm for DPOR that explore *exactly one* representative trace from each observation class, while spending polynomial time per class. Thus again our data-centric DPOR algorithm remains optimal under this trace equivalence for cyclic architectures (details in Section 6).

Experimental results. Finally, we perform a basic experimental comparison between the existing Mazurkiewicz-based DPOR and our data-centric DPOR on a set of academic benchmarks. Our results show a significant reduction in both running time and the number of explored traces (details in Section 7).

2. Preliminaries

In this section we introduce a simple model for concurrent programs that will be used for stating rigorously the key ideas of our data-centric DPOR. Similar (but syntactically richer) models have been used in [1, 12]. In Section 2.3 we discuss our various modeling choices and possible extensions.

Informal model. We consider a *concurrent system* of k processes under sequential consistency semantics. For the ease of presentation, we do not allow dynamic thread creation, i.e., k is fixed during any execution of the system. Each process is defined over a set of *local variables* specific to the process, and a set of *global variables*, which is common for all processes. Each process is represented as an acyclic *control-flow graph*, which results from unrolling the body of the process. A process consists of statements over the local and global variables, which we call *events*. The precise kind of such events is immaterial to our model, as we are only interested in the variables involved. In particular, in any such event we identify the local and global variables it involves, and distinguish between the variables that the event *reads* from and at most one variable that the event *writes* to. Such an event is *visible* if it involves global variables, and *invisible* otherwise. We consider that processes are *deterministic*, meaning that at any given time there is at most one event that each process can execute. Given the current state of the system, a scheduler chooses one process to execute a sequence of events that is invisibly maximal, that is, the sequence does not end while an invisible event from that process can be taken. The processes communicate by writing to and reading from the global variables. The system can exhibit nondeterministic behavior which is solely attributed to the scheduler, by choosing nondeterministically the next process to take an invisibly maximal sequence of events from any given state. We consider *locks* as the only synchronization primitive, with the available operations being acquiring a lock and releasing a lock. Since richer synchronization primitives are typically built using locks, this consideration is not restrictive, and helps with keeping the exposition of the key ideas simple.

2.1 Concurrent Computation Model

Here we present our model formally. Relevant notation is summarized in Table 1.

Relations and equivalence classes. A binary relation $\sim$ on a set X is an equivalence relation iff $\sim$ is reflexive, symmetric and

transitive. Given an equivalence $\sim_R$ and some $x \in X$, we denote by $[x]_R$ the equivalence class of x under $\sim_R$, i.e.,

$$[x]_R = \{y \in X : x \sim_R y\}$$

The *quotient set* $X/\sim_R := \{[x]_R \mid x \in X\}$ of X under $\sim_R$ is the set of all equivalence classes of X under $\sim_R$.

Notation on functions. We write $f : X \nrightarrow Y$ to denote that f is a partial function from X to Y. Given a (partial) function f, we denote by $\mathsf{dom}(f)$ and $\mathsf{img}(f)$ the domain and image set of f, respectively. For technical convenience, we think of a (partial) function f as a set of pairs $\{(x_i, y_i)\}_i$, meaning that $f(x_i) = y_i$ for all i, and use the shorthand notation $(x, y) \in f$ to indicate that $x \in \mathsf{dom}(f)$ and $f(x) = y$. Given (partial) functions f and g, we write $f \subseteq g$ if $\mathsf{dom}(f) \subseteq \mathsf{dom}(g)$ and for all $x \in \mathsf{dom}(f)$ we have $f(x) = g(x)$, and $f = g$ if $f \subseteq g$ and $g \subseteq f$. Finally, we write $f \subset g$ if $f \subseteq g$ and $f \neq g$.

Model syntax. We consider a *concurrent architecture* $\mathcal{P}$ that consists of a fixed number of *processes* $p_1, \ldots, p_k$, i.e., there is no dynamic thread creation. Each process p_i is defined over a set of n_i *local variables* $\mathcal{V}_i$, and a set of *global variables* $\mathcal{G}$, which is common for all processes. We distinguish a set of *lock variables* $\mathcal{L} \subseteq \mathcal{G}$ which are used for process synchronization. All variables are assumed to range over a finite domain $\mathcal{D}$. Every process p_i is represented as an acyclic control-flow graph CFG_i which results from unrolling all loops in the body of p_i. Every edge of CFG_i is labeled, and called an *event*. In particular, the architecture $\mathcal{P}$ is associated with a set of *events* $\mathcal{E}$, a set of *read events* (or *reads*) $\mathcal{R} \subseteq \mathcal{E}$, a set of *write events* (or *writes*) $\mathcal{W} \subseteq \mathcal{E}$, a set of *lock-acquire* events $\mathcal{L}^A \subseteq \mathcal{E}$ and a set of *lock-release* events $\mathcal{L}^R \subseteq \mathcal{E}$. The control-flow graph CFG_i of process p_i consists of events of the following types (where $\mathcal{V}_i = \{v_1, \ldots, v_{n_i}\}$, $g \in \mathcal{G}$, $l \in \mathcal{L}$ and $b : \mathcal{V}_i^{n_i} \to \{\mathsf{True}, \mathsf{False}\}$ is a boolean function on n_i arguments).

1. $e : v \leftarrow \mathsf{read}\ g$, in which case $e \in \mathcal{R}$,
2. $e : g \leftarrow \mathsf{write}\ f(v_1, \ldots, v_{n_i})$, in which case $e \in \mathcal{W}$,
3. $e : \mathsf{acquire}\ l$,
4. $e : \mathsf{release}\ l$,
5. $e_1 : b(v_1, \ldots, v_{n_i})$.

Each CFG_i is a directed acyclic graph with a distinguished *root* node r_i, such that there is a path $r_i \rightsquigarrow x$ to every other node x of CFG_i. Each node x of CFG_i has either

1. zero outgoing edges, or
2. one outgoing edge (x, y) labeled with an event of a type listed in Item 1-Item 4, or
3. $m \geqslant 2$ outgoing edges $(x, y_1), \ldots, (x, y_m)$ labeled with events $e_j : b_j(v_1, \ldots, v_{n_i})$ of Item 5, and such that for all values of $v_1, \ldots, v_{n_i}$, we have $b_j(v_1, \ldots, v_n) \implies \neg b_l(v_1, \ldots, v_{n_i})$ for all $j \neq l$.In this case, we call x a *branching* node.

For simplicity, we require that if x is a branching node, then for each edge (x, y) in CFG_i, the node y is not branching. Indeed, such edges can be easily contracted in a preprocessing phase. Figure 2 provides a summary of the model syntax. We let $\mathcal{E}_i \subseteq \mathcal{E}$ be the set of events that appear in CFG_i of process p_i, and similarly $\mathcal{R}_i \subseteq \mathcal{R}$ and $\mathcal{W}_i \subseteq \mathcal{W}$ the sets of read and write events of p_i. Additionally, we require that $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$ for all $i \neq j$ i.e., all $\mathcal{E}_i$ are pairwise disjoint, and denote by $\mathsf{proc}(e)$ the process of event e. The *location* of an event $\mathsf{loc}(e)$ is the unique global variable it involves. Given two events $e, e' \in \mathcal{E}_i$ for some p_i, we write $\mathsf{PS}(e, e')$ if there is a path $e \rightsquigarrow e'$ in CFG_i (i.e., we write $\mathsf{PS}(e, e')$ to denote that e is ordered before e' in the *program structure*).

We distinguish a set of *initialization events* $\mathcal{W}^{\mathcal{I}} \subseteq \mathcal{W}$ with $|\mathcal{W}^{\mathcal{I}}| = |\mathcal{G}|$ which are attributed to process p_1, and are used to initialize all the global variables to some fixed values. For every initialization write event $w^{\mathcal{I}}$ and for any event $e \in \mathcal{E}_i$ of process p_i, we define

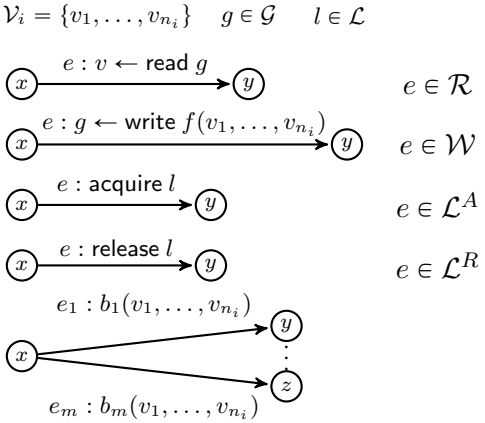

Figure 2: The control-flow graph CFG_i is a sequential composition of these five atomic graphs.

that $\mathsf{PS}(w^{\mathcal{I}}, e)$ (i.e., the initialization events occur before any event of each process). Figure 3 illustrates the above definitions on the typical bank account example.

Model semantics. A *local state* of a process p_i is a pair $s_i = (x_i, \mathsf{val}_i)$ where x_i is a node of CFG_i (i.e., the program counter) and val_i is a valuation on the local variables $\mathcal{V}_i$. A *global state* of $\mathcal{P}$ is a tuple $s = (\mathsf{val}, s_1, \ldots, s_k)$, where val is a valuation on the global variables $\mathcal{G}$ and s_i is a local state of process p_i. An event e along an edge (x, y) of a process p_i is *enabled* in s if $s_i = (x, \mathsf{val}_i)$ (i.e., the program counter is on node x) and additionally,

1. if $e : \mathsf{acquire}\ l$, then $\mathsf{val}(l) = \mathsf{False}$, and
2. if $e : b_j(v_1, \ldots, v_{n_i})$, then $b_j(\mathsf{val}_i(v_1), \ldots, \mathsf{val}_i(v_{n_i})) = \mathsf{True}$.

In words, if e acquires a lock l, then e is enabled iff l is free in s, and if x is a branching node, then e is enabled iff it respects the condition of the branch in s. Note that $\mathsf{release}\ l$ is always enabled, even if the lock is free. Given a state s, we denote by $\mathsf{enabled}(s) \subseteq \mathcal{E}$ the set of enabled events in s, and observe that there is at most one enabled event in each state s from each process. The execution of an enabled event e along an edge (x, y) of p_i in state $s = (\mathsf{val}, s_1, \ldots, s_k)$ results in a state $s' = (\mathsf{val}', s_1, \ldots, s_i', \ldots, s_k)$, where $s_i' = (y, \mathsf{val}_i')$. That is, the program counter of p_i has progressed to y, and the valuation functions val' and val_i' have been modified according to standard semantics, as follows:

1. $e : v \leftarrow \mathsf{read}\ g$ then $\mathsf{val}_i'(v) = \mathsf{val}(g)$,
2. $e : g \leftarrow \mathsf{write}\ f(v_1, \ldots, v_{n_i})$ then $\mathsf{val}'(g) = f(\mathsf{val}_i(v_1), \ldots, \mathsf{val}_i(v_{n_i}))$,
3. $e : \mathsf{acquire}\ l$ then $\mathsf{val}'(l) = \mathsf{True}$,
4. $e : \mathsf{release}\ l$ then $\mathsf{val}'(l) = \mathsf{False}$.

Moreover, val agrees with val' and val_i agrees with val_i' on all other variables. We write $s \xrightarrow{e} s'$ to denote that the execution of event e in s results in state s'. Let $\mathcal{S}_{\mathcal{P}}$ be the finite set (since variables range over a finite domain) of states of $\mathcal{P}$. The semantics of $\mathcal{P}$ are defined in terms of a transition system $\mathcal{A}_{\mathcal{P}} = (\mathcal{S}_{\mathcal{P}}, \Delta, s^0)$, where s^0 is the initial state, and $\Delta \subseteq \mathcal{S}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{P}}$ is the transition relation such that

$$(s, s') \in \mathcal{A}_{\mathcal{P}} \text{ iff } \exists e \in \mathsf{enabled}(s) : s \xrightarrow{e} s'$$

and either e is an initialization event, or the program counter of p_1 has passed all initialization edges of p_1. We write $s \xrightarrow{e_1, \ldots e_n} s'$ if there exists a sequence of states $\{s^i\}_{1 \leqslant i < n}$ such that

$$s \xrightarrow{e_1} s^1 \xrightarrow{e_2} \ldots s^{n-1} \xrightarrow{e_n} s'$$

```
Method: bool withdraw(int amount)
  Globals: int balance, lock l
  Locals : bool success, int v
  // 1. Try withdraw
1 success ← False
2 acquire(l)
3 v ← balance
4 if v − amount ⩾ 0 then
5 |   balance ← v − amount
6 |   success ← True
7 release(l)
8 print(success)
  // 2. Print balance
9 v ← balance
10 print(v)
```

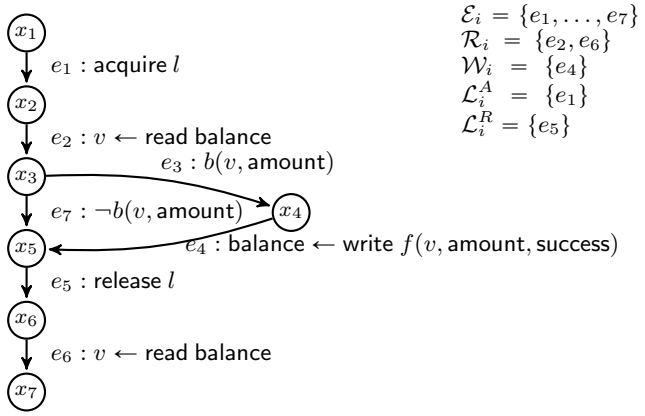

Figure 3: *(Left)*: A method withdraw executed whenever some amount is to be extracted from the balance of a bank account. *(Right)*: Representation of withdraw in our concurrent model. The root node is x_1. The program structure orders $\mathsf{PS}(e_2, e_4)$. We have $\mathsf{loc}(e_1) = \mathsf{loc}(e_5)$ and $\mathsf{loc}(e_2) = \mathsf{loc}(e_4) = \mathsf{loc}(e_7)$.

The initial state $s^0 = (\mathsf{val}, s_1^0, \ldots, s_k^0)$ is such that the value $\mathsf{val}(g)$ of each global variable g comes from the unique initialization write event w with $\mathsf{loc}(w) = g$, and for each $s_i^0 = (x_i, \mathsf{val}_i)$ we have that $x_i = r_i$ (i.e., the program counter of process p_i points to the root node of CFG_i). For simplicity we restrict $\mathcal{S}_\mathcal{P}$ to states s that are reachable from the initial state s^0 by a sequence of events $s^0 \xrightarrow{e_1,\ldots,e_n} s$. We focus our attention on state spaces $\mathcal{S}_\mathcal{P}$ that are acyclic.

Architecture topologies. The architecture $\mathcal{P}$ induces a labeled communication graph $G_\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, \lambda_\mathcal{P})$ where $V_\mathcal{P} = \{p_i\}_i$. There is an edge (p_i, p_j) if processes p_i, p_j accesses a common global variable or a common lock. The label $\lambda(p_i, p_j)$ is the set of all such global variables and locks. We call $\mathcal{P}$ *acyclic* if $G_\mathcal{P}$ does not contain cycles. The class of acyclic architectures includes, among others, all architectures with two processes, star architectures, pipelines, tree-like and hierarchical architectures.

Notation	Interpretation
$\mathcal{P} = (p_i)_{i=1}^k$	the concurrent architecture of k processes
$\mathcal{G}, \mathcal{V}, \mathcal{L}$	the global, local and lock variables
$\mathcal{E}, \mathcal{W}, \mathcal{R}, \mathcal{L}^A, \mathcal{L}^R, \mathcal{W}^\mathcal{I}$	the set of events, read, write, lock-acquire lock-release and initialization events
$\mathsf{val}_i, \mathsf{val}$	valuations of local, global variables
$\mathsf{enabled}(s) \subseteq \mathcal{E}$	the set of enabled events in s
$s \xrightarrow{e_1,\ldots,e_n} s'$	sequence of events from s to s'
$\mathsf{proc}(e), \mathsf{loc}(e)$	the process, the global variable of event e
CFG_i, $\mathsf{PS} \subseteq \mathcal{E} \times \mathcal{E}$	the control-flow graph of process p_i, and the program structure relation
$G_\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, \lambda_\mathcal{P})$	the communication graph of $\mathcal{P}$

Table 1: Notation on the concurrent architecture.

2.2 Traces

In this section we develop various helpful definitions on traces. Relevant notation is summarized in Table 2.

Notation on traces. A (concrete, concurrent) *trace* is a sequence of events $t = e_1, \ldots, e_j$ such that for all $1 \leqslant i < j$, we have $s^{i-1} \xrightarrow{e_i} s^i$, where $s^i \in \mathcal{S}_\mathcal{P}$ and s^0 is the initial state of $\mathcal{P}$. In such a case, we write succinctly $s^0 \xrightarrow{t} s^i$. We fix the first $|\mathcal{G}|$ events $e_1, \ldots, e_{|\mathcal{G}|}$ of each trace t to be initialization events that write the initial values to the global variables. That is, for all $1 \leqslant i \leqslant |\mathcal{G}|$ we have $e_i \in \mathcal{W}$, and hence every trace t starts with an *initialization trace* $t^\mathcal{I}$ as a prefix. Given a trace t, we denote by $\mathcal{E}(t)$ the set of events that appear in t, with $\mathcal{R}(t) = \mathcal{E}(t) \cap \mathcal{R}$ the read events in t, and with $\mathcal{W}(t) = \mathcal{E}(t) \cap \mathcal{W}$ the write events in t, and let $|t| = |\mathcal{E}(t)|$ be the *length* of t. For an event $e \in \mathcal{E}(t)$, we write $\mathsf{in}_t(e) \in \mathbb{N}^+$ to denote the index of e in t. Given some $\ell \in \mathbb{N}$, we denote by $t[\ell]$ the prefix of t up to position ℓ, and we say that t is an *extension* of $t[\ell]$. We let $\mathsf{enabled}(t)$ denote the set of enabled events in the state at the end of t, and call t *maximal* if $\mathsf{enabled}(t) = \varnothing$. We write $\mathcal{T}_\mathcal{P}$ for the set of all maximal traces of $\mathcal{P}$. We denote by $s(t)$ the unique state of $\mathcal{P}$ such that $s^0 \xrightarrow{t} s(t)$, and given an event $e \in \mathcal{R}(t) \cup \mathcal{W}(t)$, denote by $\mathsf{val}_t(e) \in \mathcal{D}$ the *value* that the unique global variable of e has in $s(t[\mathsf{in}_t(e)])$. We call a maximal trace t *lock-free* if the value of every lock variable in $s(t)$ is False (i.e., all locks have been released at the end of t). An event e is *inevitable* in a trace t if every every lock-free maximal extension of t contains e. 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. A sequence of events t' is called the *global projection* of another sequence t if $t' = t|(\mathcal{R} \cup \mathcal{W})$.

Sequential traces. Given a process p_i, a *sequential trace* τ_i is a sequence of events that correspond to a path in CFG_i, starting from the root node r_i. Note that a sequential trace is only wrt CFG_i, and is not necessarily a trace of the system. The notation on traces is extended naturally to sequential traces (e.g., $\mathcal{E}(\tau_i)$ and $\mathcal{R}(\tau_i)$ denote the events and read events of the sequential trace τ_i, respectively). Given k sequential traces $\tau_1, \tau_2, \ldots, \tau_k$, so that each τ_i is wrt p_i, we denote by

$$\tau_1 * \tau_2 * \ldots * \tau_k$$

the (possibly empty) set of all traces t such that $\mathcal{E}(t) = \bigcup_{1 \leqslant i \leqslant k} \mathcal{E}(\tau_i)$.

Conflicting events, dependent events and happens-before relations. Two events $e_1, e_2 \in \mathcal{R} \cup \mathcal{W}$ are said to *conflict*, written $\mathsf{Confl}(e_1, e_2)$ if $\mathsf{loc}(e_1) = \mathsf{loc}(e_2)$ and at least one is a write event. The events are said to be in *read-write conflict* if $e_1 \in \mathcal{R}$, $e_2 \in \mathcal{W}$ and $\mathsf{Confl}(e_1, e_2)$. Two events e_1, e_2 are said to be *independent* [12, 15] if

1. for each $i \in \{1, 2\}$ and pair of states s_1, s_2 such that $s_1 \xrightarrow{e_i} s_2$, we have that $e_{3-i} \in \mathsf{enabled}(s_1)$ iff $e_{3-i} \in \mathsf{enabled}(s_2)$, and

2. for any pair of states s_1, s_2 such that $e_1, e_2 \in \mathsf{enabled}(s_1)$, we have that $s_1 \xrightarrow{e_1,e_2} s_2$ iff $s_1 \xrightarrow{e_2,e_1} s_2$,

and *dependent* otherwise. Following the standard approach in the literature, we will consider two conflicting events to be always dependent [16, Chapter 3] (e.g., two conflicting write events are dependent, even if they write the same value). A sequence of events t induces a *happens-before* relation $\rightarrow_t \subseteq \mathcal{E}(t) \times \mathcal{E}(t)$, which is the smallest transitive relation on $\mathcal{E}(t)$ such that

$$e_1 \rightarrow_t e_2 \quad \text{if} \quad \mathsf{in}_t(e_1) \leqslant \mathsf{in}_t(e_2) \text{ and } e_1 \text{ and } e_2 \text{ are dependent.}$$

Observe that $\rightarrow_t$ orders all pairwise conflicting events, as well as all the events of any process.

Notation	Interpretation
t, τ_i	a trace and a sequential trace
$\mathsf{Confl}(e_1, e_2)$	conflicting events
$t[\ell], \lvert t\rvert$	the prefix up to index ℓ, and length of t
$\mathcal{E}(t), \mathcal{W}(t), \mathcal{R}(t)$	the events, write and read events of trace t
$\mathsf{in}_t(e)$, $\mathsf{val}_t(e)$	the index and value of event e in trace t
$t\lvert X$	projection of trace t on event set X
$\mathsf{enabled}(t)$	the enabled events in the state reached by t
$\rightarrow_t$	the happens-before relation on t
O_t	the observation function of t

Table 2: Notation on traces.

2.3 Discussion and Remarks

The concurrent model we consider here is minimalistic, to allow for a clear exposition of the ideas used in our data-centric DPOR. Here we discuss some of the simplifications we have adopted to keep the presentation simple.

Global variables. First, note that the location $\mathsf{loc}(e)$ of every event $e \in \mathcal{R} \cup \mathcal{W}$ is taken to be fixed in each CFG_i. The dynamic access of a static, global data structure g based on the value of a local variable v (e.g., accessing the element $g[v]$ of a global array g) can be modeled by using a different global variable g_i to encode the i-th location of g, and a sequence of branching nodes that determine which g_i should be accessed based on the value of v. Our framework can be strengthened to allow use of global arrays directly, and our algorithms apply straightforwardly to this richer framework. However, this would complicate the presentation, and is thus omitted in the theoretical exposition of the paper. A brief discussion on how arrays are handled directly is provided in the experimental results (Section 7), where we deal with arrays in the benchmark programs.

Invisible computations. Each process p_i is deterministic, and the only source of nondeterminism in the executions of the system comes from a nondeterministic scheduler that chooses an enabled event to be executed from a given state. The model uses the functions f and b on events $e : g \leftarrow \mathsf{write}\ f(v_1, \ldots, v_j)$ and $e : b(v_1, \ldots, b_n)$ respectively to collapse deterministic invisible computations of each process, and only consider the value that f writes on a global variable (in addition to the side-effects that f has on local the variables of process p_i). This is a standard approach in modeling concurrent systems, as interleaving invisible events does not change the set of reachable local states of the processes.

Locks and synchronization mechanisms. We treat lock-acquire and lock-release events as neither read, nor write events, since in every critical region protected by a pair of lock-acquire/lock-release events, the lock-release will always "observe" the preceding lock acquire. Hence lock events are independent in our model, and their reordering will occur naturally when needed (in contrast to the standard approach of [12]). Our approach can be extended to richer communication (e.g., message passing) and synchronization primitives (e.g. semaphores, wait-notify), which are often implemented using some low-level locking mechanism.

Maximal lock-free traces. We also assume that in every maximal trace of the system, every lock-acquire is followed by a corresponding lock-release. Traces without this property are typically considered erroneous, and some modern programming languages even force this restriction syntactically (e.g. in C#).

3. Observation Trace Equivalence

In this section, we introduce the observation equivalence $\sim_{\mathsf{O}}$ on traces, upon which in the later sections we develop our data-centric DPOR. We explore the relationship between the control-centric Mazurkiewicz equivalence $\sim_M$ and the observation equivalence. In particular, we show that $\sim_{\mathsf{O}}$ refines $\sim_M$, that is, every two traces that are equivalent under observations are also equivalent under reordering of independent events. We conclude by showing that $\sim_{\mathsf{O}}$ can be exponentially more succinct, both in the number of processes, and the size of each process.

3.1 Mazurkiewicz and Observation Equivalence

In this section we introduce our notion of observation equivalence. We start with the classical definition of Mazurkiewicz equivalence and then the notion of observation functions.

Mazurkiewicz trace equivalence. Two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$ are called *Mazurkiewicz equivalent* if one can be obtained from the other by swapping adjacent, independent events. Formally, we write $\sim_M$ for the Mazurkiewicz equivalence on $\mathcal{T}_{\mathcal{P}}$, and we have $t_1 \sim_M t_2$ iff

1. $\mathcal{E}(t_1) = \mathcal{E}(t_2)$, and
2. for every pair of events $e_1, e_2 \in \mathcal{E}(t_1)$ we have that $e_1 \rightarrow_{t_1} e_2$ iff $e_1 \rightarrow_{t_2} e_2$.

Observation functions. The concurrent model introduced in Section 2.1 follows *sequential consistency* [24], i.e., all processes observe the same order of events, and a read event of some variable will observe the value written by the last write event to that variable in this order. Throughout the paper, an *observation function* is going to be a partial function $\mathsf{O} : \mathcal{R} \twoheadrightarrow \mathcal{W}$. A trace t induces a total observation function $\mathsf{O}_t : \mathcal{R}(t) \rightarrow \mathcal{W}(t)$ following the sequential consistency axioms. That is, $\mathsf{O}_t(r) = w$ iff

1. $\mathsf{in}_t(w) < \mathsf{in}_t(r)$, and
2. for all $w' \in \mathcal{W}(t)$ such that $\mathsf{ConflRW}(r, w)$ we have that $\mathsf{in}_t(w') < \mathsf{in}_t(w)$ or $\mathsf{in}_t(w') > \mathsf{in}_t(r)$.

We say that t is *compatible* with an observation function O if $\mathsf{O} \subseteq \mathsf{O}_t$, and that t *realizes* O if $\mathsf{O} = \mathsf{O}_t$.

Observation equivalence. We define the *observation equivalence* $\sim_{\mathsf{O}}$ on the trace space $\mathcal{T}_{\mathcal{P}}$ as follows. For $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$ we have $t_1 \sim_{\mathsf{O}} t_2$ iff $\mathcal{E}(t_1) = \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1} = \mathsf{O}_{t_2}$, i.e., the two observation functions coincide.

We start with the following crucial lemma. In words, it states that if two traces agree on their observation functions, then they also agree on the values seen by their common read events.

Lemma 1. *Consider two traces t_1, t_2 such that $\mathsf{O}_{t_1} \subseteq \mathsf{O}_{t_2}$. Then (i) for all $r \in \mathcal{E}(t_1)$ we have that $\mathsf{val}_{t_1}(r) = \mathsf{val}_{t_2}(r)$, and (ii) $\mathcal{E}(t_1) \subseteq \mathcal{E}(t_2)$.*

Proof. Since the processes are deterministic, it suffices to argue about (i). The proof is by induction on the prefixes of t_1. We show inductively that for every $0 \leqslant \ell \leqslant |t_1|$, for all $r \in \mathcal{E}(t_1[\ell])$ we have that $\mathsf{val}_{t_1}(r) = \mathsf{val}_{t_2}(r)$. The claim is true for $\ell = 0$, since in that case $t_1[\ell] = \varepsilon$ and no read event appears in $t_1[\ell]$. Now assume that the claim holds for all prefixes $t_1[i]$ for $0 \leqslant i \leqslant \ell$, and let

$$r = \arg \min_{r' \in \mathcal{E}(t_1) \setminus \mathcal{E}(t_1[\ell])} \mathsf{in}_{t_1}(r')$$

be the next read in t_1 and $w = \mathsf{O}_{t_1}(r)$. Let $p = \mathsf{proc}(w)$, and since p is deterministic, using the induction hypothesis we see that $\mathsf{val}_{t_1}(w) = \mathsf{val}_{t_2}(w)$, and since $\mathsf{O}_{t_2}(r) = w$, we have that $\mathsf{val}_{t_2}(r) = \mathsf{val}_{t_2}(w) = \mathsf{val}_{t_1}(w) = \mathsf{val}_{t_1}(r)$, as desired. □

Soundness. Lemma 1 implies that in order to explore all local states of each process, it suffices to explore all observation functions realized by traces of $\mathcal{P}$.

The Mazurkiewicz trace equivalence is *control-centric*, i.e., equivalent traces share the same order between the dependent control locations of the program. In contrast, the observation trace equivalence is *data-centric*, as it is based on which write events are observed by the read events of each trace. Note that two conflicting events are dependent, and thus must be ordered in the same way by two Mazurkiewicz-equivalent traces. We establish the formal relationship between the two equivalences in the following theorem.

Theorem 1. *For any two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$, if $t_1 \sim_M t_2$ then $t_1 \sim_{\mathsf{O}} t_2$.*

Proof. Consider any read event $r \in \mathcal{R}(t_1)$ and assume towards contradiction that $\mathsf{O}_{t_1}(r) \neq \mathsf{O}_{t_2}(r)$. Let $w_1 = \mathsf{O}_{t_1}(r)$ and $w_2 = \mathsf{O}_{t_2}(r)$. Since $t_1 \sim_M t_2$, we have that $w_1 \in \mathcal{E}(t_2)$ and $w_2 \in \mathcal{E}(t_1)$. Then $w_1 \rightarrow_{t_1} r$ and $w_2 \rightarrow_{t_2} r$, and one of the following holds.

1. $r \rightarrow_{t_1} w_2$, and since $w_2 \rightarrow_{t_2} r$ then $t_1 \neq_{\sim_M} t_2$, a contradiction.
2. $w_2 \rightarrow_{t_1} w_1$, and since $t_1 \sim_M t_2$ we have that $w_2 \rightarrow_{t_2} w_1$, and thus $r \rightarrow_{t_2} w_1$. Since $w_1 \rightarrow_{t_1} r$, we have $t_1 \neq_{\sim_M} t_2$, a contradiction.

The desired result follows. □

Example 1. Figure 4 illustrates the difference between the Mazurkiewicz and observation trace equivalence on the example of Figure 3. Every execution of the system starts with an initialization trace $t^{\mathcal{I}}$ that initializes the lock l to False, and the initial value $\mathsf{desposit} = 4$. Consider that p_1 is executed with parameter $\mathsf{amount} = 1$ and p_2 is executed with parameter $\mathsf{amount} = 2$, (hence both withdrawals succeed). The primed events e_1', e_2' represent the system initialization.

- *(Left)*: The sequential trace of p_1, p_2.
- *(Center)*: Trace exploration using the Mazurkiewicz equivalence $\sim_M$. Solid lines represent the happens-before relation enforced by the program structure. Dashed lines represent potential happens-before relations between dependent events. A control-centric DPOR based on $\sim_M$ will resolve scheduling choices by exploring all possible realizable sets of the happens-before edges.
- *(Right)*: Trace exploration using the observation equivalence $\sim_{\mathsf{O}}$. Solid lines represent the happens-before relation enforced by the program structure. This time, dashed lines represent potential observation functions. Our data-centric DPOR based on $\sim_{\mathsf{O}}$ will resolve scheduling choices by exploring all possible realizable sets of the observation edges.

Both methods are guaranteed to visit all local states of each process. However, the data-centric DPOR achieves this by exploring potentially fewer scheduling choices.

3.2 Exponential succinctness

As we have already seen in the example of Figure 1, Theorem 1 does not hold in the other direction, i.e., $\sim_{\mathsf{O}}$ can be strictly coarser than $\sim_M$. Here we provide two simple examples in which $\sim_{\mathsf{O}}$ is exponentially more succinct than $\sim_M$. In the first example we examine a system of only two, identical processes, with n events each. In the second example we examine a system of k processes, with only two events each. Traditional enumerative model checking methods of concurrent systems are based on exploring *at least* (usually more than) one trace from every partition of the Mazurkiewicz equivalence using POR techniques that prune away equivalent traces (e.g. sleep sets [15], persistent sets [12], source sets and wakeup trees [1]). Such a search is *optimal* if it explores at most one trace from each class. Any optimal enumerative exploration based on the observation equivalence is guaranteed by Theorem 1 to examine no more traces than any enumerative exploration based on the Mazurkiewicz equivalence. The two examples show $\sim_{\mathsf{O}}$ can offer exponential improvements wrt two parameters: (i) the number of processes, and (ii) the size of each process.

Example 2 (Two processes of large size)**.** Consider the system $\mathcal{P}$ of $k = 2$ processes of Figure 5, and for $i \in \{1, \ldots, n\}, j \in \{1, 2\}$, denote by w_i^j (resp. r^j) the i-th write event (resp. the read event) of p_j. In any maximal trace, there are two ways to order the read events r^1, r^2, i.e., r^j occurs before r^{3-j} for the two choices of $j \in \{1, 2\}$. In any such ordering, r^{3-j} can only observe either w_{n-1}^{3-j} or w_{n-1}^j, whereas there are at most $n+1$ possible write events for r^j to observe (either w_n^j or one of the w_i^{3-j}). Hence $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}$ has size $O(n)$. In contrast, $\mathcal{T}_{\mathcal{P}}/\sim_M$ has size $\Omega(\binom{2 \cdot n}{n}) = \Omega(2^n)$, as there are $(2 \cdot n)!$ ways to order the $2 \cdot n$ write events of the two processes, but $n! \cdot n!$ orderings are invalid as they violate the program structure. Hence, even for only two processes, the observation equivalence reduces the number of partitions from exponential to linear.

Example 3 (Many processes of small size)**.** We now turn our attention to a system of $\mathcal{P}$ of k identical processes $p_1, \ldots, p_k$ with two events each, in Figure 6. There is only one global variable x, and each process performs a read and then a write to x. There are $O(k^k)$ realizable observation functions, by choosing for each one among k read events, one among k write events it can observe. Hence $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}$ has size $O(k^k)$. In contrast, the size of $\mathcal{T}_{\mathcal{P}}/\sim_M$ is $\Omega((k!)^2)$. This holds as there are $k!$ ways to order the k write events, and for each such permutation there $k!$ ways to assign each of the k read events to the write event that it observes. To see this second part, let $w_1, \ldots, w_k$ be any permutation of the write events, and let r_i be the read event in the same process as w_i. Then r_i can be placed right after any w_j with $i \leqslant j$, since w_i must happen before r_i, as forced by the program structure. Observe that $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}$ is exponentially more succinct than $\mathcal{T}_{\mathcal{P}}/\sim_M$, as

$$\frac{\Omega((k!)^2)}{O(k^k)} = \Omega\left(\frac{\prod_{i=1}^{k} i \cdot \lceil \frac{k}{i} \rceil}{k^k} \cdot \prod_{i=\lceil \frac{k}{2} \rceil + 1}^{k-1} i\right) = \Omega(2^k).$$

3.3 Solution Overview

Traditional DPOR algorithms exploit the Mazurkiewicz equivalence, and use various techniques such as persistent sets and sleep sets to explore each Mazurkiewicz class by few representative traces. Our goal is to develop an analogous DPOR that utilizes the observation equivalence, which by Theorem 1 is more succinct. In high level, our approach consists of the following steps.

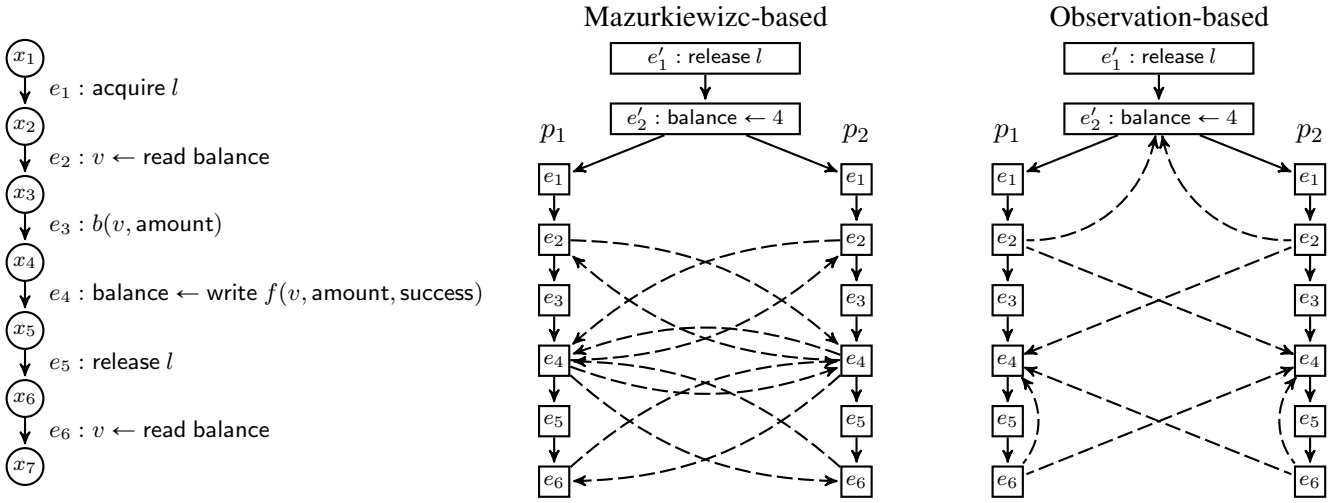

Figure 4: Trace exploration on the system of Figure 3 with two processes, where initially balance $\leftarrow 4$ and both withdrawals succeed.

Process p_1 :	Process p_2 :
1. write x	1. write x
2. write x	2. write x
...	...
$n+1$. read x	$n+1$. read x

Figure 5: An architecture of two processes with $n+1$ events each.

Process p_1 :	...	Process p_k :
1. write x		1. write x
2. read x		2. read x

Figure 6: An architecture of k processes with two events each.

1. In Section 4 we introduce the concept of annotations. An annotation is a function from read to write events, and serves as an intended observation function. Given an annotation, the goal is to obtain a trace whose observation function coincides with the annotation. We restrict our attention to a certain class of *well-formed* annotations, and show that although the problem is NP-complete in general, it admits a polynomial time (in fact, cubic in the size of the trace) solution in acyclic architectures.
2. In Section 5 we present our data-centric DPOR. Section 5.1 introduces the notion of causal past cones in a trace. The concept is similar to Lamport's *happens-before* relation [23], and is used to identify past events that may causally affect a current event in a trace. We note that this concept is different from the happens-before relation used in the Mazurkiewicz equivalence. We use the notions of annotations and causal cones to develop our algorithm, and prove its correctness and optimality (in Section 5.2).
3. In Section 6 we extend our algorithm to cyclic architectures.

Several technical proofs are relegated to Appendix A. Table 1 and Table 2 summarize relevant notation in the proofs.

4. Annotations

In this section we introduce the notion of *annotations*, which are intended constraints on the observation functions that traces discovered by our data-centric DPOR (DC-DPOR) are required to meet.

Annotations. An *annotation pair* $\mathsf{A} = (\mathsf{A}^+, \mathsf{A}^-)$ is a pair of

1. a *positive annotation* $\mathsf{A}^+ : \mathcal{R} \twoheadrightarrow \mathcal{W}$, and
2. a *negative annotation* $\mathsf{A}^- : \mathcal{R} \twoheadrightarrow 2^{\mathcal{W}}$

such that for all read events r, if $\mathsf{A}^+(r) = w$, then we have $\mathsf{ConflRW}(r, w)$ and it is not the case that $\mathsf{PS}(r, w)$. We will use annotations to guide the recursive calls of DC-DPOR towards traces that belong to different equivalence classes than the ones explored already, or will be explored by other branches of the algorithm. A positive annotation A^+ forces DC-DPOR to explore traces that are compatible with A^+ (or abort the search if no such trace can be generated). Since a positive annotation is an "intended" observation function, we say that a trace t *realizes* A^+ if $\mathsf{O}_t = \mathsf{A}^+$, in which case A^+ is called *realizable*. A negative annotation A^- prevents DC-DPOR from exploring traces t in which a read event observes a write event that belongs to its negative annotation set (i.e., $\mathsf{O}_t(r) \in \mathsf{A}^-(r)$). In the remaining section we focus on positive annotations, and the problem of deciding whether a positive annotation is realizable.

The value function $\mathsf{val}_{\mathsf{A}^+}$. Given a positive annotation A^+, we define the relation $<_{\mathsf{A}^+} \subseteq \mathsf{img}(\mathsf{A}^+) \times \mathsf{dom}(\mathsf{A}^+)$ such that $w <_{\mathsf{A}^+} r$ iff $(r, w) \in \mathsf{A}^+$. The positive annotation A^+ is *acyclic* if the relation $\mathsf{PS} \cup <_{\mathsf{A}^+}$ is a strict partial order (i.e., it contains no cycles). The *value function* $\mathsf{val}_{\mathsf{A}^+} : \mathsf{dom}(\mathsf{A}^+) \cup \mathsf{img}(\mathsf{A}^+) \to \mathcal{D}$ of an acyclic positive annotation A^+ is the unique function defined inductively, as follows.

1. For each $w \in \mathsf{img}(\mathsf{A}^+)$ of the form $w : g \leftarrow \mathsf{write}\ f(v_1, \ldots, v_{n_i})$, we have $\mathsf{val}_{\mathsf{A}^+}(w) = f(\alpha_1, \ldots, \alpha_{n_i})$, where for each α_j we have
   (a) $\alpha_j = \mathsf{val}_{\mathsf{A}^+}(r)$ if there exists a read event $r \in \mathsf{dom}(\mathsf{A}^+)$ such that (i) r is of the form $r : v_j \leftarrow \mathsf{read}\ g'$ and (ii) $\mathsf{PS}(r, w)$ and (iii) there exists no other $r' \in \mathsf{dom}(\mathsf{A}^+)$ with $\mathsf{PS}(r, r')$ and which satisfies conditions (i) and (ii).
   (b) α_j equals the initial value of v_i otherwise.
2. For each $r \in \mathsf{dom}(\mathsf{A}^+)$ we have $\mathsf{val}_{\mathsf{A}^+}(r) = \mathsf{val}_{\mathsf{A}^+}(\mathsf{A}^+(r))$.

Note that $\mathsf{val}_{\mathsf{A}^+}$ is well-defined, as for any read event r that is used to define the value of a write event w we have $\mathsf{PS}(r, w)$, and thus by the acyclicity of A^+, $\mathsf{val}_{\mathsf{A}^+}(r)$ does not depend on $\mathsf{val}_{\mathsf{A}^+}(w)$.

Remark 1. *If A^+ is realizable then it is acyclic, and for any trace t that realizes A^+ we have that $\mathsf{val}_t = \mathsf{val}_{\mathsf{A}^+}$.*

Well-formed annotations and basis of annotations. A positive annotation A^+ is called *well-formed* if it is acyclic, and there exist sequential traces $(\tau_i)_i$, one for each process p_i, such that each τ_i ends in a global event, the following conditions hold.

1. (a) for every lock-acquire event $e_a \in \mathcal{E}(\tau_i)$ there exists a lock-release event $e_r \in \mathcal{E}(t)$ such that $\mathsf{loc}(e_a) = \mathsf{loc}(e_r)$ and $\mathsf{in}_{\tau_i}(e_a) < \mathsf{in}_{\tau}(e_r)$, and
   (b) for every pair of lock-acquire events $e_a^1, e_a^2 \in \mathcal{E}(\tau_i) \cap \mathcal{L}^A$ such that $\mathsf{in}\tau_i(e_a^1) < \mathsf{in}_{\tau_i}(e_a^2)$ and $\mathsf{loc}(e_a^1) = \mathsf{loc}(e_a^2)$ there exists a lock release event $e_r \in \mathcal{E}(\tau_i) \cap \mathcal{L}^R$ such that $\mathsf{in}_{\tau_i}(e_a^1) < \mathsf{in}\tau_i(e_r) < \mathsf{in}\tau_i(e_a^2)$ and $\mathsf{loc}(e_r) = \mathsf{loc}(e_a^1) = \mathsf{loc}(e_a^2)$.
2. $\bigcup_i \mathcal{R}(\tau_i) = \mathsf{dom}(\mathsf{A}^+)$ and $\bigcup_i \mathcal{W}(\tau_i) \subseteq \mathsf{img}(\mathsf{A}^+)$, i.e., $(\tau_i)_i$ contains precisely the read events of A^+ and a superset of the write events.
3. Each τ_i corresponds to a deterministic computation of process p_i, where the value of every global event e during the computation is taken to be $\mathsf{val}_{\mathsf{A}^+}(e)$.

The sequential traces $(\tau_i)_i$ are called a *basis* of A^+ if every τ_i is minimal. The following lemma establishes properties of well-formedness and basis.

Lemma 2. *Let $X = \mathsf{dom}(\mathsf{A}^+) \cap \mathsf{img}(\mathsf{A}^+)$ be the set of events that appear in a positive annotation A^+, and $X_i = X \cap \mathcal{E}_i$ the subset of events of X from process p_i. The following assertions hold:*

1. *If A^+ is well-formed, then it has a unique basis $(\tau_i)_i$.*
2. *Computing the basis of A^+ (or concluding that A^+ is not well-formed) can be done in $O(n)$ time, where $n = \sum_i(|\tau_i|)$ if A^+ is well-formed, otherwise $n = \sum_i \ell_i$, where ℓ_i is the length of the longest path from the root r_i of CFG_i to an event $e \in X_i$.*
3. *For every trace t that realizes A^+ we have that A^+ is well-formed and $t \in \tau_1 * \ldots * \tau_k$.*

4.1 The Hardness of Realizing Positive Annotations

A core step in our data-centric DPOR algorithm is constructing a trace that realizes a positive annotation. That is, given a positive annotation A^+, the goal is to obtain a trace t (if one exists) such that $\mathsf{O}_t = \mathsf{A}^+$, i.e., t contains precisely the read events of A^+, and every read event in t observes the write event specified by A^+. Here, we show that the problem is NP-complete in the general case. Membership in NP is trivial, since, given a trace t, it is straightforward to verify that $\mathsf{O}_t = \mathsf{A}^+$ in $O(|t|)$ time. Hence our focus will be on establishing NP-hardness. For doing so, we introduce a new graph problem, namely ACYCLIC EDGE ADDITION, which is closely related to the problem of realizing a positive annotation under sequential consistency semantics. We first show that ACYCLIC EDGE ADDITION is NP-hard, and afterwards that the problem is polynomial-time reducible to realizing a positive annotation.

The problem ACYCLIC EDGE ADDITION. The input to the problem is a pair (G, H) where $G = (V, E)$ is a directed acyclic graph, and $H = \{(x_i, y_i, z_i)\}_i$ is a set of triplets of distinct nodes such that

1. $x_i, y_i, z_i \in V$, $(y_i, z_i) \in E$ and $(x_i, y_i), (z_i, x_i) \notin E$, and
2. each node x_i and y_i appears only once in H.

An *edge addition set* $X = \{e_i\}_{i=1}^{|H|}$ for (G, H) is a set of edges $e_i \in E$ such that for each e_i we have either $e_i = (x_i, y_i)$ or $e_i = (z_i, x_i)$. The problem ACYCLIC EDGE ADDITION asks whether there exists an edge addition set X for (G, H) such that the graph $G_X = (V, E \cup X)$ remains acyclic.

Lemma 3. *ACYCLIC EDGE ADDITION is NP-hard.*

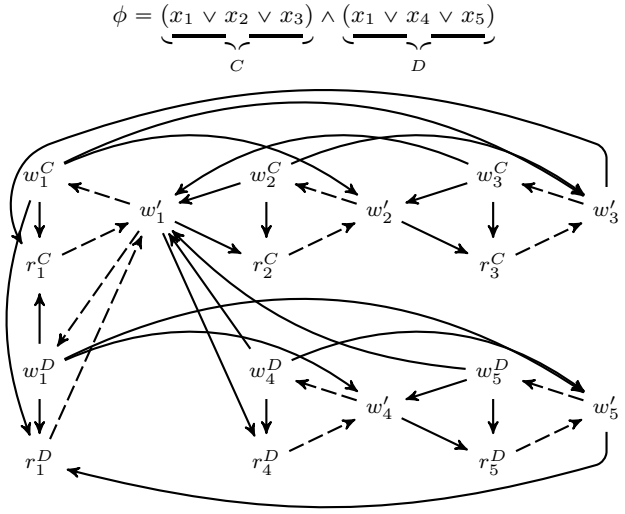

Figure 7: The reduction of 3SAT over ϕ to ACYCLIC EDGE ADDITION over (G, H). The nodes and solid edges represent the graph G, whereas the dashed edges represent the triplets in H.

Sketch. The proof is by reduction from MONOTONE ONE-IN-THREE SAT [14, LO4]. In MONOTONE ONE-IN-THREE SAT, the input is a propositional 3CNF formula ϕ in which every literal is positive, and the goal is to decide whether there exists a satisfying assignment for ϕ that assigns exactly one literal per clause to True. The reduction proceeds as follows. In the following, we let C and D range over the clauses and x_i over the variables of ϕ. We assume w.l.o.g. that no variable repeats in the same clause. For every variable x_i, we introduce a node $w_i' \in V$. For every clause $C = (x_{C_1} \vee x_{C_2} \vee x_{C_3})$, we introduce a pair of nodes $w_{C_j}^C, r_{C_j}^C \in V$ and an edge $(w_{C_j}^C, r_{C_j}^C) \in E$, where $j \in \{1, 2, 3\}$. Additionally, we introduce an edge $(w_{C_j}^C, w_{C_l}') \in E$ for every pair $j, l \in \{1, 2, 3\}$ such that $j \neq l$, and an edge $(w_{C_j}', r_{C_l}^C)$ for each $j \in \{1, 2, 3\}$, where $l = (j + 1) \mod 3 + 1$. Finally, for every pair of clauses C, D and $l_1, l_2 \in \{1, 2, 3\}$ such that $C_{l_1} = D_{l_2} = \ell$ (i.e., C and D share the same variable x_ℓ in positions l_1 and l_2), we add edges $(w_\ell^C, r_\ell^D), (w_\ell^D, r_\ell^C) \in E$. The set H consists of triplets of nodes $(w_{C_j}', w_{C_j}^C, r_{C_j}^C)$ for every clause C and $j \in \{1, 2, 3\}$. Figure 7 illustrates the construction. The detailed proof is in Appendix A.

From ACYCLIC EDGE ADDITION to annotations. Finally, we argue that ACYCLIC EDGE ADDITION is polynomial-time reducible to realizing a positive annotation. Given an instance (G, H) of ACYCLIC EDGE ADDITION, with $G = (V, E)$, we construct an architecture $\mathcal{P}$ of $k = 2 \cdot |H|$ processes $(p_i)_i$, and a positive annotation A^+. We assume, wlog, that the set of nodes V is precisely the set of nodes that appear in the triplets of H. Indeed, any other nodes can be removed while maintaining the connectivity between the nodes in the triplets of H, and any edge addition set X solves the problem in the original graph iff it does so in the reduced graph. The construction proceeds in two steps.

1. For every triplet $(x_i, y_i, z_i) \in H$, we create two events $w_i \in \mathcal{W}$, $r_i \in \mathcal{R}$ in p_i such that $\mathsf{PS}(w_i, r_i)$, and an event w_i' in process $p_{|H|+i}$. For all three events we set $\mathsf{loc}(r_i) = \mathsf{loc}(w_i) = \mathsf{loc}(w_i') = g_i$, where $g_i \in \mathcal{G}$ is some fresh global variable of $\mathcal{P}$. Finally, we introduce $(r_i, w_i) \in \mathsf{A}^+$. Given a node u, let $e(u)$ denote the event associated with u.
2. For every edge (u, v) we introduce a new global variable $g \in \mathcal{G}$, and two events $w_{u,v} \in \mathcal{W}$, $r_{u,v} \in \mathcal{R}$ such that $\mathsf{loc}(w_{u,v}) =$

$\mathsf{loc}(r_{u,v}) = g$. We make $w_{u,v}$ an event of the same process as $e(u)$, and $r_{u,v}$ an event of the same process as $e(v)$, and additionally $\mathsf{PS}(e(u), w_{u,v})$ and $\mathsf{PS}(r_{u,v}, e(v))$. Finally, we introduce $(r_{u,v}, w_{u,v}) \in \mathsf{A}^+$.

Observe that the above construction is linear in the size of (G, H). We refer to Appendix A for the formal proof of the reduction.

4.2 Realizing Positive Annotations in Acyclic Architectures

We now turn our attention to a tractable fragment of the positive annotation problem. Here we show that if $\mathcal{P}$ is an acyclic architecture, then the problem admits a polynomial-time solution (in fact, cubic in the size of the constructed trace).

Procedure Realize**.** Let $\mathcal{P}$ be an acyclic architecture, and A^+ a positive annotation over $\mathcal{P}$. We describe a procedure $\mathsf{Realize}(\mathsf{A}^+)$ which returns a trace t that realizes A^+, or $\bot$ if A^+ is not realizable. The procedure works in two phases. In the first phase, $\mathsf{Realize}(\mathsf{A}^+)$ uses Lemma 2 to extract a basis $(\tau_i)_i$ of A^+. In the second phase, $\mathsf{Realize}(\mathsf{A}^+)$ determines whether the events of $\bigcup_i \mathcal{E}(\tau_i)$ can be linearized in a trace t such that $\mathsf{O}_t = \mathsf{A}^+$. Informally, the second phase consists of constructing a 2SAT instance over variables x_{e_1,e_2}, where $e_1, e_2 \in \bigcup_i \mathcal{E}(\tau_i)$. Setting x_{e_1,e_2} to True corresponds to making e_1 happen before e_2 in the witness trace t. The clauses of the 2SAT instance capture four properties that each such ordering needs to meet, namely that

1. the resulting assignment produces a total order (totality, antisymmetry and transitivity) between all of the events that appear in adjacent processes in the communication graph $G_{\mathcal{P}}$,
2. the produced total order respects the positive annotation, i.e., every write event w' that conflicts with an annotated read/write pair $(r, w) \in \mathsf{A}^+$ must either happen before w or after r,
3. the produced total order respects the lock semantics, i.e., between every two lock-acquire events on the same lock there must be a lock-release event, and
4. the produced total order respects the partial order induced by the program structure PS and the positive annotation A^+.

The formal description of the second phase is given in Algorithm 1.

Lemma 4. *Given a well-formed positive annotation A^+ over a basis $(\tau_i)_i$,* Realize *constructs a trace t that realizes A^+ (or concludes that A^+ is not realizable) and requires $O(n^3)$ time, where $n = \sum_i |\tau_i|$.*

Proof. We present the correctness proof and complexity analysis.

Corrctness. We first argue about correctness.

1. If Realize returns a sequence of events t (Line 30) then clearly t is a trace since t respects the program structure PS (Line 24) and the lock semantics (Line 21). Additionally, t realizes A^+, as the sequential consistency axioms are satisfied because of Line 18 and Line 24.
2. If A^+ is realizable by a trace t, then t is a linearization of E^*, thus for every pair of distinct conflicting events $(e_1, e_2) \in E^*$ we have $\mathsf{in}_t(e_1) < \mathsf{in}_t(e_2)$ (Line 24). By the sequential consistency axioms, for every pair $(r, w) \in \mathsf{A}^+$ and $w' \neq w$ with $\mathsf{Confl}(r, w')$ we have either $\mathsf{in}_t(w') < \mathsf{in}_t(w)$ or $\mathsf{in}_t(r) < \mathsf{in}_t(w')$ (Line 18). Additionally, for every lock-acquire event e_a appearing in t, no other lock-acquire event e'_a that conflicts with e_a appears in t before a lock-release event e_r such that $p(e_a) = p(e_r)$ appears in t (Line 21). Finally, since t induces a total order on V, it is clearly transitive (Line 10) and antisymmetric (Line 7). Hence the set of 2SAT clauses $\mathcal{C}$ is satisfiable. It suffices to argue that $G' = (V', E')$ (Line 29) is acyclic, as then any topological order of G' will satisfy the sequential consistency axioms (Line 18). Assume towards contradiction otherwise. If G^* has a cycle, then A^+ is not realizable, as t must linearize E^*. Hence G^* must be acyclic, Thus, any cycle C in G' traverses an edge $(e_1, e_2) \in E' \backslash E^*$, hence $f(x_{e_1,e_2}) = \mathsf{True}$. We distinguish the following cases.
   (a) If there exists a cycle C which traverses a single edge $(e_1, e_2) \in E' \backslash E^*$, then $(e_2, e_1) \in E^*$, and $f(x_{e_2,e_1}) = \mathsf{True}$ (Line 24), and thus $f(x_{e_1,e_2}) = \mathsf{False}$ (Line 7), a contradiction.
   (b) Otherwise, let C be a simple cycle in G^* that traverses the fewest number of edges in $E' \backslash E^*$, and C must traverse at least two edges $(e_1, e_2), (e_3, e_4) \in E' \backslash E^*$. Observe that $\mathsf{proc}(e_1) \neq \mathsf{proc}(e_2)$ and $\mathsf{proc}(e_3) \neq \mathsf{proc}(e_4)$ as otherwise we would have $(e_2, e_1) \in E^*$ or $(e_4, e_3) \in E^*$, and there would exist a cycle that traverses a single edge from $E' \backslash E^*$ (namely, $e_1 \to e_2 \to e_1$ or $e_3 \to e_4 \to e_3$). Since $\mathcal{P}$ is acyclic, by construction (Line 6) $|\{\mathsf{proc}(e_1), \mathsf{proc}(e_2), \mathsf{proc}(e_3), \mathsf{proc}(e_4)\}| = 2$, i.e., there exist two processes p_i, p_j such that

   $e_1 \in \mathcal{E}_i$ and $e_2 \in \mathcal{E}_j$ and $e_3, e_4 \in \mathcal{E}_i \cup \mathcal{E}_j$ and $e_3, e_4 \notin \mathcal{E}_i \cap \mathcal{E}_j$

Algorithm 1: $\mathsf{Realize}(\mathsf{A}^+)$

Input: A positive annotation A^+ with basis $(\tau_i)_i$
Output: A trace t that realizes A^+ or $\bot$ if A^+ is not realizable

1 Construct a directed graph $G = (V, E)$ where
2 - $V = \bigcup_i \mathcal{E}(\tau_i)$, and
3 - $E = \{(e_1, e_2) : (e_2, e_1) \in \mathsf{A}^+ \text{ or } \mathsf{PS}(e_1, e_2)\}$
4 $G^* = (V, E^*) \leftarrow$ the transitive closure of G
// A set $\mathcal{C}$ of 2SAT clauses over variables $V_{\mathcal{C}}$
5 $\mathcal{C} \leftarrow \varnothing$
6 $V_{\mathcal{C}} \leftarrow \{x_{e_1,e_2} : e_1, e_2 \in V \text{ and } e_1 \neq e_2 \text{ and either } \mathsf{proc}(e_1) = \mathsf{proc}(e_2) \text{ or } (\mathsf{proc}(e_1), \mathsf{proc}(e_2)) \in E_{\mathcal{P}}\}$
// 1. Antisymmetry clauses
7 **foreach** $x_{e_1,e_2} \in V_{\mathcal{C}}$ **do**
8 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(x_{e_1,e_2} \Rightarrow \neg x_{e_2,e_1}), (\neg x_{e_2,e_1} \Rightarrow x_{e_1,e_2})\}$
9 **end**
// 2. Transitivity clauses
10 **foreach** $x_{e_1,e_2} \in V_{\mathcal{C}}$ **do**
11 $\quad$ **foreach** $(e_2, e_3) \in E^*$ **do**
12 $\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(x_{e_1,e_2} \Rightarrow x_{e_1,e_3})\}$
13 $\quad$ **end**
14 $\quad$ **foreach** $(e_3, e_1) \in E^*$ **do**
15 $\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(x_{e_1,e_2} \Rightarrow x_{e_3,e_2})\}$
16 $\quad$ **end**
17 **end**
// 3. Annotation clauses
18 **foreach** $(r, w) \in \mathsf{A}^+$ *and* $w' \in V \cap \mathcal{W}$ *s.t.* $\mathsf{ConflRW}(r, w')$ **do**
19 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(x_{w',r} \Rightarrow x_{w',w}), (x_{w,w'} \Rightarrow x_{r,w'})\}$
20 **end**
// 4. Lock clauses
21 **foreach** $e_a, e'_a \in V \cap \mathcal{L}^A$ *and* $e_r \in V \cap \mathcal{L}^R$ *s.t.* $\mathsf{loc}(e_a) = \mathsf{loc}(e'_a)$ *and* $\mathsf{loc}(e_a) = \mathsf{loc}(e_r)$ *and* $\mathsf{PS}(e_a, e_r)$ *and* $\nexists e''_a \in V \cap \mathcal{L}^A$ *s.t.* $\mathsf{loc}(e''_a) = \mathsf{loc}(e_r)$ *and* $\mathsf{PS}(e''_a, e_r)$ **do**
22 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(x_{e'_a,e_r} \Rightarrow x_{e'_a,e_a}), (x_{e_a,e'_a} \Rightarrow x_{e_r,e'_a})\}$
23 **end**
// 5. Fact clauses
24 **foreach** $(e_1, e_2) \in E^*$ **do**
25 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(x_{e_1,e_2})\}$
26 **end**
27 Compute a satisfying assignment $f : V_{\mathcal{C}} \to \{\mathsf{False}, \mathsf{True}\}^{|V_{\mathcal{C}}|}$ of the 2SAT over $\mathcal{C}$, or **return** $\bot$ if $\mathcal{C}$ is unsatisfiable
28 $E' \leftarrow E \cup \{(e_1, e_2) : f(x_{e_1,e_2}) = \mathsf{True}\}$
29 Let $G' = (V, E')$
30 **return** a trace t by topologically sorting the vertices of G'

Then C traverses an edge (e'_3, e'_4) such that either $e'_3 = e_1$ or $\mathsf{PS}(e'_3, e_1)$, and either $e_2 = e'_4$ or $\mathsf{PS}(e_2, e'_4)$. In all cases, we have $(e'_3, e_1), (e_2, e'_4) \in E^*$. Since $(e'_3, e'_4) \in E'$ we have $f(x_{e'_3,e'_4}) = \mathsf{True}$, and by transitivity (Line 10), we have that $f(x_{e_2,e_1}) = \mathsf{True}$, a contradiction.

Complexity. The transitive closure requires $O(n^3)$ time, since $|V| = n$. The set $V_{\mathcal{C}}$ (Line 6) has $O(n^2)$ variables and each of the loops for constructing clauses iterates over triplets of nodes, hence the 2SAT instance is constructed in $O(n^3)$ time [5]. Computing a satisfying assignment for $\mathcal{C}$ (or concluding that none exists) requires linear time in $|\mathcal{C}|$, hence this step costs $O(n^3)$. Finally, constructing G' and computing a topological sorting of its vertices requires $O(n^2)$ time in total. The desired result follows. □

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

Theorem 2. *Consider any architecture $\mathcal{P} = (p)_i$ and let A^+ be any well-formed positive annotation over a basis $(\tau)_i$. Deciding whether A^+ is realizable is NP-complete. If $\mathcal{P}$ is acyclic, the problem can be solved in $O(n^3)$ time, where $n = \sum_i |\tau_i|$.*

5. Data-centric Dynamic Partial Order Reduction

In this section we develop our data-centric DPOR algorithm called DC-DPOR and prove its correctness and compactness, namely that that the algorithm explores each observation equivalence class of $\mathcal{T}_{\mathcal{P}}$ *once*. We start with the notion of causal past cones, which will help in proving the properties of our algorithm.

5.1 Causal Cones

Intuitively, the causal past cone of an event e appearing in a trace t is the set of events that precede e in t and may be responsible for enabling e in t.

Causal cones. Given a trace t and some event $e \in \mathcal{E}(t)$, the *causal past cone* $\mathsf{Past}_t(e)$ of e in t is the smallest set that contains the following events:

1. if there is an event $e' \in \mathcal{E}(t)$ with $\mathsf{PS}(e', e)$, then $e' \in \mathsf{Past}_t(e)$,
2. if $e_1 \in \mathsf{Past}_t(e)$, for every event $e_2 \in \mathcal{E}(t)$ such that $\mathsf{PS}(e_2, e_1)$, we have that $e_2 \in \mathsf{Past}_t(e)$, and
3. if there exists a read $r \in \mathsf{Past}_t(e) \cap \mathcal{R}$, we have that $\mathsf{O}_t(r) \in \mathsf{Past}_t(e)$.

In words, the causal past cone of e in t is the set of events e' that precede e in t and may causally affect the enabling of e in t. Note that for every event $e' \in \mathsf{Past}_t(e)$ we have that $e' \rightarrow_t e$, i.e., every event in the causal past cone of e also happens before e in t. However, the inverse is not true in general, as e.g. for some read r we have $\mathsf{O}_t(r) \rightarrow_t r$ but possibly $\mathsf{O}_t(r) \notin \mathsf{Past}_t(r)$.

Remark 2. *If $e' \in \mathsf{Past}_t(e)$, then $e' \rightarrow_t e$ and $\mathsf{Past}_t(e') \subseteq \mathsf{Past}_t(e)$.*

The following lemma states the main property of causal past cones used throughout the paper. Intuitively, if the causal past of an event e in some t_1 also appears in another trace t_2, and the read events in the causal past observe the same write events, then e is inevitable in t_2, i.e., every maximal extension of t_2 will contain e.

Lemma 5. *Consider two traces t_1, t_2 and an event $e \in \mathcal{E}(t_1)$ such that for every read $r \in \mathsf{Past}_{t_1}(e)$ we have $r \in \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1}(r) = \mathsf{O}_{t_2}(r)$. Then e is inevitable in t_2.*

Proof. We argue that every event $e' \in \mathsf{Past}_{t_1}(e) \cup \{e\}$ is inevitable in t_2. Let t'_2 be any lock-free maximal extension of t_2. Assume towards contradiction that $(\mathsf{Past}_{t_1}(e) \cup \{e\}) \backslash \mathcal{E}(t'_2) \neq \emptyset$, and let

$$e_m = \arg \min_{e' \in (\mathsf{Past}_{t_1}(e) \cup \{e\}) \backslash \mathcal{E}(t'_2)} \mathsf{in}_{t_1}(e')$$

be the first such event in t_1, and let $p_i \mathsf{proc}(e_m)$ of e_m. By Remark 2, for every event $e' \in \mathsf{Past}_{t_1}(e_m)$ we have $e' \in \mathsf{Past}_{t_1}(e)$, and since $\mathsf{in}_{t_1}(e') < \mathsf{in}_{t_1}(e_m)$, we have $e' \in \mathcal{E}(t'_2)$. Let (x, y) be the edge of CFG_i labeled with e_m. Since $e' \in \mathcal{E}(t'_2)$, the program counter of p_i becomes x at some point in t'_2. We examine the number of outgoing edges from node x.

1. If x has one outgoing edge, we distinguish whether e_m is a lock-acquire event or not.
   (a) If e_m is not a lock-acquire event, then e_m is always enabled after e' in t'_2, hence t'_2 is not maximal, a contradiction.
   (b) If $e_m : \mathsf{acquire}\ l$, since t'_2 is a lock-free trace, l is released in $s(t'_2)$, hence $e_m \in \mathsf{enabled}(t'_2)$ and t'_2 is not maximal, a contradiction.
2. If x has at least two outgoing edges then it is either $e_m : v$ or $e_m : b_j(v_1, \ldots, v_{n_i})$, where $v_i \in \mathcal{V}_i$ are local variables of p_i. Since for every read $r \in \mathsf{Past}_{t_1}(e_m)$ we have $r \in \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1}(r) = \mathsf{O}_{t_2}(r)$, by Remark 2, the same holds for reads $r \in \mathsf{Past}_{t_1}(e_m)$, i.e., for every read $r \in \mathsf{Past}_{t_1}(e_m)$ we have $r \in \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1}(r) = \mathsf{O}_{t_2}(r)$. By Lemma 1, we have $\mathsf{val}_{t_1}(r) = \mathsf{val}_{t_2}(r)$ for every such read r, and since p_i is deterministic, the value of v on x is a function of those reads, and thus each v_i has the same value when the program counter of p_i reaches node x in t_1 and t'_2. Since e_m appears in t_1, e_m is always enabled after e' in t'_2, hence t'_2 is not maximal, a contradiction.

The desired result follows. □

5.2 Data-centric Dynamic Partial Order Reduction

Algorithm DC-DPOR**.** We now present our data-centric DPOR algorithm. The algorithm receives as input a maximal trace t and annotation pair $\mathsf{A} = (\mathsf{A}^+, \mathsf{A}^-)$, where t is compatible with A^+. The algorithm scans t to detect conflicting read-write pairs of events that are not annotated, i.e, a read even $r \in \mathcal{R}(t)$ and a write event $w \in \mathcal{W}(t)$ such that $r \notin \mathsf{dom}(\mathsf{A}^+)$ and $\mathsf{ConflRW}(r, w)$. If $w \notin \mathsf{A}^-(r)$, then DC-DPOR will try to *mutate* r to w, i.e., the algorithm will push (r, w) in the positive annotation A^+ and call Realize to obtain a trace that realizes the new positive annotation. If the recursive call succeeds, then the algorithm will push w to the negative annotation of r, i.e., will insert w to $\mathsf{A}^-(r)$. This will prevent recursive calls from pushing (r, w) into their positive annotation. Algorithm 2 provides a formal description of DC-DPOR. Initially DC-DPOR is executed on input (t, A) where $t = \varepsilon$ is an empty trace, and $\mathsf{A} = (\emptyset, \emptyset)$ is a pair of empty annotations.

We say that DC-DPOR *explores* a class of $\mathcal{T}_{\mathcal{P}}/\ \sim_{\mathsf{O}}$ when it is called on some annotation input $\mathsf{A} = (\mathsf{A}^+, \mathsf{A}^-)$, where A^+ is realized by some (and hence, every) trace in that class. The representative trace is then the trace t' returned by Realize. The following two lemmas show the optimality of DC-DPOR, namely that the algorithm explores every such class at most once (*compactness*) and at least once (*completeness*). They both rely on the use of annotations, and the correctness of the procedure Realize (Lemma 4). We first state the compactness property, which follows by the use of negative annotations.

Lemma 6 (Compactness)**.** *Consider any two executions of* DC-DPOR *on inputs (t_1, A_1) and (t_2, A_2). Then $\mathsf{A}_1^+ \neq \mathsf{A}_2^+$.*

We now turn our attention to completeness, namely that every realizable observation function is realized by a trace explored by DC-DPOR. The proof shows inductively that if t is a trace that realizes an observation function O, then DC-DPOR will explore

Algorithm 2: DC-DPOR(t, A)

Input: A maximal trace t, an annotation pair $\mathsf{A} = (\mathsf{A}^+, \mathsf{A}^-)$

```
// Iterate over reads not yet mutated
foreach r ∈ E(t)\dom(A+) in increasing index in_t(r) do
    // Find conflicting writes allowed by A−
    foreach w ∈ E(t) s.t. Confl(r, w) and w ∉ A−(r) do
        A+_{r,w} ← A+ ∪ {(r, w)}
        // Attempt mutation and update A−
        Let t' ← Realize(A+_{r,w})
        if t' ≠ ⊥ then
            t'' ← a maximal extension of t'
            A−(r) ← A−(r) ∪ {w}
            A_{r,w} ← (A+_{r,w}, A−)
            Call DC-DPOR(t'', A_{r,w})
    end
end
```

a trace t_i that agrees with t on the first few read events. The, Lemma 5 guarantees that the first read event r on which the two traces disagree appears in t_i, and so does the write event w that r observes in O. Hence DC-DPOR either will mutate $r \to w$ (if $w \notin \mathsf{A}^-(r)$), or it has already done so in some earlier steps of the recursion (if $w \in \mathsf{A}^-(r)$).

Lemma 7 (Completeness)**.** *For every realizable observation function* O*,* DC-DPOR *generates a trace* t *that realizes* O*.*

See Section 2 for the formal proofs. We thus arrive to the following theorem.

Theorem 3. *Consider a concurrent acyclic architecture* $\mathcal{P}$ *of processes on an acyclic state space, and* $n = \max_{t \in \mathcal{T}_\mathcal{P}} |t|$ *the maximum length of a trace of* $\mathcal{P}$*. The algorithm* DC-DPOR *explores each class of* $\mathcal{T}_\mathcal{P}/\ \sim_\mathsf{O}$ *exactly once, and requires* $O\left(|\mathcal{T}_\mathcal{P}/\sim_\mathsf{O}| \cdot n^5\right)$ *time.*

We note that our main goal is to explore the exponentially large $\mathcal{T}_\mathcal{P}/\sim_\mathsf{O}$ by spending polynomial time in each class. The n^5 factor in the bound comes from a crude complexity analysis (also see Section 7 for some optimizations over this n^5 bound).

6. Beyond Acyclic Architectures

In the current section we turn our attention to cyclic architectures. Recall that according to Theorem 2, procedure Realize is guaranteed to find a trace that realizes a positive annotation A^+, provided that the underlying architecture is acyclic.

Architecture acyclic reduction. Consider a cyclic architecture $\mathcal{P}$, and the corresponding communication graph $G_\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, \lambda_\mathcal{P})$. We call a set of edges $X \subseteq E_\mathcal{P}$ an *all-but-two cycle set* of $G_\mathcal{P}$ if every cycle of $G_\mathcal{P}$ contains at most two edges outside of X. Given an all-but-two cycle set, $X \subseteq E_\mathcal{P}$ we construct a second architecture $\mathcal{P}^X$, called the *acyclic reduction* of $\mathcal{P}$ over X, by means of the following process.

1. Let $Y = \bigcup_{(p_i,p_j)\in X} \lambda_\mathcal{P}(p_i, p_j)$ be the set of variables that appear in the removed edges X. We introduce a set of *observable locks* $\mathcal{L}^\mathcal{O}$ in $\mathcal{P}^X$ such that we have exactly one observable lock $l_g \in \mathcal{L}^\mathcal{O}$ for each variable $g \in Y$. For every lock-acquire event e_a with $\mathsf{loc}(e_a) \in \mathcal{L}^\mathcal{O}$ we have $e_a \in \mathcal{R}$. Similarly, for every lock-release event e_r with $\mathsf{loc}(e_r) \in \mathcal{L}^\mathcal{O}$ we have $e_r \in \mathcal{W}$. That is, lock-acquire and lock-release events on observable locks are *read* and *write* events, respectively (in contrast with the traditional locks which are neither write nor read events).
2. For every process p_i, every write event $w \in \mathcal{W}_i$ with $\mathsf{loc}(w) \in Y$ is surrounded by an acquire/release pair on the observable lock variable $l_{\mathsf{loc}(w)}$. Similarly, every lock-acquire event $e \in \mathcal{L}^A$ with $\mathsf{loc}(e) \in Y$ is surrounded by an acquire/release pair on the observable lock variable $l_{\mathsf{loc}(e)}$.

Observation equivalence refined by an edge set. Consider a cyclic architecture $\mathcal{P}$ and X an edge set of the underlying communication graph $G_\mathcal{P}$. We define a new equivalence on the trace space $\mathcal{T}_\mathcal{P}$ as follows. Two traces $t_1, t_2 \in \mathcal{T}_\mathcal{P}$ are *observationally equivalent refined by* X, denoted by $\sim_\mathsf{O}^X$, if the following hold:

1. $t_1 \sim_\mathsf{O} t_2$, and
2. for every edge $(p_i, p_j) \in X$, for every pair of distinct write events $w_1, w_2 \in \mathcal{W}(t_1) \cap (\mathcal{W}_i \cup \mathcal{W}_j)$ with $\mathsf{loc}(w_1) = \mathsf{loc}(w_2) = g$ and $g \in \lambda_\mathcal{P}(p_i, p_j)$, we have that $\mathsf{in}_{t_1}(w_1) < \mathsf{in}_{t_1}(w_2)$ iff $\mathsf{in}_{t_2}(w_1) < \mathsf{in}_{t_2}(w_2)$

Clearly, $\sim_\mathsf{O}^X$ refines the observation equivalence $\sim_\mathsf{O}$. The following lemma captures that the Mazurkiewicz equivalence refines the observation equivalence refined by an edge set X.

Lemma 8. *For any two traces* $t_1, t_2 \in \mathcal{T}_\mathcal{P}$*, if* $t_1 \sim_M t_2$ *then* $t_1 \sim_\mathsf{O}^X t_2$*.*

Exponential succinctness. Similar to $\sim_\mathsf{O}$, we present examples (in Appendix A.3) of cyclic architectures where the equivalence $\sim_\mathsf{O}^X$ is exponentially more succinct than $\sim_M$, since in general it considers fewer reorderings of events that access variables of the edges of $E_\mathcal{P}\backslash X$, than the Mazurkiewicz reorderings on those events.

Data-centric DPOR on a cyclic architecture. We are now ready to outline the steps of the data-centric DPOR algorithm on a cyclic architecture $\mathcal{P}$, called DC-DPOR-Cyclic. First, we determine an all-but-two cycle set X of the underlying communication graph $G_\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, \lambda_\mathcal{P})$, and construct the acyclic reduction $\mathcal{P}^X$ of $\mathcal{P}$ over X. Then, we execute DC-DPOR on $\mathcal{P}^X$, with the following two modifications on the procedure Realize.

1. After the transitive closure graph G^* has been computed (Line 4), if G^* contains a cycle, or there exists a pair $(r, w) \in \mathsf{A}^+$ and a $w' \in V \cap \mathcal{W}$ with $\mathsf{ConflRW}(r, w')$ and such that $(w, w') \in E^*$ and $(w', r) \in E^*$, the procedure returns $\bot$.
2. In Line 6 we use the edge set $E_{\mathcal{P}^X}\backslash X$. Hence for every variable x_{e_1,e_2} used in the 2SAT reduction, we have either $\mathsf{proc}(e_1) = \mathsf{proc}(e_2)$ or $(\mathsf{proc}(e_1), \mathsf{proc}(e_2)) \in E_{\mathcal{P}^X}\backslash X$.

We arrive to the following theorem. We refer to Appendix A for the formal proofs of correctness and complexity.

Theorem 4. *Consider a concurrent architecture* $\mathcal{P}$ *of processes on an acyclic state space, and* $n = \max_{t\in\mathcal{T}_\mathcal{P}} |t|$ *the maximum length of a trace of* $\mathcal{P}$*. Let* X *be an all-but-two cycle set of the communication graph* $G_\mathcal{P}$*. The algorithm* DC-DPOR-Cyclic *explores each class of* $\mathcal{T}_\mathcal{P}/\ \sim_\mathsf{O}^X$ *exactly once, and requires* $O\left(|\mathcal{T}_\mathcal{P}/\sim_\mathsf{O}^X| \cdot n^5\right)$ *time.*

7. Experiments

We report on a basic experimental evaluation of our data-centric DPOR, and compare it to the standard DPOR of [12].

Experimental setup. We have implemented our DC-DPOR algorithm in Python, and evaluate its performance on a set of academic benchmarks (the details of the benchmarks can be found in Appendix B). Our basis for comparison is also a Python implementation of [12, Figure 4]. All benchmarks are straightforwardly translated to our model syntax. The experiments were run on a standard desktop computer with 3.5GHz CPU.

Handling static arrays. The challenge in handling arrays (and other data structures) lies in the difficulty of determining whether two global events access the same location of the array (and thus are in conflict) or not. Indeed, this is not evident from the CFG of each process, but depends on the values of indexing variables (e.g. the value of local variable i in an access to $\mathsf{table}[i]$). DPOR methods offer increased precision, as during the exploration of the trace space, backtracking points are computed dynamically, given a trace, where the value of indexing variables is known. In our case, the value of indexing variables is also needed when procedure Realize is invoked to construct a trace which realizes a positive annotation A^+. Observe that the values of all such variables are determined by the value function $\mathsf{val}_{\mathsf{A}^+}$, and thus in every sequential trace τ_i of the basis $(\tau_i)_i$ of A^+ these values are also known. In our Eratosthenes benchmark, arrays are thus handled this way.

Optimizations. Since our focus is on demonstrating a new, data-centric principle of DPOR, we focused on a basic implementation and avoided engineering optimizations. We outline two straightforward algorithmic optimizations which were simple and useful.

1. (*Burst mutations*). Instead of performing one mutation at a time, the algorithm performs a sequence of several mutations at once. In particular, given a trace t, any time we want to add a pair (r, w) to the positive annotation, we also add $(r', \mathsf{O}_t(r'))$, where $r' \in \mathsf{Past}_t(r) \cup \mathsf{Past}_t(w)$ ranges over all read events in the causal past of r and w in t. This makes the recursion tree shallower, as now we do not need to apply any mutation (r, w), where $w = \mathsf{O}_t(r)$, individually.
2. (*Cycle detection*). As a preprocessing step, before executing procedure Realize on some positive annotation A^+ input, we test whether the graph G (in Line 1) already contains a cycle. The existence of a cycle is a proof that A^+ is not realizable, and requires linear instead of cubic time, as the graph is sparse.

Results. We report the results of our evaluation in Table 3. Each section of the table starts with the benchmark name, the parameter that was used in order to obtain the results for the specific benchmark (e.g., number loop iterations, number of threads) and the time bound that was used for timeouts. We varied the timeout bound in order to obtain enough points for a comparison. Timeout cases are reported with "-". The first column of the table reports the parameter values that were used to obtain the respective row. The values of the corresponding table columns were calculated as follows.

1. $|\ \sim_M\ |$ reports an underapproximation of number of Mazurkiewicz classes, wrt *not necessarily maximal* traces. See Appendix B.1 for details on this underapproximation.
2. DC-DPOR (# A^+) reports the number of positive annotations (i.e., observation classes, not necessarily maximal) explored by DC-DPOR.
3. DPOR (s) and DC-DPOR (s) report the running time (in seconds) of the corresponding method.

We see that running times scale better in our DC-DPOR than the standard DPOR. Additionally, the number of observation classes examined by DC-DPOR scales better than (the underapproximation of) $|\sim_M|$.

#	$\lvert\sim_M\rvert$	DC-DPOR (# A^+)	DPOR (s)	DC-DPOR (s)
(2 Processes) **Alternate write-read** (n) $\mathsf{TO} = 5\mathsf{h}$				
(4)	8335	1107	19	5
(5)	87260	8953	327	63
(6)	931090	73789	5514	790
(7)	-	616227	-	10015
(k Processes) **Pipeline** (k) $\mathsf{TO} = 4\mathsf{h}$				
(9)	163840	6561	816	43
(10)	655360	19683	3170	156
(11)	2621440	59049	14561	544
(12)	-	177147	-	1893
(13)	-	531441	-	6520
(2 Processes) **Commits** (c) $\mathsf{TO} = 6\mathsf{h}$				
(2)	1060	293	10.7	1.8
(3)	77830	16401	1977	216
(4)	-	1008755	-	23768
(2 Processes) **Eratosthenes** (N) $\mathsf{TO} = 12\mathsf{h}$				
(5)	995	305	550	8
(6)	3980	1045	6697	26
(7)	6720	2840	34849	173
(8)	-	4333	-	444
(2 Processes) **Peterson** (c, a) $\mathsf{TO} = 5\mathsf{h}$				
(1,2)	145	90	7	0.6
(1,3)	275	222	32	2
(2,1)	2795	1499	1096	18
(2,2)	-	26110	-	649
(2,3)	-	155419	-	7240
(3,1)	-	117916	-	3593
(2 Processes) **KeyRotation** (c) $\mathsf{TO} = 4\mathsf{h}$				
(2)	570	181	6	0.9
(3)	18620	5503	1757	65
(4)	-	176503	-	4291

Table 3: Experimental evaluation and comparison to [12].

8. Related Work

The analysis of concurrent programs is a major challenge in program analysis and verification, and has been a subject of extensive study [6, 8, 10, 11, 22, 26, 33]. The hardness of reproducing bugs by testing, due to scheduling non-determinism, makes model checking a very relevant approach [3, 4, 9, 13, 17, 30], and in particular stateless model checking to combat the state-space explosion. To combat the exponential number of interleaving explosion faced by the early model checking [16], several reduction techniques have been proposed such as POR and context bounding [30, 32]. Several POR methods, based on persistent set [7, 7, 15, 39] and sleep set techniques [16], have been explored. DPOR [12] presents on-the-fly construction of persistent sets, and several variants and improvements have been considered [25, 35–38]. In [1], source sets and wakeup trees techniques were developed to make DPOR optimal, in the sense that the enumerative procedures explores exactly one representative from each Mazurkiewicz class. Other important works include normal form representation of concurrent executions [20] using SAT or SMT-solvers; or using unfoldings for optimal reduction in number of interleavings [19, 29, 34]. Techniques for transition-based POR for message passing programs have also been considered [15, 18, 21], and some works extend POR to relaxed memory models [2, 40].

9. Conclusions

We introduce the new observation equivalence on traces that refines the Mazurkiewicz equivalence and can event be exponentially more succinct. We develop an optimal, data-centric DPOR algorithm for acyclic architectures based on this new equivalence, and also extend a finer version of it to cyclic architectures. Our experimental results show significant improvement on academic examples. There are several future directions based on the current work. First, it is interesting to determine whether other, coarser equivalence classes can be developed for cyclic architectures, which can be used by some enumerative exploration of the trace space. Another promising direction is phrasing our observation equivalence on other memory models and developing DPOR algorithms for such models. Finally, it would be interesting to explore the engineering challenges in applying our approach to real-life examples.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, 2014.

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. *Stateless Model Checking for TSO and PSO*. 2015.

[3] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification - 25th International Conference*, pages 141–157, 2013.

[4] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. *Zing: A Model Checker for Concurrent Software*. 2004.

[5] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121 – 123, 1979.

[6] J.-M. Cadiou and J.-J. Lévy. Mechanizable proofs about parallel processes. In *14th Annual IEEE Symposium on Switching and Automata Theory*, 1973.

[7] E. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2), 1986.

[9] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[10] A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 297–308, 2012.

[11] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, 2009.

[12] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, 2005.

[13] C. Flanagan and S. Qadeer. Softmc 2003, workshop on software model checking (satellite workshop of cav '03) transactions for software model checking. 2003.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

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

[16] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, 1997.

[17] P. Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[18] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. *Form. Methods Syst. Des.*, 7(3):227–241, 1995.

[19] K. Kähkönen, O. Saarikivi, and K. Heljanko. Using unfoldings in automated testing of multithreaded programs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[20] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, 2009.

[21] S. Katz and D. Peled. Defining conditional independence using collapses. *Theor. Comput. Sci.*, 101(2):337–359, 1992.

[22] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.*, 35(1):73–97, 2009.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[24] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[25] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, pages 308–322, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[27] T. B. Madan Musuvathi, Shaz Qadeer. Chess: A systematic testing tool for concurrent software. Technical report, November 2007.

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

[29] K. L. McMillan. A technique of state space search based on unfolding. *Form. Methods Syst. Des.*, 6(1):45–65, 1995.

[30] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.

[31] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2008.

[32] D. Peled. All from one, one for all: On model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV, 1993.

[33] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[34] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening. Unfolding-based partial order reduction. In *26th International Conference on Concurrency Theory, CONCUR*, 2015.

[35] O. Saarikivi, K. Kahkonen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In *Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design*, pages 132–141, 2012.

[36] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, pages 339–356, 2006.

[37] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proceedings of the 2Nd International Haifa Verification Conference on Hardware and Software, Verification and Testing*, pages 166–182, Berlin, Heidelberg, 2007. Springer-Verlag.

[38] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. FMOODS'12/FORTE'12, pages 219–234, Berlin, Heidelberg, 2012. Springer-Verlag.

[39] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.

[40] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

A. Missing Proofs

A.1 Proofs of Section 4

Lemma 2. *Let $X = \mathsf{dom}(\mathsf{A}^+) \cap \mathsf{img}(\mathsf{A}^+)$ be the set of events that appear in a positive annotation A^+, and $X_i = X \cap \mathcal{E}_i$ the subset of events of X from process p_i. The following assertions hold:*

1. *If A^+ is well-formed, then it has a unique basis $(\tau_i)_i$.*
2. *Computing the basis of A^+ (or concluding that A^+ is not well-formed) can be done in $O(n)$ time, where $n = \sum_i(|\tau_i|)$ if A^+ is well-formed, otherwise $n = \sum_i \ell_i$, where ℓ_i is the length of the longest path from the root r_i of CFG_i to an event $e \in X_i$.*
3. *For every trace t that realizes A^+ we have that A^+ is well-formed and $t \in \tau_1 * \ldots * \tau_k$.*

Proof. 1. Assume that A^+ has two distinct bases $(\tau_i)_i$ and $(\tau'_i)_i$. Since each sequential trace corresponds to a deterministic computation of the corresponding process using $\mathsf{val}_{\mathsf{A}^+}$ as the value function of global events, we have that each τ_i and τ'_i share a prefix relationship (i.e., one is prefix of the other). Since the two basis are distinct, for some j we have that one of τ_j and τ'_j is a proper prefix of the other. Assume wlog that τ'_j is a strict prefix of τ_j. Then replacing τ_i with τ'_i in $(\tau_i)_i$ yields another basis, thus τ_i is not minimal, a contradiction.

2. We outline the process of constructing the basis $(\tau_i)_i$. As a preprocessing step, we compute for each process p_i the unique event $e_i \in X_i$ which is maximal wrt the program structure PS. Note that if e_i is not unique for each process, then A^+ is not well-formed. This requires $O(|\mathsf{A}^+|)$ time for all processes p_i, simply by iterating over all events in X. Then, the unique basis $(\tau_i)_i$ can be constructed by executing each process locally, and using the value function $\mathsf{val}_{\mathsf{A}^+}$ for assigning values to global events. The execution stops when e_i is reached, and the constructed sequential trace τ_i is returned. Let τ'_i be the global projection of τ_i. If $\mathcal{E}(\tau'_i) \triangle X_i \neq \emptyset$ (where $\triangle$ is the symmetric difference operator), return that A^+ is not well-formed. Otherwise, return $(\tau_i)_i$ as the basis of A^+.

3. It follows easily from Remark 1 and the above construction that if t realizes A^+, then A^+ must be well-formed and $t \in \tau_1 * \ldots * \tau_k$.

□

Lemma 3. *ACYCLIC EDGE ADDITION is NP-hard.*

Proof. The proof is by reduction from MONOTONE ONE-IN-THREE SAT [14, LO4]. In MONOTONE ONE-IN-THREE SAT, the input is a propositional 3CNF formula ϕ in which every literal is positive, and the goal is to decide whether there exists a satisfying assignment for ϕ that assigns exactly one literal per clause to True.

The reduction proceeds as follows. In the following, we let C and D range over the clauses and x_i over the variables of ϕ. We assume w.l.o.g. that no variable repeats in the same clause. For every variable x_i, we introduce a node $w'_i \in V$. For every clause $C = (x_{C_1} \vee x_{C_2} \vee x_{C_3})$, we introduce a pair of nodes $w^C_{C_j}, r^C_{C_j} \in V$ and an edge $(w^C_{C_j}, r^C_{C_j}) \in E$, where $j \in \{1,2,3\}$. Additionally, we introduce an edge $(w^C_{C_j}, w'_{C_l}) \in E$ for every pair $j, l \in \{1,2,3\}$ such that $j \neq l$, and an edge $(w'_{C_j}, r^C_{C_l})$ for each $j \in \{1,2,3\}$, where $l = (j+1) \mod 3 + 1$. Finally, for every pair of clauses C, D and $l_1, l_2 \in \{1,2,3\}$ such that $C_{l_1} = D_{l_2} = \ell$ (i.e., C and D share the same variable x_ℓ in positions l_1 and l_2), we add edges $(w^C_\ell, r^D_\ell), (w^D_\ell, r^C_\ell) \in E$. The set H consists of triplets of nodes $(w'_{C_j}, w^C_{C_j}, r^C_{C_j})$ for every clause C and $j \in \{1,2,3\}$. Figure 7 illustrates the above construction.

Let X be an edge addition set that solves ACYCLIC EDGE ADDITION on input (G, H) and note that for every pair of triplets $(w'_i, w^C_i, r^C_i), (w'_i, w^D_i, r^D_i) \in H$, we have that $(w'_i, w^C_i) \in X$ iff $(w'_i, w^D_i) \in X$, i.e., for every node w'_i, the set X contains either only all incoming, or only all outgoing edges of w'_i specified by H. To see this, observe that if there exists such a pair of triplets with $(w'_i, w^C_i), (r^D_i, w'_i) \in H$, then $G_X = (V, E \cup X)$ would contain the cycle

$$w'_i \to w^C_i \to r^D_i \to w'_i$$

which contradicts that X is a solution to the problem. Given such an edge addition set X, we obtain an assignment on the variables of ϕ by setting $x_i = \mathsf{True}$ iff X contains an edge $(w'_{C_j}, w^C_{C_j})$ for some clause C and $j \in \{1,2,3\}$. By the previous remark, the assignment of values to variables of ϕ is well-defined.

It is easy to verify that the construction takes polynomial time in the size of ϕ. In the following, we argue that the answer to MONOTONE ONE-IN-THREE SAT is true iff the answer to ACYCLIC EDGE ADDITION is also true.

($\Rightarrow$). Let X be a solution to ACYCLIC EDGE ADDITION on (G, H), and we argue that every clause C of ϕ contains exactly one variable set to True. Indeed, C contains at least one such variable, otherwise G_X would contain a cycle

$$r^C_{C_1} \to w'_{C_1} \to r^C_{C_2} \to w'_{C_2} \to r^C_{C_3} \to w'_{C_3} \to r^C_{C_1}$$

Similarly, C contains at most one variable of C set to True, as two such variables x_i, x_j would imply that G_X contains a cycle

$$w'_i \to w^C_i \to w'_j \to w^C_j \to w'_i$$

($\Leftarrow$). Consider any satisfying assignment of ϕ, and construct an edge addition set $X = \{e_i\}_i$ such that for each triplet $(w'_i, w^C_i, r^C_i) \in H$ we have

$$e_i = \begin{cases} (w'_i, w^C_i), & \text{if } x_i = \mathsf{True} \\ (r^C_i, w'_i), & \text{if } x_i = \mathsf{False} \end{cases}$$

Given a clause C, we denote by $V_C = \bigcup_{i=1}^3 \{w'_{C_i}, w^C_{C_i}, r^C_{C_i}\}$, and by $G_X[V_C]$ the subgraph of G_X restricted to nodes in V_C. We now argue that G_X does not contain a cycle. Assume towards contradiction otherwise, and let $\mathcal{C}$ be such a cycle.

1. If $\mathcal{C}$ contains a node of the form $w^C_{C_j}$ for some $j \in \{1,2,3\}$ then $\mathcal{C}$ traverses the edge $(w'_{C_j}, w^C_{C_j})$, and thus x_ℓ is assigned True, where $\ell = C_j$ Since G_X contains no edge of the form (r^D_ℓ, w'_ℓ) for any clause D, $\mathcal{C}$ must traverse an edge $(w^D_{D_i}, w'_\ell)$ for some clause D. Hence there is a first node $w^D_{D_i}$ traversed by $\mathcal{C}$ after w^C_ℓ, for some clause D and $i \in \{1,2,3\}$ and thus G_X contains an edge $(w'_{D_i}, w^D_{D_i})$, and hence x_l is assigned True, where $l = D_i$. By our choice of $w^D_{D_i}$, the node w'_{D_i} can only be reached via the edge $(r^D_{D_i}, w'_{D_i})$, which requires that x_l is assigned False, a contradiction.
2. If $\mathcal{C}$ contains no node of the form $w^C_{C_j}$, then it must be a cycle in $G[V_C]$, for some clause C. The only such cycle that traverses no $w^C_{C_j}$ can be

   $$r^C_{C_1} \to w'_{C_1} \to r^C_{C_2} \to w'_{C_2} \to r^C_{C_3} \to w'_{C_3} \to r^C_{C_1}$$

   which requires that all x_{C_i} are assigned False, for each $i \in \{1,2,3\}$, which contradicts that C has a variable assigned True.

The desired result follows. □

Lemma 9. *The decision problem of ACYCLIC EDGE ADDITION on input $(G = (V, E), H)$ admits a positive answer iff the positive annotation A^+ is realizable in $\mathcal{P}$.*

Proof. We present both directions of the proof.

- ($\Rightarrow$). Let t be any trace that realizes A^+. Observe that for any edge $(u, v) \in E$ we have $\mathsf{in}_t(e(u)) < \mathsf{in}_t(e(v))$, since $\mathsf{PS}(e(u), w_{u,v})$ and $\mathsf{PS}(r_{u,v}, e(v))$ and $\mathsf{A}^+(r_{u,v}) = w_{u,v}$. Additionally, t satisfies that either $\mathsf{in}_t(w_i') < w_t(w_i)$, or $\mathsf{in}_t(r_i) < w_t(w_i')$. In the former case we an edge (x_i, y_i), and in the latter case we add (z_i, x_i) in an edge set X. Since t induces a total order on the vertices of G, G_X is acyclic, and thus X is an edge addition set for (G, H).
- ($\Leftarrow$). If X is an edge addition set for (G, H) then G_X is acyclic and any topological order of the vertices of $G_X = (V, E \cup X)$ induces a trace that realizes A^+.

The desired result follows. □

A.2 Proofs of Section 5

Lemma 6 (Compactness)**.** *Consider any two executions of* DC-DPOR *on inputs* (t_1, A_1) *and* (t_2, A_2)*. Then* $\mathsf{A}_1^+ \neq \mathsf{A}_2^+$.

Proof. Examine the recursion tree T generated by DC-DPOR, where every node u is labeled with the trace t_u and annotation input $\mathsf{A}_u = (\mathsf{A}_u^+, \mathsf{A}_u^-)$ given to DC-DPOR. Let x and y be the nodes that correspond to inputs (t_1, A_1) and (t_1, A_2) respectively, and we argue that $\mathsf{A}_x \neq \mathsf{A}_y$. If x is an ancestor of y, then when DC-DPOR was executed on input (t_x, A_x), a read r created $\mathsf{A}_{r,w}^+$ in Line 3 on the branch from x to the direction of y in T, with the property that $r \notin \mathsf{dom}(\mathsf{A}_x^+)$. Since the algorithm never removes pairs (r, w) from the positive annotations, we have that $(r, w) \in \mathsf{A}_y$, hence $\mathsf{A}_x \neq \mathsf{A}_y$. A similar argument holds if y is an ancestor of x. Now consider the case that x and y do not have an ancestor-descendant relationship. Let z be the lowest common ancestor of x and y in T, and z_x (resp. z_y) the child of z in the direction of x (resp. y). Let r_x (resp. r_y) be the read on Line 3 that generated $\mathsf{A}_{z_x}^+$ (resp. $\mathsf{A}_{z_y}^+$), i.e.,

$$\mathsf{A}_{z_x}^+ = \mathsf{A}_z^+ \cup \{(r_x, w_x)\} \quad \text{and} \quad \mathsf{A}_{z_y}^+ = \mathsf{A}_z^+ \cup \{(r_y, w_y)\}$$

If $r_x = r_y = r$ then $w_x \neq w_y$, thus $\mathsf{A}_{z_x}^+(r) \neq \mathsf{A}_{z_y}^+$, and since the algorithm never removes pairs (r, w) from positive annotations we have that $\mathsf{A}_x \neq \mathsf{A}_y$. Now assume that $r_x \neq r_y$, and w.l.o.g. that $\mathsf{in}_{t_j}(r_x) < \mathsf{in}_{t_j}(r_y)$. Then, before DC-DPOR$(\mathsf{A}_{z_y})$ is executed, Line 7 adds $w_x \in \mathsf{A}^-(r_x)$. Since the algorithm never removes entries from the negative annotation, by Line 3, we have that $(r_x, w_x) \notin \mathsf{A}_y^+$. In all cases we have $\mathsf{A}_x \neq \mathsf{A}_y$, as desired. □

Lemma 7 (Completeness)**.** *For every realizable observation function* O*,* DC-DPOR *generates a trace* t *that realizes* O*.*

Proof. Let T be the recursion tree of DC-DPOR. Given a node u of T, we denote by t_u and $\mathsf{A}_u = (\mathsf{A}_u^+, \mathsf{A}_u^-)$ the input of DC-DPOR on u. Since t_u is always a maximal extension of a trace returned by procedure Realize on input A_u^+, by Lemma 4 we have that t_u is compatible with A_u^+, thus it suffices to show that DC-DPOR is called with a positive annotation being equal to O. We define a traversal on T with the following properties:

1. If u is the current node of the traversal, then $\mathsf{A}_u^+ \subseteq \mathsf{O}$.
2. If $\mathsf{A}_u^+ \subset \mathsf{O}$, then the traversal proceeds either
   (a) to a node v of T with $\mathsf{A}_u^+ \subset \mathsf{A}_v^+$,
   (b) to some other node of T,
   and Item 2b happens a finite number of times.

Observe that every time the traversal executes Item 2a, O agrees with A_v^+ on more reads than A_u^+. Since Item 2b is executed a finite number of times, any such traversal is guaranteed to reach a node w with $\mathsf{A}_w = \mathsf{O}$.

The traversal starts from u being the root of T, and Item 1 holds as then $\mathsf{A}_u^+ = \varnothing \subseteq \mathsf{O}$. Now consider that the traversal is on any node u that satisfies Item 1. Since t_u is a maximal extension of a trace returned by procedure Realize on input A_u^+, Lemma 4 guarantees that t_u is a maximal trace that realizes A_u^+. Let t^* be a trace that realizes O, and r be the first read of t^* not in A_u^+, i.e.,

$$r = \arg \min_{r' \in \mathcal{E}(t^*) \setminus \mathsf{dom}(\mathsf{A}_u^+)} \mathsf{in}_{t^*}(r')$$

and $w = \mathsf{O}_{t^*}(r)$. Then for every $r' \in \mathsf{Past}_{t^*}(r)$, we have $\mathsf{in}_{t^*}(r') < \mathsf{in}_{t^*}(r')$ and thus $r' \in \mathcal{E}(t_u)$ and $\mathsf{O}_{t^*}(r') = \mathsf{O}_{t_u}(r')$. By Lemma 5, we have $r \in \mathcal{E}(t_u)$. Since $\mathsf{in}_{t^*}(w) < \mathsf{in}_{t^*}(r)$, a similar argument yields that $w \in \mathcal{E}(t_u)$. We now distinguish two cases, based on whether $w \in \mathsf{A}_u^-(r)$ or not.

1. If $w \notin \mathsf{A}_u^-(r)$, then in Line 4 the algorithm will generate a trace t' that is compatible with the strengthened annotation $\mathsf{A}_u^+ \cup (r, w)$, and call itself recursively on some child v of u with $\mathsf{A}_v^+ = \mathsf{A}_u^+ \cup (r, w)$. The traversal proceeds to v and Item 1 holds, as desired.
2. If $w \in \mathsf{A}_u^-(r)$, then there exists a highest ancestor x of u in T where DC-DPOR was called with $(r, w) \in \mathsf{A}_x^-$. Following Line 7, this can only have happened if x has a sibling v in T with $(r, w) \in \mathsf{A}_v^+$. Let z be the parent of x, v, and we have $\mathsf{A}_z^+ \subset \mathsf{A}_u^+ \subset \mathsf{O}$, and $\mathsf{A}_v^+ = \mathsf{A}_z^+ \cup (r, w) \subseteq \mathsf{O}$. Thus, the traversal proceeds to v and Item 1 holds, as desired. In this case, we say that the traversal *backtracks to* x *through* z.

Finally, we argue that Item 2b will only occur a finite number of times in the traversal. Since T is finite, it suffices to argue that the traversal backtracks through any node z a finite number of times. Indeed, fix such a node z and let $x_1, x_2, \ldots$ be the sequence of (not necessarily distinct) children of z that the traversal backtracks to, through z. Let r_i be the unique read in $\mathsf{dom}(\mathsf{A}_{x_i}^+) \setminus \mathsf{dom}(\mathsf{A}_z^+)$. By Line 1, we have that $\mathsf{in}_{t_z}(r_{i+1}) < \mathsf{in}_{t_z}(r_i)$, hence no node repeats in the sequence $(x_i)_i$, and thus the traversal backtracks through z a finite number of times. The desired result follows. □

Theorem 3. *Consider a concurrent acyclic architecture* $\mathcal{P}$ *of processes on an acyclic state space, and* $n = \max_{t \in \mathcal{T}_\mathcal{P}} |t|$ *the maximum length of a trace of* $\mathcal{P}$*. The algorithm* DC-DPOR *explores each class of* $\mathcal{T}_\mathcal{P}/\sim_\mathsf{O}$ *exactly once, and requires* $O\left(|\mathcal{T}_\mathcal{P}/\sim_\mathsf{O}| \cdot n^5\right)$ *time.*

Proof. Lemma 6 and Lemma 7 guarantee the optimality of DC-DPOR, i.e., that DC-DPOR explores each class of $\mathcal{T}_\mathcal{P}/\sim_\mathsf{O}$ exactly once. The time spent in each class is the time for attempting all possible mutations on the witness trace t which is the trace used by the algorithm to explore the corresponding observation function. There are at most n^2 such mutations, and according to Theorem 2, each such mutation requires $O(n^3)$ time to be applied (or conclude that t cannot be mutated in the attempted way). The desired result follows. □

A.3 Proofs of Section 6

Lemma 8. *For any two traces* $t_1, t_2 \in \mathcal{T}_\mathcal{P}$*, if* $t_1 \sim_M t_2$ *then* $t_1 \sim_\mathsf{O}^X t_2$*.*

Proof. By Theorem 2, we have $t_1 \sim_\mathsf{O} t_2$. Consider any pair of distinct write events $w_1, w_2 \in \mathcal{W}(t_1) \cap (\mathcal{W}_i \cup \mathcal{W}_j)$ with $\mathsf{loc}(w_1) = \mathsf{loc}(w_2) = g$ and $g \in \lambda_\mathcal{P}(p_i, p_j)$, and observe that w_1 and w_2 are dependent. Hence, we have $w_1 \rightarrow_{t_1} w_2$ iff $w_1 \rightarrow_{t_2} w_2$, and thus $\mathsf{in}_{t_1}(w_1) < \mathsf{in}_{t_1}(w_2)$ iff $\mathsf{in}_{t_2}(w_1) < \mathsf{in}_{t_2}(w_2)$, as desired. □

Lemma 10. *Given a well-formed positive annotation* A^+ *over a basis* $(\tau_i)_i$*, and the modified communication graph* $G'_{\mathcal{P}X} = (V_{\mathcal{P}X}, E_{\mathcal{P}X} \backslash X, \lambda_{\mathcal{P}X})$*,* Realize *constructs a trace* t *that realizes* A^+ *(or concludes that* A^+ *is not realizable) and requires* $O(n^3)$ *time, where* $n = \sum_i |\tau_i|$*.*

(Sketch). First, note that due to the observable locks $\mathcal{L}^{\mathcal{O}}$, A^+ already induces a total order on the lock-acquire and lock-release events that access the same lock $l_g \in \mathcal{L}^{\mathcal{O}}$. Hence A^+ already induces a total order between the write and lock-acquire events that are protected by the same lock $l_g \in \mathcal{L}^{\mathcal{O}}$. Thus, given the basis $(\tau_i)_i$, A^+ is realizable iff that total order respects A^+, and there is a way to order the remaining pairwise dependent events between pairs of processes $(p_i, p_j) \notin X$, such that the ordering respects the sequential consistency axioms and the lock semantics. The crucial property is that since X is an all-but-two cycle set of $G_{\mathcal{P}}$, the transitivity (Line 10) and antisymmetry (Line 7) clauses ensure that a satisfying assignment preserves acyclicity of the graph G' constructed in Line 29. The complexity analysis is similar to Lemma 4. □

Theorem 4. *Consider a concurrent architecture* $\mathcal{P}$ *of processes on an acyclic state space, and* $n = \max_{t \in \mathcal{T}_{\mathcal{P}}} |t|$ *the maximum length of a trace of* $\mathcal{P}$*. Let* X *be an all-but-two cycle set of the communication graph* $G_{\mathcal{P}}$*. The algorithm* DC-DPOR-Cyclic *explores each class of* $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^{X}$ *exactly once, and requires* $O\left(|\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^{X}| \cdot n^5\right)$ *time.*

Proof. We argue that DC-DPOR-Cyclic is then optimal for the cyclic architecture $\mathcal{P}$ wrt the equivalence $\sim_{\mathsf{O}}^{X}$.

1. (*Compactness*). For any two distinct positive annotations A_1^+, A_2^+ examined by DC-DPOR when exploring the trace space of $\mathcal{P}^X$, Lemma 6 guarantees that $\mathsf{A}_1^+ \neq \mathsf{A}_2^+$. Let t_1 and t_2 be the traces returned by Realize on inputs A_1^+ and A_2^+ respectively. Assume that t_1 is not a prefix of t_2 (the argument is similar if t_2 is not a prefix of t_1, and since $\mathsf{A}_1^+ \neq \mathsf{A}_2^+$, it is not the case that each is a prefix of the other). This implies that at least one of the following holds.
   (a) There is a read event $r \in \mathcal{E}(t_1)$ such that $(r, \mathsf{O}_{t_1}(r)) \notin \mathsf{O}_{t_2}$, in which case two different classes of $\sim_{\mathsf{O}}$ are explored. Since $\sim_{\mathsf{O}}^{X}$ refines $\sim_{\mathsf{O}}$ it follows that two different classes of $\sim \mathsf{O}^X$ are explored.
   (b) There is a lock-acquire event $e_a \in \mathcal{E}(t_1)$ such that $(e_a, \mathsf{O}_{t_1}(e_a)) \notin \mathsf{O}_{t_2}$. In this case, the write event w protected by the lock-acquire event e_a either does not appear t_2 or there exists a conflicting write event w' in t_1 such that t_1 and t_2 are ordered differently in t_1 and t_2. Hence the two annotations A_1^+ and A_2^+ are used to explore different classes of $\sim_{\mathsf{O}}^{X}$.
2. (*Completeness*). The completeness statement of Lemma 7 guarantees that for every observation function O of the trace space of $\mathcal{P}^X$, there is an annotation function A^+ used by DC-DPOR such that $\mathsf{O} = \mathsf{A}^+$. Since the lock-acquire and lock-release events on observable locks are read and write events respectively, any two traces which have a different order on a pair of write events w, w' such that w and w' are protected by observable locks, will also be explored.

Finally, the maximum size of a trace in $\mathcal{P}$ is asymptotically equal to the maximum size of a trace in $\mathcal{P}^X$, from which the complexity bound follows. □

Exponential succinctness of $\sim_{\mathsf{O}}^{X}$ in cyclic architectures. Here we present a very simple cyclic architecture where the observation equivalence $\sim_{\mathsf{O}}^{X}$ refined by an all-but-two cycle set X is exponentially more succinct than the Mazurkiewicz equivalence $\sim_M$. Consider the architecture $\mathcal{P}$ in Figure 8, which consists of three processes and two single global variables x and y. We choose an edge set as $X = \{(p_1, p_2)\}$, and X is an all-but-two cycle set of $G_{\mathcal{P}}$. We argue that $\sim_{\mathsf{O}}^{X}$ is exponentially more succinct than $\sim_M$ by showing exponentially many traces which are pairwise equivalence under $\sim_{\mathsf{O}}^{X}$ but not under $\sim_M$. Indeed, consider the set T which consists of all traces such that the following hold

Process p_1 :	Process p_2 :	Process p_3 :
1. write x	1. write x	1. write x
2. read x	2. write y	2. write y
	...	...
	$n+2$. write y	$n+2$. write y
	$n+3$. read y	$n+3$. read y
	$n+4$. read x	$n+4$. read x

Figure 8: A cyclic architecture of three processes.

1. All traces start with p_1 executing to completion, then p_2 executing its first statement, and p_3 executing its first statement.
2. All traces end with the last three events of p_2 followed by the last two events of p_3.

Note that $|T| = \binom{2 \cdot n}{n}$ as there are $(2 \cdot n)!$ ways to order the $2 \cdot n$ write y events of the two processes, but $n! \cdot n!$ orderings are invalid as they violate the program structure. All traces in T have the same observation function, yet they are inequivalent under $\sim_M$ since every pair of them orders two write y events differently. Finally, $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^{X}$ is only exponentially large, and since

$$|(\mathcal{T}_{\mathcal{P}}/\sim_M) \backslash (\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^{x})| \geqslant |T| - 1$$

we have that $\sim_{\mathsf{O}}^{X}$ is exponentially more succinct than $\sim_M$.

B. Benchmarks

B.1 Details

Underapproximating the non-necessarily maximal Mazurkiewicz classes.

Here we outline the process we used to underapproximate the number of not-necessarily maximal Mazurkiewicz classes reported in Table 3. Recall that each such class contains Mazurkiewicz-equivalent traces which are not necessarily maximal. We obtained this underapproximation by counting the number of maximal Mazurkiewicz classes discovered throughout our experiments. Consider any execution of DPOR, and let x be the number of maximal Mazurkiewicz classes discovered. Hence x represents the number of traces $(t_i)_{i=1}^{x}$ found at the leaves of the recursion tree T of DPOR, up to the Mazurkiewicz equivalence. In all cases where we used this underapproximation scheme, every path in the recursion tree from the root to a leaf contained at least 3 conflicting write events. Let u_i be the leaf of T that corresponds to t_i, and v_i be the first ancestor of v_i where the corresponding trace t'_i (i.e., t'_i is a prefix of t_i) has one less write event than t_i. Hence, for all but one processes, all write events of those processes that appear in t_i also appear in t'_i.

1. Every t'_i is Mazurkiewicz inequivalent to every t_j, otherwise t_j would not be maximal (as it could be extended to t_i).
2. Every v_i corresponds to a different u_i. Indeed, assume that there exists $j \neq i$ such that v_i corresponds to both u_i and u_j. Let s_i and s_j be the suffix of u_i and u_j respectively that is missing

from t'_i. Note that in this case s_i and s_j start with the same, single write event, and thus must be Mazurkiewicz equivalent, and since they both extend t'_i, then t_i and t_j would also be Mazurkiewicz equivalent, a contradiction.
3. Every t'_i is Mazurkiewicz inequivalent to every other t_j, with $i \neq j$, otherwise the next event on the path $v_i \rightsquigarrow u_i$ and $v_j \rightsquigarrow u_j$ would the same write event, and t_i and t_j would also be Mazurkiewicz equivalent.

Hence, the number of non-maximal, pairwise Mazurkiewicz inequivalent traces defined by the nodes v_i is at least x. Now let z_i be the first ancestor of v_i in T where the corresponding trace t''_i has one less write event than t'_i.

1. Every t'_i is Mazurkiewicz inequivalent to every t_j, for similar reasons as Item 1 above.
2. Every t''_i corresponds to at most two t'_i, t'_j. The argument is similar to Item 2 above, with the difference that this time t''_i can be extended with at most two different write events towards t'_i and t'_j.
3. Every t''_i is Mazurkiewicz inequivalent to every other t''_j, with $i \neq j$, for similar reasons as Item 3 above.

Hence there are at least $(1 + 1 + 1/2) \cdot x = 3/2 \cdot x$ not-necessarily maximal Mazurkiewicz classes in the trace space.

B.2 Pseudocode of Benchmark

Here we provide the formal pseudocode used in the benchmarks of Section 7. In all cases, k refers to the number of processes in the system. Besides the toy cases of Alternate write $-$ read, Pipeline our experiments include the following concurrent programming paradigms.

1. Peterson's solution to mutual exclusion for two processes.
2. A concurrent version of the sieve of Eratosthenes for finding prime numbers.
3. A scheme of optimistic Commits where each process writes to a variable, does some work optimistically assuming no other process has reached that region, and afterwards checks whether that variable was modified,
4. A scheme of two processes using a common encryption key to communicate with the outer-world. After every three iterations, each process modifies the encryption key to a new one.

Benchmark: Alternate write $-$ read

```
   Globals: int x
   Locals : int n, v, i ← 0
1  while i < n do
2  |  x ← v
3  |  v ← x
4  end
```

Benchmark: Pipeline

```
   Globals: int[k − 1] x
   // --------- Process j = 1 ---------
1  v ← x[j]
2  x[j] ← v

   // --------- Process 0 < j < k − 2 ---------
   Locals : int i ← 0, v
3  v ← x[j − 1]
4  x[j − 1] ← v
5  v ← x[j]
6  x[j] ← v

   // --------- Process j = k − 1 ---------
7  v ← x[j − 1]
8  x[j − 1] ← v
```

Method: Peterson

```
   Globals: int turn ← 0
            int c, a
            bool interested_1 ← False, interested_2 ← False
   // --------- Process 1 ⩽ j ⩽ 2 ---------
   Locals : int total ← 0, i ← 0
1  while total < c do
2  |  interested_j ← True
3  |  turn ← 1 − j
4  |  i ← 0
5  |  while True do
6  |  |  if interested_{1−j} and turn = 1 − j and then
   |  |  |  // Wait for some time
7  |  |  |  i ← i + 1
8  |  |  |  if i = a then
9  |  |  |  |  interested_j ← False
10 |  |  |  |  return
11 |  end
12 |  interested_j ← False
13 |  total ← total + 1
14 end
```

Method: Commits

```
   Globals: int c
            int x
   // --------- Process 1 ⩽ j ⩽ 2 ---------
   Locals : int i ← 0
1  while i < c do
   |  // Stamp
2  |  x ← j
   |  // Do stuff, then read stamp
3  |  if x = j then
   |  |  // Commit was successful
4  |  |  i ← i + 1
5  end
```

Method: Eratosthenes

```
   Globals: int N ← 16, primes ← 0
            int[N] naturals ← (0, ..., 0)
   Locals : int i ← 0, j ← 0
 1 while i < N do
 2 |  if naturals[i] = 0 then
 3 |  |  naturals[i] ← 1
 4 |  |  j ← 2 · i + 2
 5 |  |  while j < N do
 6 |  |  |  naturals[j] ← 2
 7 |  |  |  j ← j + i + 2
 8 |  |  end
 9 |  i ← i + 1
10 end
```

Method: KeyRotation

```
   Globals: int key ← False
            int c
   // --------- Thread 0 ⩽ j < 2 ---------
   Locals : int i ← 0, m ← 0, x ← 0
 1 while i < c do
 2 |  if i  mod 3 = 0 then
   |  |  // Rotate encryption key
 3 |  |  x ← GenerateKey()
 4 |  |  key ← x
 5 |  m ← NewMessage()
 6 |  SendMessage(m, key)
 7 end
```