Data-centric Dynamic Partial Order Reduction

MAREK CHALUPA, Masaryk University
KRISHNENDU CHATTERJEE, Institute of Science and Technology, Austria
ANDREAS PAVLOGIANNIS, Institute of Science and Technology, Austria
NISHANT SINHA, Kena Labs
KAPIL VAIDYA, Indian Institute of Technology, Bombay

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 [Musuvathi et al. 2008]), and the problem needs to be addressed by a systematic exploration of the state space. Model checking [Clarke et al. 1999a] 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 [Godefroid 1996], which avoids the above problem by exploring the state space without explicitly storing the global states. This is typically achieved by a scheduler, which drives the program execution based on the current interaction between the processes. Well-known tools such as VeriSoft [Godefroid 1997, 2005] and CHESS [Madan Musuvathi 2007] have successfully employed stateless model checking.

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 [Lal and Reps 2009; Musuvathi and Qadeer 2007], the most well-known method is partial order reduction (POR) [Clarke et al. 1999b; Godefroid 1996; Peled 1993]. 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 [Mazurkiewicz 1987], 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 [Godefroid 1996].

Dynamic Partial-order Reduction (DPOR). Dynamic partial-order reduction (DPOR) [Flanagan and Godefroid 2005] 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 [Flanagan and Godefroid 2005; Godefroid 2005; Musuvathi et al. 2008]. Recently, an optimal method for DPOR was developed [Abdulla et al. 2014]. We refer to Section 8 for more detailed references to related work.

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 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.

\begin{align*}
t_1 &: \text{w}_1, \text{r}_1, \text{w}_2, \text{r}_2 \\
t_2 &: \text{w}_1, \text{w}_2, \text{r}_1, \text{r}_2 \\
t_3 &: \text{w}_2, \text{w}_1, \text{r}_1, \text{r}_2 \\
t_4 &: \text{w}_2, \text{r}_2, \text{w}_1, \text{r}_1
\end{align*}

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 consistent 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 Mazurkiewicz 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 account read-write and memory consistency restrictions as opposed to the event-dependency relation of the Mazurkiewicz equivalence.

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. The complexity of the problem depends on the communication graph of the system, called the architecture. Intuitively, the nodes of the architecture represent the processes of the concurrent system, and there is an (undirected) edge between two nodes if the respective processes access a common shared variable. We show that 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.

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 [Abdulla et al. 2014; Flanagan and Godefroid 2005]. 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 \textit{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 \textit{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.

\subsection{2.1 Concurrent Computation Model}

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

\textbf{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 \(\sim\), i.e.,

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

The quotient set \(X/\sim_R := \{[x]_R | x \in X\}\) of \(X\) under \(\sim\) is the set of all equivalence classes of \(X\) under \(\sim\).

\textbf{Notation on functions.} We write \(f : X \to Y\) to denote that \(f\) is a partial function from \(X\) to \(Y\). Given a (partial) function \(f\), we denote by \(\text{dom}(f)\) and \(\text{img}(f)\) 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 \(x \in \text{dom}(f)\) and \(f(x) = y\). Given (partial) functions \(f\) and \(g\), we write \(f \subseteq g\) if \(\text{dom}(f) \subseteq \text{dom}(g)\) and for all \(x \in \text{dom}(f)\) 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 \(f \neq g\).

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

1. \(e : v \leftarrow \text{read} \ g\), in which case \(e \in \mathcal{R}\),
2. \(e : g \leftarrow \text{write} (v_1, \ldots, v_{n_i})\), in which case \(e \in \mathcal{W}\),
3. \(e : \text{acquire} \ l\), in which case \(e \in \mathcal{R}\),
4. \(e : \text{release} \ l\), in which case \(e \in \mathcal{W}\),
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 $r_i \leadsto x$ 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 1-4, or
3. $m \geq 2$ outgoing edges $(x, y_1), \ldots, (x, y_m)$ labeled with events $e_j : b_j(v_1, \ldots, v_n)$ of Item 5, and such that for all values of $v_1, \ldots, v_n$, we have $b_j(v_1, \ldots, v_n) \implies \neg b_j(v_1, \ldots, v_n)$ 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 $E_i \subseteq E$ be the set of events that appear in CFG$_i$ of process $p_i$, and similarly $R_i \subseteq R$ and $W_i \subseteq W$ the sets of read and write events of $p_i$. Additionally, we require that $E_i \cap E_j = \emptyset$ for all $i \neq j$ i.e., all $E_i$ are pairwise disjoint, and denote by proc($e$) the process of event $e$. The location of an event loc($e$) is the unique global variable it involves. Given two events $e, e' \in E_i$ for some $p_i$, we write PS($e, e'$) if there is a path $e \leadsto e'$ in CFG$_i$ (i.e., we write PS($e, e'$) to denote that $e$ is ordered before $e'$ in the program structure).

We distinguish a set of initialization events $W^I \subseteq W$ with $|W^I| = |G|$ which are attributed to process $p_i$, and are used to initialize all the global variables to some fixed values. For every initialization write event $w^I$ and for any event $e \in E_i$ of process $p_i$, we define that PS($w^I, e$) (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 $s_i = (x_i, \text{val}_i)$ where $x_i$ is a node of CFG$_i$ (i.e., the program counter) and val$_i$ is a valuation on the local variables $V_i$. A global state of $P$ is a tuple $s = (\text{val}, s_1, \ldots, s_k)$, where val is a valuation on the global variables $G$ 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 $s_i = (x, \text{val}_i)$ (i.e., the program counter is on node $x$) and additionally,

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

Fig. 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 $PS(e_2, e_4)$. We have $loc(e_1) = loc(e_5)$ and $loc(e_2) = loc(e_4) = loc(e_6)$.

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 if it respects the condition of the branch in $s$. Given a state $s$, we denote by $enabled(s) \subseteq E$ 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 $p_1$ in state $s = (val, s_1, \ldots, s_k)$ results in a state $s' = (val', s_1', \ldots, s_k')$, where $s' = (y, val')$. That is, the program counter of $p_1$ has progressed to $y$, and the valuation functions $val'$ and $val_i'$ have been modified according to standard semantics, as follows:

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

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 $\mathcal{S}_P$ 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 $\mathcal{T}_\mathcal{P} = (\mathcal{S}_\mathcal{P}, \Delta, s^0)$, 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

$$(s, s') \in \mathcal{T}_\mathcal{P} \text{ iff } \exists e \in enabled(s) : s \xrightarrow{e} s'$$

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 $\{s^i\}_{1 \leq i < n}$ such that

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

The initial state $s^0 = (val, s^0_1, \ldots, s^0_k)$ is such that the value $val(g)$ of each global variable $g$ comes from the unique initialization write event $w$ with $loc(w) = g$, and for each $s^0_i = (x_i, val_i)$ 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 $S_P$ 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 $S_P$ that are acyclic.

**Architecture topologies.** The architecture $P$ induces a labeled undirected communication graph $G_P = (V_P, E_P, \lambda_P)$ where $V_P = \{p_1\}$. There is an edge $(p_i, p_j)$ if processes $p_i, p_j$ access 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 $P$ acyclic if $G_P$ 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.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = (p_i)_{i=1}^k$</td>
<td>the concurrent architecture of $k$ processes</td>
</tr>
<tr>
<td>$G, \mathcal{V}, \mathcal{L}$</td>
<td>the global, local and lock variables</td>
</tr>
<tr>
<td>$E, W, R, \mathcal{L}^A, \mathcal{L}^R, \mathcal{W}^I$</td>
<td>the set of events, write, read, lock-acquire lock-release and initialization events</td>
</tr>
<tr>
<td>val, val</td>
<td>valuations of local, global variables</td>
</tr>
<tr>
<td>enabled(s) $\subseteq E$</td>
<td>the set of enabled events in $s$</td>
</tr>
<tr>
<td>$s \xrightarrow{e_1,\ldots,e_n} s'$</td>
<td>sequence of events from $s$ to $s'$</td>
</tr>
<tr>
<td>proc(e), loc(e)</td>
<td>the process, the global variable of event $e$</td>
</tr>
<tr>
<td>CFG, PS $\subseteq \mathcal{E} \times \mathcal{E}$</td>
<td>the control-flow graph of process $p_i$, and the program structure relation</td>
</tr>
<tr>
<td>$G_P = (V_P, E_P, \lambda_P)$</td>
<td>the communication graph of $P$</td>
</tr>
</tbody>
</table>

**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 $t = e_i, \ldots, e_j$ such that for all $1 \leq i < j$, we have $s^0 \xrightarrow{e_i} s^i$, where $s^i \in S_P$ and $s^0$ is the initial state of $P$. In such a case, we write succinctly $s^0 \xrightarrow{t} s^j$. We fix the first $|G|$ events $e_1, \ldots, e_{|G|}$ of each trace $t$ to be initialization events that write the initial values to the global variables. That is, for all $1 \leq i \leq |G|$ we have $e_i \in \mathcal{W}$, and hence every trace $t$ starts with an initialization trace $t^I$ as a prefix. Given a trace $t$, we denote by $E(t)$ the set of events that appear in $t$, with $R(t) = E(t) \cap R$ the read events in $t$, and with $W(t) = E(t) \cap W$ the write events in $t$, and let $|t| = |E(t)|$ be the length of $t$. For an event $e \in E(t)$, we write $in_t(e) \in \mathbb{N}^+$ 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 $\ell$, and we say that $t$ is an extension of $t[\ell]$. We let enabled($t$) denote the set of enabled events in the state at the end of $t$, and call $t$ maximal if enabled($t$) = $\emptyset$. We write $T_P$ for the set of all maximal traces of $P$. We denote by $s(t)$ the unique state of $P$ such that $s^0 \xrightarrow{t} s(t)$, and given an event $e \in R(t) \cup W(t)$, denote by val$_t(e) \in \mathcal{D}$ the value that the unique global variable of $e$ has in $s(t[\text{in}_t(e)])$. 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 \(\tau_i\) is a sequence of events that correspond to a path in \(\text{CFG}_{i}\), starting from the root node \(r_i\). Note that a sequential trace is only wrt \(\text{CFG}_{i}\), 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 \(\tau_i\), respectively).

Given \(k\) sequential traces \(\tau_1, \tau_2, \ldots, \tau_k\), so that each \(\tau_i\) is wrt \(p_i\), we denote by \(\tau_1 * \tau_2 * \ldots * \tau_k\) the (possibly empty) set of all traces \(t\) such that \(\mathcal{E}(t) = \bigcup_{1 \leq i \leq k} \mathcal{E}(\tau_i)\).

**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 \(\text{Conf}(e_1, e_2)\) if \(\text{loc}(e_1) = \text{loc}(e_2)\) 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 \(\text{Conf}(e_1, e_2)\). Two events \(e_1, e_2\) are said to be independent [Flanagan and Godefroid 2005; Godefroid 1996] if \(p(e_1) \neq p(e_2)\) and

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 \(e_{3-i} \in \text{enabled}(s_1)\) iff \(e_{3-i} \in \text{enabled}(s_2)\), and
2. for any pair of states \(s_1, s_2\) such that \(e_1, e_2 \in \text{enabled}(s_1)\), we have that \(s_1 \xrightarrow{e_1 \cdot e_2} s_2\) iff \(s_1 \xrightarrow{e_2 \cdot e_1} s_2\), and dependent otherwise. Following the standard approach in the literature, we will consider two conflicting events to be always dependent [Godefroid 1997, 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

\[ e_1 \rightarrow_t e_2 \quad \text{if} \quad \text{in}_t(e_1) \leq \text{in}_t(e_2) \quad \text{and} \quad e_1 \text{ and } e_2 \text{ are dependent}. \]

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

<table>
<thead>
<tr>
<th>Notation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t, \tau_i)</td>
<td>a trace and a sequential trace</td>
</tr>
<tr>
<td>Conf((e_1, e_2))</td>
<td>conflicting events</td>
</tr>
<tr>
<td>(t</td>
<td>\ell,</td>
</tr>
<tr>
<td>(\mathcal{E}(t), \mathcal{W}(t), \mathcal{R}(t))</td>
<td>the events, write and read events of trace (t)</td>
</tr>
<tr>
<td>in(_t)((e)), val(_t)((e))</td>
<td>the index and value of event (e) in trace (t)</td>
</tr>
<tr>
<td>(t</td>
<td>X)</td>
</tr>
<tr>
<td>enabled(_t)</td>
<td>the enabled events in the state reached by (t)</td>
</tr>
<tr>
<td>(\rightarrow_t)</td>
<td>the happens-before relation on (t)</td>
</tr>
<tr>
<td>(O_t)</td>
<td>the observation function of (t)</td>
</tr>
</tbody>
</table>

**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 and arrays. First, note that the location \( \text{loc}(e) \) of every event \( e \in \mathcal{R} \cup \mathcal{W} \) is taken to be fixed in each \( \text{CFG}_i \). 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. Arrays are handled naturally in our implementation, and we refer to the Experiments Section 7.1 for a description.

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 \( e : g \leftarrow \text{write} f(v_1, \ldots, v_j) \) and \( e : b(v_1, \ldots, v_n) \) 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 events as reads and lock-release events as writes. In a trace \( t \), a lock-acquire event \( e \) is considered to read the value of the last lock-release event \( e' \) on the same lock \( l \) (or some initialization event \( \text{Init} \) if \( e \) is the first lock event on \( l \) in \( t \)). 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.

3 OBSERVATION TRACE EQUIVALENCE

In this section, we introduce the observation equivalence \( \sim_O \) 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 \( \sim_O \) refines \( \sim_M \), that is, every two traces that are equivalent under reordering of independent events are also equivalent under observations. We conclude by showing that \( \sim_O \) 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 \( t_1, t_2 \in \mathcal{T}_P \) 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 \( \mathcal{T}_P \), and we have \( t_1 \sim_M t_2 \) iff

\[
(1) \ E(t_1) = E(t_2),
\]
We start with the following crucial lemma. In words, it states that if two traces agree on their observation function.

Observation functions. The concurrent model introduced in Section 2.1 follows sequential consistency [Lampert 1979], 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 $O : \mathcal{R} \to \mathcal{W}$. A trace $t$ induces a total observation function $O_t : \mathcal{R}(t) \to \mathcal{W}(t)$ following the sequential consistency axioms. That is, $O_t(r) = w$ iff

(1) $in_t(w) < in_t(r)$, and

(2) for all $w' \in \mathcal{W}(t)$ such that $\text{Confl}(r, w')$ we have that $in_t(w') < in_t(w)$ or $in_t(w') > in_t(r)$.

We say that $t$ is compatible with an observation function $O$ if $O \subseteq O_t$, and that $t$ realizes $O$ if $O = O_t$.

Observation equivalence. We define the observation equivalence $\sim_O$ on the trace space $\mathcal{T}_P$ as follows. For $t_1, t_2 \in \mathcal{T}_P$ we have $t_1 \sim_O t_2$ iff $\mathcal{E}(t_1) = \mathcal{E}(t_2)$ and $O_{t_1} = O_{t_2}$, 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 3.1. Consider two traces $t_1, t_2$ such that $O_{t_1} \subseteq O_{t_2}$. Then

- for all read events $r \in \mathcal{R}(t_1)$ we have that $\text{val}_{t_1}(r) = \text{val}_{t_2}(r)$, and
- for all write events $w \in \mathcal{W}(t_1) \cap \mathcal{W}(t_2)$ we have that $\text{val}_{t_1}(w) = \text{val}_{t_2}(w)$.

Proof. The proof is by induction on the prefixes of $t_1$. We show inductively that for every $0 \leq \ell \leq |t_1|$, for all global events $e \in \mathcal{E}(t_1[\ell])$ we have that if $e \in \mathcal{E}(t_2)$ then $\text{val}_{t_1}(e) = \text{val}_{t_2}(e)$. Note that in the case where $e = r$ is a read event, then $r \in \mathcal{E}(t_2)$ follows directly from $O_{t_1} \subseteq O_{t_2}$. The claim is true for $\ell = 0$, since in that case $t_1[\ell] = \epsilon$ and no event appears in $t_1[\ell]$. Now assume that the claim holds for all prefixes $t_1[j]$ for $0 \leq j < \ell$, and let $e = \arg \min_{e' \in \mathcal{E}(t_1), \mathcal{E}(t_1[\ell])} in_{t_1}(e')$ be the next global event in $t_1$. We distinguish two cases based on the type of $e$.

- $e = w \in \mathcal{W}$ is a write event of the form $g \leftarrow f(v_1, \ldots, v_{n_i})$, such that $p_i = \text{proc}(w)$. Then the value of each local variable $v_j$ equals some $\alpha_j = \text{val}_{t_1}(r_j)$, with $\text{in}_{t_1}(r_j) \leq \ell$, and by the induction hypothesis we have $r_j \in \mathcal{E}(t_2)$ and $\text{val}_{t_1}(r_j) = \text{val}_{t_2}(r_j) = \alpha_j$. Since $p_i$ is deterministic, it easily follows that if $w \in \mathcal{E}(t_2)$ then
  
  \begin{align*}
  \text{val}_{t_2}(w) &= f(\text{val}_{t_1}(r_1), \ldots, \text{val}_{t_1}(r_{n_i})) \\
  &= f(\alpha_1, \ldots, \alpha_{n_i}) = f(\text{val}_{t_2}(r_1), \ldots, \text{val}_{t_2}(r_{n_i})) \\
  &= \text{val}_{t_2}(w)
  \end{align*}

- $e = r \in \mathcal{R}$ is a read event, and let $w = O_{t_1}(r)$. Let $p_i = \text{proc}(w)$, and since $p_i$ is deterministic, it easily follows from the induction hypothesis that $r \in \mathcal{E}(t_2)$, and moreover that $\text{val}_{t_1}(w) = \text{val}_{t_2}(w)$. Since $O_{t_2}(r) = w$, we have that

\begin{align*}
\text{val}_{t_2}(r) &= \text{val}_{t_2}(w) = \text{val}_{t_1}(w) = \text{val}_{t_1}(r)
\end{align*}

The desired result follows.
The following is an easy consequence of Lemma 3.1.

**Lemma 3.2.** Consider two traces \( t_1, t_2 \) such that \( O(t_1) \subseteq O(t_2) \) and \( t_2 \) is maximal. Then (i) \( E(t_1) \subseteq E(t_2) \), and (ii) for all events \( e \in R(t_1) \cup W(t_1) \) we have that \( val_{t_1}(e) = val_{t_2}(e) \).

**Soundness.** Lemma 3.2 implies that two maximal traces which agree on their observation function have the same observable behavior, i.e., each global event has the same value in the two traces. Since all local states of each process can be explored by exploring maximal traces, it suffices to explore all the (maximal) observation functions 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. The formal relationship between the two equivalences is established in the following theorem.

**Theorem 3.3.** For any two traces \( t_1, t_2 \in T_{\mathcal{P}} \), if \( t_1 \sim_M t_2 \) then \( t_1 \sim_O t_2 \).

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

1. \( r \rightarrow_t w_2 \), and since \( w_2 \rightarrow_t r \) then \( t_1 \sim_M t_2 \), a contradiction.
2. \( w_2 \rightarrow_t w_1 \), and since \( t_1 \sim_M t_2 \) we have that \( w_2 \rightarrow_t w_1 \), and thus \( r \rightarrow_t w_1 \). Since \( w_1 \rightarrow_t r \), we have \( t_1 \sim_M t_2 \), a contradiction.

The desired result follows.

\[ 
\begin{align*}
X_1 &
| e_1 : \text{acquire } l \\
X_2 &
| e_2 : v \leftarrow \text{read balance} \\
X_3 &
| e_3 : b(v, \text{amount}) \\
X_4 &
| e_4 : \text{balance} \leftarrow f(v, \text{amount, success}) \\
X_5 &
| e_5 : \text{release } l \\
X_6 &
| e_6 : v \leftarrow \text{read balance} \\
X_7 &
\end{align*}
\]

\[ 
\begin{align*}
\text{Mazurkiewicz-based} &
| e'_1 : \text{release } l \\
&
| e'_2 : \text{balance} \leftarrow 4 \\
\text{Observation-based} &
| e'_1 : \text{release } l \\
&
| e'_2 : \text{balance} \leftarrow 4 \\
\end{align*}
\]

*Fig. 4.* Trace exploration on the system of Figure 3 with two processes, where initially balance \( \leftarrow 4 \) and both withdrawals succeed.
Example 3.4 (Mazurkiewicz-based-based vs observation exploration.). 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 $t^I$ that initializes the lock $l$ to False, and the initial value deposit = 4. Consider that $p_1$ is executed with parameter amount = 1 and $p_2$ is executed with parameter 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 $\sim_O$. 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 $\sim_O$ 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 3.3 does not hold in the other direction, i.e., $\sim_O$ can be strictly coarser than $\sim_M$. Here we provide two simple examples in which $\sim_O$ is exponentially more succinct than $\sim_M$. Traditional enumerative model checking methods of concurrent systems are based on exploring at least one trace from every partition of the Mazurkiewicz equivalence using POR techniques that prune away equivalent traces (e.g. sleep sets [Godefroid 1996], persistent sets [Flanagan and Godefroid 2005], source sets and wakeup trees [Abdulla et al. 2014]). 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 3.3 to examine no more traces than any enumerative exploration based on the Mazurkiewicz equivalence. The two examples show $\sim_O$ can offer exponential improvements wrt two parameters: (i) the number of processes, and (ii) the size of each process.

Example 3.5 (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 $w^j_i$ (resp. $r^i$) 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 $w^{3-j}_{\frac{n}{2}}$ or $w^{j}_{\frac{n}{2}-1}$, whereas there are at most $n + 1$ possible write events for $r^j$ to observe (either $w^j_i$ or one of the $w^{3-j}_{\frac{n}{2}}$).

Hence $\mathcal{T}_P/\sim_O$ has size $O(n)$. In contrast, $\mathcal{T}_P/\sim_M$ has size $\Omega((\frac{2^n}{n})^k) = \Omega(2^n)$, 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 3.6 (Many processes of small size). We now turn our attention to a system $\mathcal{P}$ 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 $O(k^2)$ realizable observation functions, by choosing for each one among $k$ read events, one among $k$ write events it can observe. Hence $\mathcal{T}_P/\sim_O$ has size $O(k^2)$. In contrast, the size of $\mathcal{T}_P/\sim_M$ is $\Omega((k!)^2)$. This holds as there are $k!$ ways to
order the k write events, and for each such permutation there are k! ways to assign each of the k read events to the write event that it observes. To see this second part, let w₁, ..., wₖ be any permutation of the write events, and let rᵢ be the read event in the same process as wᵢ. Then rᵢ can be placed right after any wⱼ with i ≤ j. Observe that $T_P/\sim_O$ is exponentially more succinct than $T_P/\sim_M$, as

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

### 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 3.3 is more succinct. In high level, our approach consists of the following steps.

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 [Lamport 1978], 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.

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 $A = (A^+, A^-)$ is a pair of

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

such that for all read events $r$, if $A^+(r) = w$, then we have $\text{Confl}(r, w)$ and it is not the case that $\text{PS}(r, w)$. 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 $O_t = A^+$, 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., $O_t(r) \in A^-(r)$). In the remaining section we focus on positive annotations, and the problem of deciding whether a positive annotation is realizable.

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

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

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

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

Well-formed annotations and basis of annotations. A positive annotation $A^+$ is called well-formed if the following conditions hold:

1. $A^+$ is acyclic.
2. For every lock-release event $e_a \in \text{img}(A^+) \cap \mathcal{L}^A$ there is at most one lock-acquire event $e_r \cap \mathcal{L}^R$ such that $A^+(e_a) = e_r$.
3. There exist sequential traces $(\tau_i)_i$, one for each process $p_i$, such that each $\tau_i$ ends in a global event, and the following conditions hold.
(a) for every pair of lock-acquire events $e_1^A, e_2^A \in E(\tau_i) \cap L^A$ such that $\text{in}_r(e_1^A) < \text{in}_r(e_2^A)$ and $\text{loc}(e_1^A) = \text{loc}(e_2^A)$ there exists a lock release event $e_r \in E(\tau_i) \cap L^R$ such that $\text{in}_r(e_1^A) < \text{in}_r(e_r) < \text{in}_r(e_2^A)$ and $\text{loc}(e_r) = \text{loc}(e_1^A) = \text{loc}(e_2^A)$.

(b) $\bigcup_i R(\tau_i) = \text{dom}(A^+) \text{ and } \text{img}(A^+) \subseteq \bigcup_i W(\tau_i)$, i.e., $(\tau_i)_i$, contains precisely the read events of $A^+$ and a superset of the write events.

(c) Each $\tau_i$ corresponds to a deterministic computation of process $p_i$, where the value of every global event $e$ during the computation is taken to be $\text{val}_{A^+}(e)$.

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

**Lemma 4.1.** Let $X = \text{dom}(A^+) \cup \text{img}(A^+)$ be the set of events that appear in a positive annotation $A^+$, and $X_i = X \cap E_i$ 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 $n = \sum_i |(\tau_i)_i|$ if $A^+$ is well-formed, otherwise $n = \sum_i \ell_i$, where $\ell_i$ 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 $t \in \tau_1 \ast \ldots \ast \tau_k$.

**Proof.** (1) Assume towards contradiction 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 $\text{val}_{A^+}$ as the value function of global events, we have that each $\tau_i$ and $\tau'_i$ share a prefix relationship (i.e., one is prefix of the other). Since the two bases are distinct, for some $j$ we have that one of $\tau_j$ and $\tau'_j$ is a proper prefix of the other. Assume w.l.o.g. that $\tau'_j$ is a strict prefix of $\tau_j$. Then replacing $\tau_j$ with $\tau'_j$ in $(\tau_i)_i$ yields another basis, thus $\tau_j$ is not minimal, a contradiction.

(2) First, testing the conditions of Item 1 and Item 2 of well-formedness can be done in $O(|A^+|)$ time. We now 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 $O(|A^+|)$ 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 $\text{val}_{A^+}$ for assigning values to global events. The execution stops when $e_i$ is reached, and the constructed sequential trace $\tau_i$ is returned. Finally, $(\tau_i)_i$ constitute a basis of $A^+$ if the conditions (a) and (b) of Item 3 of well-formedness are met, which can be done in $O(n)$ time.

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

\[\square\]

### 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 $O_t = A^+$, 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 $O_t = A^+$ 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 \(H = \{(x_i, y_i, z_i)\}_{i=1}^n\) 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 4.2.** ACYCLIC EDGE ADDITION is NP-hard.

**Proof.** The proof is by reduction from MONOTONE ONE-IN-THREE SAT [Garey and Johnson 1979, LO4]. In MONOTONE ONE-IN-THREE SAT, the input is a propositional 3CNF formula \(\phi\) in which every literal is positive, and the goal is to decide whether there exists a satisfying assignment for \(\phi\) 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 \(\phi\). 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 \(C = (x_i \lor x_j \lor \neg x_k)\), we introduce a pair of nodes \(w^C_{ij}, r^C_{ij} \in V\) and an edge \((w^C_{ij}, r^C_{ij}) \in E\), where \(j \in \{1, 2, 3\}\). Additionally, we introduce an edge \((w^C_{ij}, w^D_{ij}) \in E\) for every pair \(j, l \in \{1, 2, 3\}\) such that \(j \neq l\), and an edge \((w^C_{ij}, r^C_{ij})\) 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 \(x_\ell\) in positions \(l_1\) and \(l_2\)), we add edges \((w^C_{ij}, r^D_{ij}), (w^D_{ij}, r^C_{ij}) \in E\). The set \(H\) consists of triplets of nodes \((w^C_{ij}, w^D_{ij}, r^C_{ij})\) 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 \((w'_i, w^C_{ij}, r^C_{ij}), (w'_i, w^D_{ij}, r^D_{ij})\) \(\in H\), we have that \((w'_i, w^C_{ij}) \in X\) iff \((w'_i, w^D_{ij}) \in X\), 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 \((w'_i, w^C_{ij}), (r^D_{ij}, w'_i) \in H\), then \(G_X = (V, E \cup X)\) would contain the cycle \(w'_i \rightarrow w^C_{ij} \rightarrow r^D_{ij} \rightarrow w'_i\) 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 \(\phi\) by setting \(x_i = \text{True}\) if \(X\) contains an edge \((w^C_{ij}, w^D_{ij})\) for some clause \(C\) and \(j \in \{1, 2, 3\}\). By the previous remark, the assignment of values to variables of \(\phi\) is well-defined.

It is easy to verify that the construction takes polynomial time in the size of \(\phi\). 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 \(\phi\) contains exactly one variable set to True. Indeed, \(C\) contains at least one such variable,
otherwise \(G_X\) would contain a cycle

\[
    r_{C_1}^C \rightarrow w'_{C_1} \rightarrow r_{C_2}^C \rightarrow w'_{C_2} \rightarrow r_{C_3}^C \rightarrow w'_{C_3} \rightarrow r_{C_1}^C
\]

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

\[
    w'_i \rightarrow w_i^C \rightarrow w'_j \rightarrow w_j^C \rightarrow w'_i
\]

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

\[
    e_i = \begin{cases} 
        (w'_i, w_i^C), & \text{if } x_i = \text{True} \\
        (r_i^C, w'_i), & \text{if } x_i = \text{False} 
    \end{cases}
\]

Given a clause \(C\), we denote by \(V_C = \bigcup_{i=1}^3 \{w_{C_i}', w_{C_i}^C, r_{C_i}^C\}\), 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 \(C\) be such a cycle.

(1) If \(C\) contains a node of the form \(w_{C_j}^C\) for some \(j \in \{1, 2, 3\}\) then \(C\) traverses the edge \((w_{C_j}^C, w_{C_j}^C)\), and thus \(x_{C_j}\) is assigned True, where \(\ell = C_j\) Since \(G_X\) contains no edge of the form \((r_{C_j}^D, w_{C_j}^C)\) for any clause \(D\), \(C\) must traverse an edge \((w_{D_j}^D, w_{D_j}^C)\) for some clause \(D\). Hence there is a first node \(w_{D_j}^D\) traversed by \(C\) after \(w_{C_j}^C\), for some clause \(D\) and \(i \in \{1, 2, 3\}\) and thus \(G_X\) contains an edge \((w_{D_i}^D, w_{D_i}^C)\), and hence \(x_l\) is assigned True, where \(l = D_i\). By our choice of \(w_{D_i}^D\), the node \(w_{C_j}^C\) can only be reached via the edge \((r_{D_i}^D, w_{D_i}^C)\), which requires that \(x_l\) is assigned False, a contradiction.

(2) If \(C\) contains no node of the form \(w_{C_j}^C\), then it must be a cycle in \(G[V_C]\), for some clause \(C\). The only such cycle that traverses no \(w_{C_j}^C\) can be

\[
    r_{C_1}^C \rightarrow w_{C_1}' \rightarrow r_{C_2}^C \rightarrow w_{C_2}' \rightarrow r_{C_3}^C \rightarrow w_{C_3}' \rightarrow r_{C_1}^C
\]

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

The desired result follows. \(\square\)

**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, w.l.o.g., 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 if 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 \(w_i \in W, r_i \in R\) in \(p_i\) such that \(PS(w_i, r_i)\), and an event \(w'_i\) in process \(p_{|H|+i}\). For all three events we set \(loc(r_i) = loc(w_i) = loc(w'_i) = g_i\), where \(g_i \in G\) is some fresh global variable of \(\mathcal{P}\). Finally, we introduce \((r_i, w_i) \in A^+\). Given any node \(u \in V\), let \(e(u)\) denote the event associated with \(u\).
\[ \phi = (x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_4 \lor x_5) \]

\[ \psi = (x_1 \lor x_2) \land (x_1 \lor x_4) \land (x_1 \lor x_5) \]

**Fig. 7.** The reduction of 3SAT over \( \phi \) to ACYCLIC EDGE ADDITION over \((G, H)\). The nodes and solid edges represent the graph \( G \). The dashed edges represent the triplets in \( H \).

(2) For every edge \((u, v)\) we introduce a new global variable \( g \in G \), and two events \( w_{u,v} \in W \), \( r_{u,v} \in R \) such that \( \text{loc}(w_{u,v}) = \text{loc}(r_{u,v}) = g \). 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 \( PS(e(u), w_{u,v}) \) and \( PS(r_{u,v}, e(v)) \). Finally, we introduce \((r_{u,v}, w_{u,v}) \in A^+\).

Observe that the above construction is linear in the size of \((G, H)\). The following lemma states the correctness of the reduction.

**Lemma 4.3.** 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 \( P \).

**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 \( \text{in}_t(e(u)) < \text{in}_t(e(v)) \), since \( PS(e(u), w_{u,v}) \) and \( PS(r_{u,v}, e(v)) \) and \( A^+(r_{u,v}) = w_{u,v} \). Additionally, \( t \) satisfies that either \( \text{in}_t(w'_i) < \text{in}_t(w_i) \), or \( \text{in}_t(r_i) < \text{in}_t(w'_i) \). In the former case we add \((x_i, y_i)\), 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. \( \square \)

### 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 \( P \) is an acyclic architecture, then the problem admits a polynomial-time solution (in fact, cubic in the size of the constructed trace).

**Intuition.** The hardness of realizing positive annotations in general architectures comes from transitivity constraints that ensure that the resulting happens-before relation is acyclic. That is,
for every triplet of events $e_1, e_2, e_3$, deciding that (i) $e_1$ happens before $e_2$ and (ii) $e_2$ happens before $e_3$ must lead in (iii) $e_1$ happening before $e_3$. In general architectures, such a triplet of events is, in general, unrelated a-priori, and hence an algorithm that constructs a trace out of a positive annotation need to make the above decisions (i)-(iii) consistently. In contrast, acyclic architectures have the property that in every such triplet of events, a pair of them always belongs to the same process and thus is ordered a-priori by the program structure. This allows to express transitivity constraints by means of a 2SAT encoding. For example assume that $e_2, e_3$ belong to the same process and $e_2$ is ordered before $e_3$ by the program structure. Then the transitivity constraints can be simply encoded in a 2SAT clause $(x_{e_2, e_3} \Rightarrow x_{e_3, e_1})$, where $x_{e, e'}$ is interpreted as a boolean variable indicating that $e$ happens before $e'$. Besides transitivity constraints, observe that positive annotation constraints can also be encoded in a 2SAT clause. That is, every positive annotation constraint $A^+(r) = w$ can be encoded in a 2SAT clause $(x_{w'}, r \Rightarrow x_{w', w})$, for every write event $w' \neq w$ that conflicts with $w$.

**Procedure** Realize. Let $P$ be an acyclic architecture, and $A^+$ a positive annotation over $P$. We describe a procedure Realize($A^+$) 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, Realize($A^+$) uses Lemma 4.1 to extract a basis $(\tau_i)_i$ of $A^+$. In the second phase, Realize($A^+$) determines whether the events of $\bigcup_i \mathcal{E}(\tau_i)$ can be linearized in a trace $t$ such that $O_t = A^+$. Informally, the second phase consists of constructing a 2SAT instance over variables $x_{e_i, e_j}$, where $e_i, e_j \in \bigcup_i \mathcal{E}(\tau_i)$. Setting $x_{e_i, e_j}$ to true corresponds to making $e_i$ happen before $e_j$ 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 $G_P$,
2. the produced total order respects the positive annotation, i.e., every write event $w'$ that conflicts with an annotated read/write pair $(r, w) \in A^+$ must either happen before $w$ or after $r$, and
3. 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 4.4.** 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.

**Correctness.** We first argue about correctness.

1. If Realize returns a sequence of events $t$ (Line 27) then clearly $t$ is a trace since $t$ respects the program structure PS (Line 21), and by the definition of the well-formed annotation, lock semantics are respected (i.e., critical regions protected by locks do not overlap). Additionally, $t$ realizes $A^+$, as the sequential consistency axioms are satisfied because of Line 18 and Line 21.

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 $in_t(e_1) < in_t(e_2)$ (Line 21). By the sequential consistency axioms, for every pair $(r, w) \in A^+$ and $w' \neq w$ with $Confl(r, w')$ we have either $in_t(w') < in_t(w)$ or $in_t(r) < in_t(w')$ (Line 18). 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 $C$ is satisfiable. It suffices to argue that $G' = (V', E')$ (Line 26) is acyclic, as then any topological order of $G'$ will satisfy the sequential consistency axioms (Line 18). Assume towards contradiction otherwise. If $G'$
Data-centric Dynamic Partial Order Reduction

Algorithm 1: Realize($A^+$)

Input: A positive annotation $A^+$ with basis $(\tau_1)$.

Output: A trace $t$ that realizes $A^+$ or $\perp$ if $A^+$ is not realizable.

1. Construct a directed graph $G = (V, E)$ where

   $V = \bigcup_{\tau \in \tau} E(\tau)$.

2. $E = \{(e_1, e_2) : (e_2, e_1) \in A^+ \text{ or } \mathsf{PS}(e_1, e_2)\}$

3. $G^* = (V, E^*) \leftarrow$ the transitive closure of $G$

   // A set $C$ of 2SAT clauses over variables $V$

4. $C \leftarrow \emptyset$

5. $V_C \leftarrow \{x_{e_1, e_2} : e_1, e_2 \in V \text{ and } e_1 \neq e_2 \text{ and either } \mathsf{proc}(e_1) = \mathsf{proc}(e_2) \text{ or } (\mathsf{proc}(e_1), \mathsf{proc}(e_2)) \in E_F\}$

   // 1. Antisymmetry clauses

7. foreach $x_{e_1, e_2} \in V_C$ do

   8. $C \leftarrow C \cup \{(x_{e_1, e_2} \Rightarrow \neg x_{e_2, e_1}), (\neg x_{e_2, e_1} \Rightarrow x_{e_1, e_2})\}$

9. end

   // 2. Transitivity clauses

10. foreach $x_{e_1, e_2} \in V_C$ do

11.    foreach $(e_2, e_3) \in E^*$ do

12.       $C \leftarrow C \cup \{(x_{e_1, e_2} \Rightarrow x_{e_1, e_3})\}$

13. end

14. foreach $(e_3, e_1) \in E^*$ do

15.       $C \leftarrow C \cup \{(x_{e_1, e_2} \Rightarrow x_{e_3, e_2})\}$

16. end

   // 3. Annotation clauses

18. foreach $(r, w) \in A^+ \text{ and } w' \in V \cap \tau' W \text{ s.t. } \mathsf{Confl}(r, w')$ do

19.       $C \leftarrow C \cup \{(x_{w', r} \Rightarrow x_{w', w})\}$

20. end

   // 4. Fact clauses

21. foreach $(e_1, e_2) \in E^*$ with $e_1 \neq e_2$ do

22.       $C \leftarrow C \cup \{(x_{e_1, e_2})\}$

23. end

24. Compute a satisfying assignment $f : V_C \rightarrow \{\text{False}, \text{True}\}^{\mid V_C \mid}$ of the 2SAT over $C$, or return $\perp$ if $C$ is unsatisfiable

25. $E' \leftarrow E \cup \{(e_1, e_2) : f(x_{e_1, e_2}) = \text{True}\}$

26. Let $G' = (V, E')$

27. return a trace $t$ by topologically sorting the vertices of $G'$

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 $(e_1, e_2) \in E' \setminus E^*$, hence $f(x_{e_1, e_2}) = \text{True}$. We distinguish the following cases.

(a) If there exists a cycle $C$ which traverses a single edge $(e_1, e_2) \in E' \setminus E^*$, then there is a path $e_2 \leadsto e_1$ traversing only edges of $E^*$. Since $E^*$ is transitively closed, we have that $(e_2, e_1) \in E^*$ and hence $f(x_{e_2, e_1}) = \text{True}$ (Line 21). Thus, by antisymmetry, $f(x_{e_1, e_2}) = \text{False}$ (Line 7), a contradiction.

(b) Otherwise, let $C$ be a simple cycle in $G^*$ that traverses the fewest number of edges in $E' \setminus E^*$, and $C$ must traverse at least two edges $(e_1, e_2), (e_3, e_4) \in E' \setminus E^*$. Observe that
The following theorem summarizes the results of this section.

**Theorem 4.5.** Consider any architecture $\mathcal{P}$ 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 $E' \setminus E^*$ (namely, $e_1 \rightarrow e_2 \rightarrow e_1$ or $e_3 \rightarrow e_4 \rightarrow e_3$). Since $\mathcal{P}$ is acyclic, by construction (Line 6) $|\{\text{proc}(e_1), \text{proc}(e_2), \text{proc}(e_3), \text{proc}(e_4)\}| = 2$, hence, there exist two distinct processes $p_i, p_j$ such that

$$e_1 \in \mathcal{E}_i \quad \text{and} \quad e_2 \in \mathcal{E}_j \quad \text{and} \quad e_3, e_4 \in \mathcal{E}_i \cup \mathcal{E}_j.$$

Observe that either (i) $\text{PS}(e_4, e_1)$ or (ii) $\text{PS}(e_2, e_2)$. We focus on case (i), as case (ii) is similar. Since $\text{PS}(e_4, e_1)$ and $G'$ is transitively closed, we have $(e_4, e_1) \in E^*$. Additionally, since $\text{proc}(e_3) \neq \text{proc}(e_4)$ we have $\text{proc}(e_3) = \text{proc}(e_2)$, and thus $(e_2, e_3) \in E^*$. Since $f(x_{e_3, e_1}) = \text{True}$, by transitivity (Line 10), we have that $f(x_{e_1, e_1}) = \text{True}$ and thus, by transitivity again, $f(x_{e_2, e_1}) = \text{True}$. This leads to a contradiction, as $(e_1, e_2) \in E' \setminus E^*$ and thus $f(x_{e_1, e_2}) = \text{True}$, which violates the transitivity constraint (Line 10).

The desired result follows.

**Complexity.** The transitive closure requires $O(n^3)$ time, since $|V| = n$. The set $V_C$ (Line 6) has $O(n^3)$ 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. Computing a satisfying assignment for $C$ (or concluding that none exists) requires linear time in $|C|$ [Aspvall et al. 1979], hence this step costs $O(n^3)$. Finally, constructing $G'$ and computing a topological sorting of its vertices requires $O(n^3)$ time in total. The desired result follows.

The following theorem summarizes the results of this section.

**Theorem 4.5.** Consider any architecture $\mathcal{P} = (p)$, 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 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 $\text{Past}_t(e)$ 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 $\text{PS}(e', e)$, then $e' \in \text{Past}_t(e)$,
2. if $e_1 \in \text{Past}_t(e)$, for every event $e_2 \in \mathcal{E}(t)$ such that $\text{PS}(e_2, e_1)$, we have that $e_2 \in \text{Past}_t(e)$, and
3. if there exists a read $r \in \text{Past}_t(e)R$, we have that $O_t(r) \in \text{Past}_t(e)$.

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 $e' \in \text{Past}_t(e)$ we have that $e' \rightarrow^* e$, 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 \( O_t(r) \rightarrow_t r \) but possibly \( O_t(r) \notin \text{Past}_t(r) \).

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

**Remark 3.** For every trace \( t \) and event \( e \in \mathcal{E}(t) \) we have that \( t|(\text{Past}_t(e) \cup e) \) is a valid trace.

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 trace \( t_1 \) also appears in another trace \( t_2 \), and the read events in the causal past observe the same write events in both traces, then \( e \) is inevitable in \( t_2 \), i.e., every maximal extension of \( t_2 \) will contain \( e \).

**Lemma 5.1.** Consider two traces \( t_1, t_2 \) and an event \( e \in \mathcal{E}(t_1) \) such that for every read \( r \in \text{Past}_{t_1}(e) \) we have \( r \in \mathcal{E}(t_2) \) and \( O_{t_1}(r) = O_{t_2}(r) \). Then \( e \) is inevitable in \( t_2 \).

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

\[
 e_m = \arg \min_{e' \in (\text{Past}_{t_1}(e) \cup \{e\}) \setminus \mathcal{E}(t'_2)} \text{in}_{t_1}(e')
\]

be the first such event in \( t_1 \), and let \( p_l = \text{proc}(e_m) \) be the process of \( e_m \). By Remark 2, for every event \( e' \in \text{Past}_{t_1}(e_m) \) we have \( e' \in \text{Past}_{t_1}(e) \), and since \( \text{in}_{t_1}(e') < \text{in}_{t_1}(e_m) \), 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 \( p_l \) 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 \( e_m : \text{acquire} \ l \), since \( t'_2 \) is a lock-free trace, \( l \) is released in \( s(t'_2) \), hence \( e_m \in \text{enabled}(t'_2) \) and \( t'_2 \) is not maximal, a contradiction.

2. If \( x \) has at least two outgoing edges then \( e_m \) is of the form \( e_m : b_j(v_1, \ldots, v_n) \), where \( v_i \in \mathcal{V}_i \) are local variables of \( p_l \). Let \( t_{e_m} = t'_2 | \text{Past}_{t_1}(e_m) \). Since for every read \( r \in \text{Past}_{t_1}(e) \) we have \( r \in \mathcal{E}(t_2) \) and \( O_{t_1}(r) = O_{t_2}(r) \), by Remark 2, the same holds for reads \( r \in \text{Past}_{t_1}(e_m) \), i.e., for every read \( r \in \text{Past}_{t_1}(e_m) \) we have \( r \in \mathcal{E}(t_2) \) and \( O_{t_1}(r) = O_{t_2}(r) \). It is easy to see that additionally \( r \in \mathcal{E}(t_{e_m}) \) and \( O_{t_1}(r) = O_{t_{e_m}}(r) \). Thus, we have \( O_{t_{e_m}} \subseteq O_{t_1} \). By Lemma 3.1, we have \( \text{val}_{t_{e_m}}(r) = \text{val}_{t_1}(r) \) for every read \( r \), and since \( p_l \) 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 \( p_l \) 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.

\[ \square \]

### 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 \( A = (A^+, A^-) \), 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 event \( r \in \mathcal{R}(t) \) and a write event \( w \in \mathcal{W}(t) \) such that \( r \notin \text{dom}(A^+) \) and \( \text{ConfRW}(r, w) \). If \( w \notin A^-(r) \), 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 $A^-(r)$. 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 $t$ is some arbitrary maximal trace, and $A = (\emptyset, \emptyset)$ is a pair of empty annotations.

We say that DC-DPOR explores a class of $\mathcal{T}_p / \sim_Q$ when it is called on some annotation input $A = (A^+, A^-)$, 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 (Theorem 4.5). We first state the compactness property, which follows by the use of negative annotations.

**Lemma 5.2 (Compactness).** Consider any two executions of DC-DPOR on inputs $(t_1, A_1)$ and $(t_2, A_2)$. Then $A^+_{t_1} \neq A^+_{t_2}$.

**Proof.** Examine the recursion tree $T$ generated by DC-DPOR, where every node $u$ is labeled with the trace $t_u$ and annotation input $A_u = (A^+_u, A^-_u)$ given to DC-DPOR. Let $x$ and $y$ be the nodes that correspond to inputs $(t_1, A_1)$ and $(t_2, A_2)$ respectively, and we argue that $A_x \neq A_y$. If $x$ is an ancestor of $y$, then when DC-DPOR was executed on input $(t_x, A_x)$, a read $r$ created $A^+_{r, w}$ in Line 3 on the branch from $x$ to the direction of $y$ in $T$, with the property that $r \notin \text{dom}(A^+_x)$. Since the algorithm never removes pairs $(r, w)$ from the positive annotations, we have that $(r, w) \notin A_y$, hence $A_x \neq A_y$. 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 $A^+_{z_x}$ (resp. $A^+_{z_y}$), i.e.,

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

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 a trace \( t_i \) that agrees with \( t \) on the first few read events. Then, Lemma 5.1 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 \( r \rightarrow w \) (if \( w \notin A^-(r) \)), or it has already done so in some earlier steps of the recursion (if \( w \in A^-(r) \)).

**LEMMA 5.3 (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 \( A_u = (A^+_u, A^-_u, A^=u) \) 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 4.4 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 \( A^+_u \subseteq O \).
2. If \( A^+_u \subseteq O \), then the traversal proceeds either
   a) to a node \( v \) of \( T \) with \( A^+_u \subseteq A^+_v \),
   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 \( A^+_u \) on more reads than \( A^+_v \). Since Item 2b is executed a finite number of times, any such traversal is guaranteed to reach a node \( w \) with \( A^-_w = O \).

The traversal starts from \( u \) being the root of \( T \), and Item 1 holds as then \( A^+_u = \emptyset \subseteq O \). 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 4.4 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.,

\[
r = \arg \min_{r' \in \mathcal{E}(t_i) \cap \text{dom}(A^-_u)} \text{in}_{t_i}(r')
\]

and \( w = O_{t^*}(r) \). Then for every \( r' \in \text{Past}_{t^*}(r) \), we have \( \text{in}_{t^*}(r') < \text{in}_{t^*}(r) \) and thus \( r' \in \mathcal{E}(t_i) \) and \( O_{t^*}(r') = O_{t^*_i}(r') \). By Lemma 5.1, we have \( r \in \mathcal{E}(t_i) \). Since \( \text{in}_{t^*_i}(w) < \text{in}_{t^*_i}(r) \), a similar argument yields that \( w \in \mathcal{E}(t_i) \). We now distinguish two cases, based on whether \( w \in A^-_u(r) \) or not.

1. If \( w \notin A^-_u(r) \), then in Line 4 the algorithm will generate a trace \( t' \) that is compatible with the strengthened annotation \( A^+_{u'} \cup (r, w) \), and call itself recursively on some child \( v \) of \( u \) with \( A^+_v = A^+_u \cup (r, w) \). The traversal proceeds to \( v \) and Item 1 holds, as desired.
2. If \( w \in A^-_u(r) \), then there exists a highest ancestor \( x \) of \( u \) in \( T \) where DC-DPOR was called with \((r, w) \in A^-_x \). Following Line 7, this can only have happened if \( x \) has a sibling \( v \) in \( T \) with \((r, w) \in A^+_v \). Let \( z \) be the parent of \( x, v \), and we have \( A^+_x \subseteq A^+_u \subseteq O \), and \( A^+_z = A^+_x \cup (r, w) \subseteq O \).
Thus, the traversal proceeds to \( v \) and Item1 holds, as desired. In this case, we say that the traversal \textit{backtracks to} \( x \) \textit{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 \( \text{dom}(A^+_x) \setminus \text{dom}(A^+_z) \). By Line 1, we have that \( \text{in}_t(r_{i+1}) < \text{in}_t(r_i) \), 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. \( \square \)

We thus arrive to the following theorem.

**Theorem 5.4.** 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 \( \mathcal{T}_\mathcal{P} / \sim_O \) exactly once, and requires \( O \left( |\mathcal{T}_\mathcal{P} / \sim_O| \cdot n^5 \right) \) time.

**Proof.** Lemma 5.2 and Lemma 5.3 guarantee the optimality of DC-DPOR, i.e., that DC-DPOR explores each class of \( \mathcal{T}_\mathcal{P} / \sim_O \) 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 4.5, 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. \( \square \)

We note that our main goal is to explore the exponentially large \( \mathcal{T}_\mathcal{P} / \sim_O \) by spending polynomial time in each class. The \( n^5 \) factor in the bound comes from a crude complexity analysis.

## 6 Beyond Acyclic Architectures

In the current section we turn our attention to cyclic architectures. Recall that according to Theorem 4.5, procedure \textit{Realize} is guaranteed to find a trace that realizes a positive annotation \( A^+ \), provided that the underlying architecture is acyclic. Here we show that the trace space of cyclic architectures can be partitioned wrt an equivalence that is finer than the observation equivalence, but remains (possibly exponentially) coarser than the Mazurkiewicz equivalence. The current section makes a formal treatment of cyclic architectures with the aim to prove that exponentially coarser equivalences can be used to guide the search. We refer to our implementation in Section 6 for a description of how cyclic architectures are handled in practice.

**Intuition.** Recall that the architecture of the concurrent system is an acyclic graph, where nodes represent the processes of the system, and two nodes are connected by an edge if the respective processes communicate over a common shared variable. In high level, our approach for handling cyclic architectures consists of the following steps.

1. We choose a set of edges \( X \) such that removing all edges in \( X \) makes the architecture acyclic. Such a choice can be made arbitrarily.
2. For every variable that is used by at least two processes which have an edge in \( X \), we introduce a fresh lock in the system.
3. We transform the concurrent program so that every write event to every such variable is protected by its respective lock.
Consider a cyclic architecture. For any two traces \( t_1 \), \( t_2 \) following hold:

1. Let \( Y = \bigcup_{(p_i, p_j) \in X} \lambda_p(p_i, p_j) \) be the set of variables that appear in edges of the set \( X \). We introduce a set of new locks \( L^O \) in \( P^X \) such that we have exactly one new lock \( l_g \in L^O \) for each variable \( g \in Y \).
2. For every process \( p_i \), every write event \( w \in W_i \) with \( \text{loc}(w) \in Y \) is surrounded by an acquire/release pair on the new lock variable \( l_{\text{loc}(w)} \).

**Observation equivalence refined by an edge set.** Consider a cyclic architecture \( P \) and an edge set of the underlying communication graph \( G_P \). We define a new equivalence on the trace space \( T_P \) as follows. Two traces \( t_1, t_2 \in T_P \) are observationally equivalent refined by \( X \), denoted by \( \sim^X_O \), if:

1. \( t_1 \sim t_2 \), and
2. for every edge \((p_i, p_j) \in X\), for every pair of distinct write events \( w_1, w_2 \in W(t_1) \cap (W_i \cup W_j) \) with \( \text{loc}(w_1) = \text{loc}(w_2) = g \) and \( g \in \lambda_p(p_i, p_j) \), we have that \( \text{in}_{t_1}(w_1) < \text{in}_{t_2}(w_2) \) iff \( \text{in}_{t_1}(w_2) < \text{in}_{t_2}(w_1) \).

Clearly, \( \sim^X_O \) refines the observation equivalence \( \sim_O \). The following lemma captures that the Mazurkiewicz equivalence refines the observation equivalence refined by an edge set \( X \).

**Lemma 6.1.** For any two traces \( t_1, t_2 \in T_P \), if \( t_1 \sim_M t_2 \) then \( t_1 \sim^X_O t_2 \).

**Proof.** By Theorem 4.5, we have \( t_1 \sim_O t_2 \). Consider any pair of distinct write events \( w_1, w_2 \in W(t_1) \cap (W_i \cup W_j) \) with \( \text{loc}(w_1) = \text{loc}(w_2) = g \) and \( g \in \lambda_p(p_i, p_j) \), 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 \( \text{in}_{t_1}(w_1) < \text{in}_{t_2}(w_2) \) iff \( \text{in}_{t_1}(w_2) < \text{in}_{t_2}(w_1) \), as desired.

**Exponential succinctness of \( \sim^X_O \) in cyclic architectures.** Here we present a very simple cyclic architecture where the observation equivalence \( \sim^X_O \) refined by an all-but-two cycle set \( X \) is exponentially more succinct than the Mazurkiewicz equivalence \( \sim_M \). Consider the architecture \( P \) 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 \( G_P \). We argue that \( \sim^X_O \) is exponentially more succinct than \( \sim_M \) by showing exponentially many traces which are pairwise equivalent under \( \sim^X_O \) but not under \( \sim_M \). Indeed, consider the set \( T \) which consists of all traces such that the following hold:

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 $|T| = \binom{2n}{n}$ 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, $T_P/\sim_M^X$ is only exponentially large, and since

$$|(T_P/\sim_M^X) \setminus (T_P/\sim_X)| \geq |T| - 1$$

we have that $\sim_X$ is exponentially more succinct than $\sim_M$.

**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 $P$, called DC-DPOR-Cyclic. First, we determine an all-but-two cycle set $X$ of the underlying communication graph $G_P = (V_P, E_P, \lambda_P)$, and construct the acyclic reduction $P^X$ of $P$ over $X$. The set $X$ can be chosen arbitrarily, e.g. by letting $|X| = |E_P| - 2$ (i.e., adding in $X$ all the edges of $G_P$ except for two). Then, we execute DC-DPOR on $P^X$, with the following two modifications on the procedure $\text{Realize}$.

1. Consider the graph $G = (V, E)$ constructed in Line 1 of $\text{Realize}$ (Algorithm 1). For every pair of write events $w, w'$ protected by some of the new locks $\ell \in L^O$, for every read event $r$ such that $A^+(r) = w$, if $(w, w') \in E$ then we add an edge $(r, w)$ in $E$, and if $(w', r) \in E$, then we add an edge $(w', w) \in E$.

2. If at the end of Item 1 $G$ has a cycle, $\text{Realize}$ returns $\perp$.

3. In Line 6 we use the edge set $E_{\perp X} \setminus X$. Hence for every variable $x_{e_1, e_2}$ used in the 2SAT reduction, we have either $\text{proc}(e_1) = \text{proc}(e_2)$ or $(\text{proc}(e_1), \text{proc}(e_2)) \in E_{\perp X} \setminus X$.

**Lemma 6.2.** Given a well-formed positive annotation $A^+$ over a basis $(\tau_i)_i$, and the modified communication graph $G'_{p\perp X} = (V_{\perp X}, E_{\perp X} \setminus X, \lambda_{\perp X})$, $\text{Realize}$ 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 new locks $L^O$, $A^+$ already induces a total order on the lock-acquire and lock-release events that access the same lock $l_q \in L^O$. Hence $A^+$ already induces a total order between the write events that are protected by the same lock $l_q \in L^O$. 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. The crucial property is that since $X$ is an all-but-two cycle set of $G_P$, the transitivity (Line 10) and antisymmetry (Line 7) clauses ensure that a satisfying assignment preserves acyclicity of the graph $G'$ constructed in Line 26. The complexity analysis is similar to Lemma 4.4. □

| Process $p_1$ : | Process $p_2$ : | Process $p_3$ :
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. write $x$</td>
<td>1. write $x$</td>
<td>1. write $y$</td>
</tr>
<tr>
<td>2. read $x$</td>
<td>2. write $y$</td>
<td>2. write $y$</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n$ + 2. write $y$</td>
<td>$n$ + 2. write $y$</td>
<td>$n$ + 2. write $y$</td>
</tr>
<tr>
<td>$n$ + 3. read $y$</td>
<td>$n$ + 3. read $y$</td>
<td>$n$ + 3. read $y$</td>
</tr>
<tr>
<td>$n$ + 4. read $x$</td>
<td>$n$ + 4. read $x$</td>
<td>$n$ + 4. read $x$</td>
</tr>
</tbody>
</table>

Fig. 8. A cyclic architecture of three processes.
We arrive at the following theorem.

**Theorem 6.3.** Consider a concurrent architecture $\mathcal{P}$ of processes on an acyclic state space, and $n = \max_{t \in \mathcal{T}_P} |t|$ the maximum length of a trace of $\mathcal{P}$. Let $X$ be an all-but-two cycle set of the communication graph $G_P$. The algorithm DC-DPOR-Cyclic explores each class of $\mathcal{T}_P / \sim^X_O$ exactly once, and requires $O\left(\left|\mathcal{T}_P / \sim^X_O \right| \cdot n^5\right)$ time.

**Proof of Theorem 6.3.** We argue that DC-DPOR-Cyclic is then optimal for the cyclic architecture $\mathcal{P}$ wrt the equivalence $\sim^X_O$.

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 5.2 guarantees that $A^+_1 \neq A^+_2$. 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 $A^+_1 \neq A^+_2$, 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 $r \in E(t_1)$ such that $(r, O_{t_1}(r)) < O_{t_2}$, in which case two different classes of $\sim^X_O$ are explored. Since $\sim^X_O$ refines $\sim_O$ it follows that two different classes of $\sim_O^X$ are explored.

   b. There is a lock-acquire event $e_a \in E(t_1)$ such that $(e_a, O_{t_1}(e_a)) < O_{t_2}$. 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 $\sim^X_O$.

2. *(Completeness).* Lemma 6.2 together with the completeness statement of Lemma 5.3 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 $O = A^+$. Since the lock-acquire and lock-release events 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. □

Theorem 6.3 establishes that for cyclic architectures, DC-DPOR-Cyclic explores a partitioning of the trace space that is coarser than the Mazurkiewicz partitioning, and spends only polynomial time per class. We refer to our implementation in Section 6 for a description of how cyclic architectures are handled in practice.

### 7 EXPERIMENTS

Here we report on the implementation and experimental evaluation of our data-centric DPOR algorithm.

#### 7.1 Implementation Details

**Implementation.** We have implemented our data-centric DPOR in C++, by extending the tool Nidhugg\(^1\). Nidhugg is a powerful tool that utilizes the LLVM compiler infrastructure, and hence our treatment of programs is in the level of LLVM’s intermediate representation (IR). Concurrent architectures are supported via POSIX threads.

\(^1\)https://github.com/nidhugg/nidhugg
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 $\text{table}[i]$). 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 $\text{val}_{A^+}$, and thus in every sequential trace $\tau_i$ of the basis $(\tau_i)_i$ of $A^+$ these values are also known. Hence, arrays are handled naturally by the dynamic flavor of the exploration.

Handling cyclic architectures. In order to effectively handle cyclic architectures, we followed the following process. Wlog, we considered that the input architecture always has the most difficult topology, namely it is a clique. First, the cyclic architecture is converted to a star architecture, by choosing some distinguished process $p_1$ as the root of the star, and the remaining processes $p_2, \ldots, p_k$ are the leaves. Recall that a a positive annotation yields a sequential trace for each process. We use the Mazurkiewicz equivalence to generate all possible Mazurkiewicz-based interleavings between traces of the leaf processes, and our observation equivalence to generate all possible observation-based interleavings between the root and every leaf process. Hence the observation equivalence is wrt the star sub-architecture, which is acyclic, and thus our techniques from Theorem 5.4 are applicable. We note that since the Mazurkiewicz interleavings always have to be generated among sequential traces (i.e., straight-line programs), we are generating them optimally (i.e., obtaining exactly one trace per Mazurkiewicz class) easily, using vector clocks [Mattern 1989]. See Figure 9 for an illustration.

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 $(r', O_t(r'))$, where $r' \in \text{Past}_t(r) \cup \text{Past}_t(w)$ 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 $w = O_t(r)$, 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.
7.2 Experimental Results

We now turn our attention to the experimental results. Our comparison is with the Source-DPOR algorithm from [Abdulla et al. 2014] and the tool Nidhugg that implements it [Abdulla et al. 2015]. To our knowledge, Source-DPOR is the latest and state-of-the-art DPOR which has implemented for C programs.

**Experimental setup.** In our experiments, we have compared our data-centric DPOR, with the Mazurkiewicz-based Source-DPOR introduced recently in [Abdulla et al. 2014] as an important improvement over the traditional DPOR [Flanagan and Godefroid 2005]. Our benchmark set consists of synthetic benchmarks, as well as benchmarks obtained from the TACAS Competition on Software Verification (SV-COMP). Most of the benchmarks have tunable size, by specifying a loop-unroll bound, or the number of threads running in parallel. In all cases, we compared the running time and number of traces explored by DC-DPOR and Source-DPOR. We have set a timeout of 1 hour. All benchmarks were executed on an Ubuntu-based virtual machine, given 4GB of memory and one 2GHz CPU.

**Two synthetic benchmarks.** First we analyze the two synthetic benchmarks `lastzero` and `opt_lock` found in Table 3a and Table 3b, respectively. The benchmark `lastzero` was introduced in [Abdulla et al. 2014] to demonstrate the superiority of Source-DPOR over the traditional DPOR from [Flanagan and Godefroid 2005]. It consists of $n$ threads writing to an array, and 1 thread reading from it. We observe that our DC-DPOR explores exponentially fewer traces than Source-DPOR. In fact, the number of traces explored by our data-centric approach scales polynomially, whereas the number explored by the Mazurkiewicz-based approach grows exponentially with the number of threads. Consequently, our DC-DPOR runs much faster, and manages to scale on larger input sizes. We note that the number of traces explored from Source-DPOR differs from the number reported in [Abdulla et al. 2014]. This is natural as the implementation of [Abdulla et al. 2014] handles programs written in Erlang, a functional language with concurrency mechanisms much different from C.

The benchmark `opt_lock` mimics an optimistic locking scheme of 2 threads. Each thread tries to update some variable, and afterwards checks if it was interrupted. If not, it terminates, otherwise it tries again, up to a total $n$ number of attempts. Again, we see that the number of explored traces by DC-DPOR grows polynomially, whereas the number explored by Source-DPOR grows exponentially. Hence, our algorithm manages to handle much larger input sizes than the Mazurkiewicz-based Source-DPOR. Recall that, as Theorem 5.4 states, this exponential reduction in the explored traces comes with polynomial-time guarantees per trace.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Traces</th>
<th>Time (s)</th>
<th>DC-DPOR</th>
<th>S-DPOR</th>
<th>DC-DPOR</th>
<th>S-DPOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>lastzero(4)</td>
<td>38</td>
<td>2,118</td>
<td>0.21</td>
<td>0.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastzero(5)</td>
<td>113</td>
<td>53,172</td>
<td>0.34</td>
<td>19.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastzero(6)</td>
<td>316</td>
<td>1,765,876</td>
<td>0.63</td>
<td>856</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastzero(7)</td>
<td>937</td>
<td>-</td>
<td>1.8</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastzero(8)</td>
<td>12,190</td>
<td>-</td>
<td>9.32</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastzero(9)</td>
<td>52,841</td>
<td>-</td>
<td>47.97</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Experiments on lastzero(n), for n + 1 threads. '-' indicates a timeout after 1 hour.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Traces</th>
<th>Time (s)</th>
<th>DC-DPOR</th>
<th>S-DPOR</th>
<th>DC-DPOR</th>
<th>S-DPOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt_lock(12)</td>
<td>141</td>
<td>785,674</td>
<td>0.35</td>
<td>252.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opt_lock(13)</td>
<td>153</td>
<td>2,056,918</td>
<td>0.36</td>
<td>703.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opt_lock(14)</td>
<td>165</td>
<td>5,385,078</td>
<td>0.43</td>
<td>1,880.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opt_lock(15)</td>
<td>177</td>
<td>-</td>
<td>0.46</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opt_lock(50)</td>
<td>597</td>
<td>-</td>
<td>5.91</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opt_lock(100)</td>
<td>1,197</td>
<td>-</td>
<td>43.82</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opt_lock(200)</td>
<td>2,397</td>
<td>-</td>
<td>450.99</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Experiments on opt_lock(n), where n is the number of attempts to optimistically lock. '-' indicates a timeout after 1 hour.

Table 3. Experimental results on two synthetic benchmarks.

---

**lastzero(n): n + 1 processes**

Globals: int array[n + 1]

```c
// --------- Process j = 0 ---------
Locals : int i
1 i ← n
2 while array[i] ≠ 0 do
3 │ i ← i - 1
4 end

// --------- Process 1 < j ≤ n ---------
5 array[j] ← array[j - 1] + 1
```

**opt_lock(n): 2 processes and n attempts**

Globals: int last_id, x

```c
// --------- Process 0 < j < 2 ---------
Locals : int i
1 i ← 0
2 while i < n do
3 │ i ← i + 1
4 │ last_id ← j
5 │ x ← get_message(j)
6 │ if last_id = j then
7 │ │ return x
8 │ end
9 │ return -1
```

---

**Benchmarks from SV-COMP.** We now turn our attention to benchmarks from SV-COMP, namely `fib_bench`, `pthread_demo`, `sigma_false` and `parker`, which are found in Table 4a, Table 4b, Table 4c and Table 4d, respectively. Similarly to our findings on the synthetic benchmarks, the data-centric DC-DPOR manages to explore fewer traces than the Mazurkiewicz-based Source-DPOR. In almost all cases, our algorithm run much faster, offering exponential gains in terms of time. One exception is the benchmark `parker`, where our DC-DPOR is slower. Although the number of traces explored is less than that of Source-DPOR, the latter method managed to spend less time in discovering each trace, which led to a smaller overall time. We note, however, that the improvement of Source-DPOR over DC-DPOR appears to grow only as a small polynomial wrt the input size. Recall that new traces are discovered by DC-DPOR using the procedure Realize, which can take cubic time in the worst case (Theorem 4.5). Hence, we identify optimizations to Realize as an important challenge that will contribute further to the scalability of our approach.

---

**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 [Cadiou and Lévy 1973; Clarke et al. 1986; Farzan and Kincaid 2012; Farzan and Madhusudan 2009; Lal and Reps 2009; Lipton 1975; Petri 1962]. The hardness of
The table shows experimental results on four benchmarks from SV-COMP.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Traces</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC-DPOR</td>
<td>S-DPOR</td>
<td>DC-DPOR</td>
</tr>
<tr>
<td>fib_bench(4)</td>
<td>1,233</td>
<td>19,605</td>
</tr>
<tr>
<td>fib_bench(5)</td>
<td>8,897</td>
<td>218,243</td>
</tr>
<tr>
<td>fib_bench(6)</td>
<td>70,765</td>
<td>2,364,418</td>
</tr>
</tbody>
</table>

(a) Experiments on fib_bench(n), where n is the loop-unroll bound.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Traces</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC-DPOR</td>
<td>S-DPOR</td>
<td>DC-DPOR</td>
</tr>
<tr>
<td>pthread_demo(8)</td>
<td>256</td>
<td>12,870</td>
</tr>
<tr>
<td>pthread_demo(10)</td>
<td>1,024</td>
<td>184,756</td>
</tr>
<tr>
<td>pthread_demo(12)</td>
<td>4,096</td>
<td>2,704,156</td>
</tr>
</tbody>
</table>

(b) Experiments on pthread_demo(n), where n is the loop-unroll bound.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Traces</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC-DPOR</td>
<td>S-DPOR</td>
<td>DC-DPOR</td>
</tr>
<tr>
<td>sigma_false(6)</td>
<td>16</td>
<td>10,395</td>
</tr>
<tr>
<td>sigma_false(7)</td>
<td>22</td>
<td>135,135</td>
</tr>
<tr>
<td>sigma_false(8)</td>
<td>29</td>
<td>2,027,025</td>
</tr>
<tr>
<td>sigma_false(9)</td>
<td>37</td>
<td>-</td>
</tr>
<tr>
<td>sigma_false(10)</td>
<td>46</td>
<td>-</td>
</tr>
</tbody>
</table>

(c) Experiments on sigma_false(n), where n is the loop-unroll bound. `-` indicates a timeout after 1 hour.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Traces</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC-DPOR</td>
<td>S-DPOR</td>
<td>DC-DPOR</td>
</tr>
<tr>
<td>parker(8)</td>
<td>1,254</td>
<td>3,343</td>
</tr>
<tr>
<td>parker(10)</td>
<td>2,411</td>
<td>6,212</td>
</tr>
<tr>
<td>parker(12)</td>
<td>4,132</td>
<td>10,361</td>
</tr>
<tr>
<td>parker(14)</td>
<td>6,529</td>
<td>16,022</td>
</tr>
<tr>
<td>parker(16)</td>
<td>9,714</td>
<td>23,427</td>
</tr>
</tbody>
</table>

(d) Experiments on parker(n), where n is the loop-unroll bound.

Table 4. Experimental results on four benchmarks from SV-COMP.

Reproducing bugs by testing, due to scheduling non-determinism, makes model checking a very relevant approach [Alglave et al. 2013; Andrews et al. 2004; Clarke et al. 1999a; Godefroid 2005; Musuvathi and Qadeer 2007], 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 [Godefroid 1997], several reduction techniques have been proposed such as POR and context bounding [Musuvathi and Qadeer 2007; Peled 1993]. Several POR methods, based on persistent set [Clarke et al. 1999b;b; Godefroid 1996; Valmari 1991] and sleep set techniques [Godefroid 1997], have been explored. DPOR [Flanagan and Godefroid 2005] presents on-the-fly construction of persistent sets, and several variants and improvements have been considered [Lauterburg et al. 2010; Saarikivi et al. 2012; Sen and Agha 2006, 2007; Tasharofi et al. 2012]. In [Abdulla et al. 2014], source sets and wakeup trees 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 [Kahlon et al. 2009] using SAT or SMT-solvers; or using unfoldings for optimal reduction in number of interleavings [Kähkönen et al. 2012; McMillan 1995; Rodríguez et al. 2015]. Techniques for transition-based POR for message passing programs have also been considered [Godefroid 1996; Godefroid et al. 1995; Katz and Peled 1992], and some works extend POR to relaxed memory models [Abdulla et al. 2015; Wang et al. 2008].

Another direction of DPOR is SAT/SMT-based, such as Maximal Causality Reduction (MCR) [Huang 2015], and SATCheck [Demsky and Lam 2015]. Such techniques require an NP oracle to guide each step of the search (i.e., a SAT/SMT solver), and thus suffer scalability issues. On the other hand, they have the possibility of exploring fewer traces than the traditional DPOR methods. Among these works, MCR [Huang 2015] is closer to ours, hence we make a more extensive reference to it.

Comparison with [Huang 2015]. Maximal Causality Reduction is a form of partial-order reduction that is based on coarsening the Mazurkiewicz equivalence. The main principle is to try and explore traces in which read events observe different values (as opposed to different write events as in our case). As a result, MCR can potentially create an equivalence that is coarser than the
observation equivalence that we introduce here. However, the MCR approach is very different from ours, in the following three important aspects.

(1) MCR uses an SMT solver to explore each class of the partitioning. In other words, the MCR relies on an NP-oracle, which has exponential worst-case complexity, even for exploring a single class of the partitioning. In contrast, our approach spends provably polynomial time per class. We also note that the experimental results of [Huang 2015, Page 9] identify the SMT procedure as the bottleneck of the whole approach, and ask for an efficient method for each explored trace.

(2) MCR is not optimal wrt to its partitioning. In fact, many equivalence classes of the partitioning can be explored exponentially many times. We provide here a minimal example. Consider the program depicted in Figure 10a. We have two processes $p_1, p_2$, and two global variables $x, y$. The first processes is $p_1 = w_1^x r_1^x$, and the second process is $p_2 = w_2^x r_2^y$. Additionally, we let $w_0^x$ and $w_0^y$ denote the two initialization write events.

![Fig. 10. A simple concurrent system of two processes, and the corresponding MCR exploration.](image)

We now consider the MCR exploration of the above system.

(a) Initially, the process starts in node $a$ with an empty seed interleaving. The process will generate two seed interleavings $\{b, c\}$, forcing $r_1^x$ to observe the value of $w_2^x$, and $r_2^y$ to observe the value of $w_1^y$, respectively.

(b) When in b, the process will create the seed interleaving $\{d\}$, forcing $r_2^y$ to observe the value of $w_1^y$.

(c) When in c, the process will create the seed interleaving $\{e\}$, forcing $r_1^x$ to observe the value of $w_2^x$.

Hence, all traces represented in the leaves $\{d, e\}$ of the above tree have the same observation function (and thus all reads observe the same values). In contrast, the optimality of our algorithm guarantees that the exploration will never explore both $d$ and $e$.

The example can easily be generalized to one where the MCR will explore the same class exponentially many times. The only principle necessary to make the example work is that different branches of the recursion can accumulate the same read-to-write observations in different order, leading to the same read-to-write observations overall.

Hence, compared to our approach, MCR suffers an exponentiation in complexity in two parts: (i) visiting each class of the partitioning exponentially many times, and (ii) using an NP-oracle with exponential worst-case behavior for each visit.
9 CONCLUSIONS

We introduce the new observation equivalence on traces that refines the Mazurkiewicz equivalence and can even 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. 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, such as using static analysis to obtain the best way for transforming cyclic architectures to acyclic.

ACKNOWLEDGMENTS

The research was partly supported by Austrian Science Fund (FWF) Grant No P23499- N23, FWF NFN Grant No S11407-N23 (RiSE/SHiNE), ERC Start grant (279307: Graph Games), and Czech Science Foundation grant GBP202/12/G061.

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In TACAS.
Azadeh Farzan and Zachary Kincaid. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. In CAV.
Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In PLDI.


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


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


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


