

# Data-Centric Dynamic Partial Order Reduction

MAREK CHALUPA, Masaryk University, Czech Republic

KRISHNENDU CHATTERJEE, Institute of Science and Technology, Austria, Austria

ANDREAS PAVLOGIANNIS, Institute of Science and Technology, Austria, Austria

NISHANT SINHA, Kena Labs, India

KAPIL VAIDYA, Indian Institute of Technology, Bombay, India

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.

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

Additional Key Words and Phrases: Partial-order Reduction, Concurrency, Stateless model-checking

## ACM Reference Format:

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (January 2018), 30 pages. <https://doi.org/10.1145/3158119>

Authors' addresses: Marek Chalupa, Masaryk University, Brno, Czech Republic, [mchalupa@mail.muni.cz](mailto:mchalupa@mail.muni.cz); Krishnendu Chatterjee, Institute of Science and Technology, Austria, Am Campus 1, Klosterneuburg, 3400, Austria, [krishnendu.chatterjee@ist.ac.at](mailto:krishnendu.chatterjee@ist.ac.at); Andreas Pavlogiannis, Institute of Science and Technology, Austria, Am Campus 1, Klosterneuburg, 3400, Austria, [pavlogiannis@ist.ac.at](mailto:pavlogiannis@ist.ac.at); Nishant Sinha, Kena Labs, India, [nishantsinha@acm.org](mailto:nishantsinha@acm.org); Kapil Vaidya, Indian Institute of Technology, Bombay, IIT Area, Powai, Mumbai, 400076, India.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2018 Copyright held by the owner/author(s).

2475-1421/2018/1-ART31

<https://doi.org/10.1145/3158119>

## 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 CHES [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

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

Fig. 1. A system of two processes with two events each.

events. The traditional DPOR based on the Mazurkiewicz equivalence  $\sim_M$  will explore at least one representative trace from every class induced on the trace space by the Mazurkiewicz equivalence. There exist  $\frac{2^3}{2} = 4$  possible orderings of dependent events, as there are  $2^3$  possible interleavings, but half of those reorder the independent events  $r_1, r_2$ , and thus will not be considered. The traditional DPOR will explore the following four traces.

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

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

## 1.2 Our Contributions

In this work our contributions are as follows.

**Observation equivalence.** We introduce a new notion of *observation equivalence* (Section 3.1), which is intuitively as follows: An observation function of a trace maps every read event to the write event it observes under sequentially 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.

Due to lack of space, full proofs can be found in full version of this paper [Chalupa et al. 2017].

## 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 *deterministic*, meaning that at any given time there is at most one event that each process can execute. Given the current state of the system, a scheduler chooses one process to execute a sequence of events that is invisibly maximal, that is, the sequence does not end while an invisible event from that process can be taken. The processes communicate by writing to and reading from the global variables. The system can exhibit nondeterministic behavior which is solely attributed to the scheduler, by choosing nondeterministically the next process to take an invisibly maximal sequence of events from any given state. We consider *locks* as the only synchronization primitive, with the available operations being acquiring a lock and releasing a lock. Since richer synchronization primitives are typically built using locks, this consideration is not restrictive, and helps with keeping the exposition of the key ideas simple.

## 2.1 Concurrent Computation Model

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

**Relations and equivalence classes.** A binary relation  $\sim$  on a set  $X$  is an equivalence relation iff  $\sim$  is reflexive, symmetric and transitive. Given an equivalence  $\sim_R$  and some  $x \in X$ , we denote by  $[x]_R$  the equivalence class of  $x$  under  $\sim_R$ , i.e.,

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

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

**Notation on functions.** We write  $f : X \rightarrow Y$  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  $f \neq g$ .

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

- (1)  $e : v \leftarrow \text{read } g$ , in which case  $e \in \mathcal{R}$ ,
- (2)  $e : g \leftarrow \text{write } f(v_1, \dots, v_{n_i})$ , 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)  $e_1 : b(v_1, \dots, v_{n_i})$ .

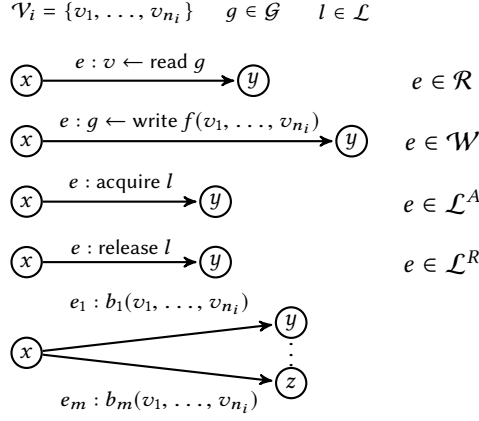


Fig. 2. The control-flow graph  $\text{CFG}_i$  is a sequential composition of these five atomic graphs.

Each  $\text{CFG}_i$  is a directed acyclic graph with a distinguished *root* node  $r_i$ , such that there is a path  $r_i \rightsquigarrow x$  to every other node  $x$  of  $\text{CFG}_i$ . Each node  $x$  of  $\text{CFG}_i$  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  $(x, y_1), \dots, (x, y_m)$  labeled with events  $e_j : b_j(v_1, \dots, v_{n_i})$  of Item 5, and such that for all values of  $v_1, \dots, v_{n_i}$ , we have  $b_j(v_1, \dots, v_n) \implies \neg b_l(v_1, \dots, v_n)$  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  $\text{CFG}_i$ , the node  $y$  is not branching. Indeed, such edges can be easily contracted in a preprocessing phase. Figure 2 provides a summary of the model syntax. We let  $\mathcal{E}_i \subseteq \mathcal{E}$  be the set of events that appear in  $\text{CFG}_i$  of process  $p_i$ , and similarly  $\mathcal{R}_i \subseteq \mathcal{R}$  and  $\mathcal{W}_i \subseteq \mathcal{W}$  the sets of read and write events of  $p_i$ . Additionally, we require that  $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$  for all  $i \neq j$  i.e., all  $\mathcal{E}_i$  are pairwise disjoint, and denote by  $\text{proc}(e)$  the process of event  $e$ . The *location* of an event  $\text{loc}(e)$  is the unique global variable it involves. Given two events  $e, e' \in \mathcal{E}_i$  for some  $p_i$ , we write  $\text{PS}(e, e')$  if there is a path  $e \rightsquigarrow e'$  in  $\text{CFG}_i$  (i.e., we write  $\text{PS}(e, e')$  to denote that  $e$  is ordered before  $e'$  in the *program structure*).

We distinguish a set of *initialization events*  $\mathcal{W}^I \subseteq \mathcal{W}$  with  $|\mathcal{W}^I| = |\mathcal{G}|$  which are attributed to process  $p_1$ , and are used to initialize all the global variables to some fixed values. For every initialization write event  $w^I$  and for any event  $e \in \mathcal{E}_i$  of process  $p_i$ , we define that  $\text{PS}(w^I, e)$  (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  $\text{CFG}_i$  (i.e., the program counter) and  $\text{val}_i$  is a valuation on the local variables  $\mathcal{V}_i$ . A *global state* of  $\mathcal{P}$  is a tuple  $s = (\text{val}, s_1, \dots, s_k)$ , where  $\text{val}$  is a valuation on the global variables  $\mathcal{G}$  and  $s_i$  is a local state of process  $p_i$ . An event  $e$  along an edge  $(x, y)$  of a process  $p_i$  is *enabled* in  $s$  if  $s_i = (x, \text{val}_i)$  (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  $e : b_j(v_1, \dots, v_{n_i})$ , then  $b_j(\text{val}_i(v_1), \dots, \text{val}_i(v_{n_i})) = \text{True}$ .

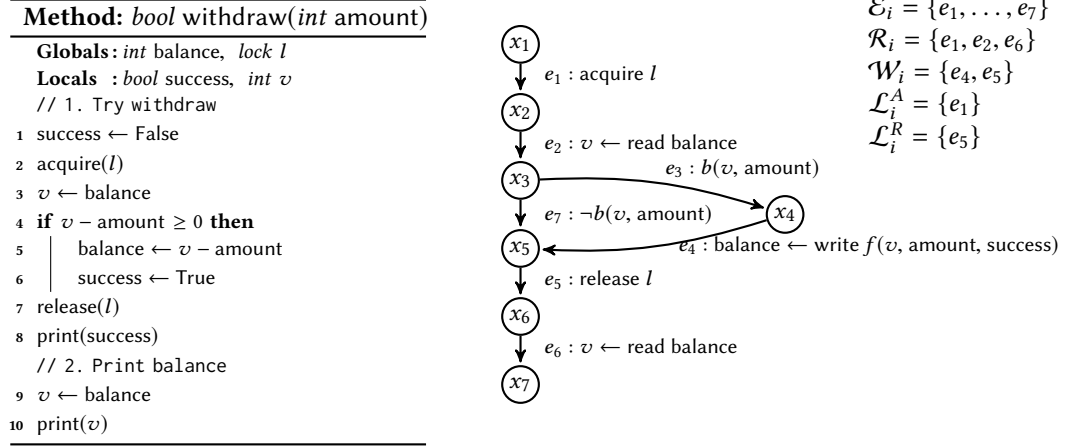


Fig. 3. (Left): A method withdraw executed whenever some amount is to be extracted from the balance of a bank account.

(Right): Representation of withdraw in our concurrent model. The root node is  $x_1$ . The program structure orders  $\text{PS}(e_2, e_4)$ . We have  $\text{loc}(e_1) = \text{loc}(e_5)$  and  $\text{loc}(e_2) = \text{loc}(e_4) = \text{loc}(e_6)$ .

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 iff it respects the condition of the branch in  $s$ . Given a state  $s$ , we denote by  $\text{enabled}(s) \subseteq \mathcal{E}$  the set of enabled events in  $s$ , and observe that there is at most one enabled event in each state  $s$  from each process. The execution of an enabled event  $e$  along an edge  $(x, y)$  of  $p_i$  in state  $s = (\text{val}, s_1, \dots, s_k)$  results in a state  $s' = (\text{val}', s_1, \dots, s'_i, \dots, s_k)$ , where  $s'_i = (y, \text{val}'_i)$ . That is, the program counter of  $p_i$  has progressed to  $y$ , and the valuation functions  $\text{val}'$  and  $\text{val}'_i$  have been modified according to standard semantics, as follows:

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

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

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

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

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

The initial state  $s^0 = (\text{val}, s_1^0, \dots, s_k^0)$  is such that the value  $\text{val}(g)$  of each global variable  $g$  comes from the unique initialization write event  $w$  with  $\text{loc}(w) = g$ , and for each  $s_i^0 = (x_i, \text{val}_i)$  we have that  $x_i = r_i$  (i.e., the program counter of process  $p_i$  points to the root node of  $\text{CFG}_i$ ). For



simplicity we restrict  $\mathcal{S}_{\mathcal{P}}$  to states  $s$  that are reachable from the initial state  $s^0$  by a sequence of events  $s^0 \xrightarrow{e_1, \dots, e_n} s$ . We focus our attention on state spaces  $\mathcal{S}_{\mathcal{P}}$  that are acyclic.

**Architecture topologies.** The architecture  $\mathcal{P}$  induces a labeled undirected communication graph  $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, \lambda_{\mathcal{P}})$  where  $V_{\mathcal{P}} = \{p_i\}_i$ . There is an edge  $(p_i, p_j)$  if processes  $p_i, p_j$  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  $\mathcal{P}$  *acyclic* if  $G_{\mathcal{P}}$  does not contain cycles. The class of acyclic architectures includes, among others, all architectures with two processes, star architectures, pipelines, tree-like and hierarchical architectures.

Table 1. Notation on the concurrent architecture.

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

## 2.2 Traces

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

**Notation on traces.** A (concrete, concurrent) *trace* is a sequence of events  $t = e_1, \dots, e_j$  such that for all  $1 \leq i < j$ , we have  $s^{i-1} \xrightarrow{e_i} s^i$ , where  $s^i \in \mathcal{S}_{\mathcal{P}}$  and  $s^0$  is the initial state of  $\mathcal{P}$ . In such a case, we write succinctly  $s^0 \xrightarrow{t} s^j$ . We fix the first  $|\mathcal{G}|$  events  $e_1, \dots, e_{|\mathcal{G}|}$  of each trace  $t$  to be initialization events that write the initial values to the global variables. That is, for all  $1 \leq i \leq |\mathcal{G}|$  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  $\mathcal{E}(t)$  the set of events that appear in  $t$ , with  $\mathcal{R}(t) = \mathcal{E}(t) \cap \mathcal{R}$  the read events in  $t$ , and with  $\mathcal{W}(t) = \mathcal{E}(t) \cap \mathcal{W}$  the write events in  $t$ , and let  $|t| = |\mathcal{E}(t)|$  be the *length* of  $t$ . For an event  $e \in \mathcal{E}(t)$ , we write  $\text{in}_t(e) \in \mathbb{N}^+$  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  $\text{enabled}(t)$  denote the set of enabled events in the state at the end of  $t$ , and call  $t$  *maximal* if  $\text{enabled}(t) = \emptyset$ . We write  $\mathcal{T}_{\mathcal{P}}$  for the set of all maximal traces of  $\mathcal{P}$ . We denote by  $s(t)$  the unique state of  $\mathcal{P}$  such that  $s^0 \xrightarrow{t} s(t)$ , and given an event  $e \in \mathcal{R}(t) \cup \mathcal{W}(t)$ , denote by  $\text{val}_t(e) \in \mathcal{D}$  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  $\tau_1, \tau_2, \dots, \tau_k$ , so that each  $\tau_i$  is wrt  $p_i$ , we denote by  $\tau_1 * \tau_2 * \dots * \tau_k$  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  $\text{Confl}(e_1, e_2)$  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  $\text{Confl}(e_1, e_2)$ . 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  $s_1 \xrightarrow{e_1, e_2} s_2$  iff  $s_1 \xrightarrow{e_2, e_1} s_2$ ,

and *dependent* otherwise. Following the standard approach in the literature, we will consider two conflicting events to be always dependent [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

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

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

Table 2. Notation on traces.

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

### 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  $e : g \leftarrow \text{write } f(v_1, \dots, v_j)$  and  $e : b(v_1, \dots, b_n)$  respectively to collapse deterministic invisible computations of each process, and only consider the value that  $f$  writes on a global variable (in addition to the side-effects that  $f$  has on local the variables of process  $p_i$ ). This is a standard approach in modeling concurrent systems, as interleaving invisible events does not change the set of reachable local states of the processes.

**Locks and synchronization mechanisms.** We treat lock-acquire 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  $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$  are called *Mazurkiewicz equivalent* if one can be obtained from the other by swapping adjacent, independent events. Formally, we write  $\sim_M$  for the Mazurkiewicz equivalence on  $\mathcal{T}_{\mathcal{P}}$ , and we have  $t_1 \sim_M t_2$  iff

(1)  $\mathcal{E}(t_1) = \mathcal{E}(t_2)$ , and

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

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

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

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  $\mathcal{T}_{\mathcal{P}}$  as follows. For  $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$  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  $\text{val}_{t_1}(w) = \text{val}_{t_2}(w)$ .*

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)  $\mathcal{E}(t_1) \subseteq \mathcal{E}(t_2)$ , and (ii) for all events  $e \in \mathcal{R}(t_1) \cup \mathcal{W}(t_1)$  we have that  $\text{val}_{t_1}(e) = \text{val}_{t_2}(e)$ .*

**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  $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$ , if  $t_1 \sim_M t_2$  then  $t_1 \sim_O t_2$ .*

PROOF. Consider any read event  $r \in \mathcal{R}(t_1)$  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  $w_1 \in \mathcal{E}(t_2)$  and  $w_2 \in \mathcal{E}(t_1)$ . Then  $w_1 \rightarrow_{t_1} r$  and  $w_2 \rightarrow_{t_2} r$ , and one of the following holds.

- (1)  $r \rightarrow_{t_1} w_2$ , and since  $w_2 \rightarrow_{t_2} r$  then  $t_1 \not\sim_M t_2$ , a contradiction.
- (2)  $w_2 \rightarrow_{t_1} w_1$ , and since  $t_1 \sim_M t_2$  we have that  $w_2 \rightarrow_{t_2} w_1$ , and thus  $r \rightarrow_{t_2} w_1$ . Since  $w_1 \rightarrow_{t_1} r$ , we have  $t_1 \not\sim_M t_2$ , a contradiction.

The desired result follows. □

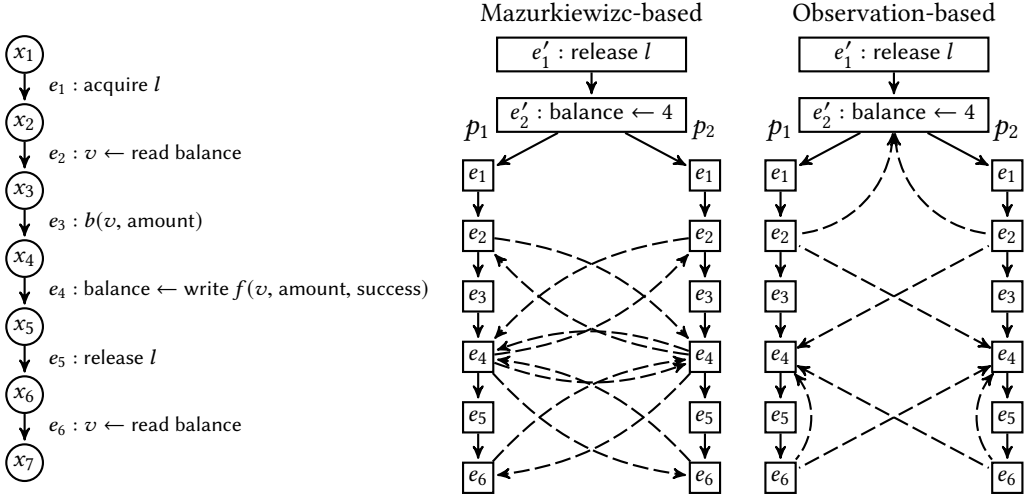


Fig. 4. Trace exploration on the system of Figure 3 with two processes, where initially  $\text{balance} \leftarrow 4$  and both withdrawals succeed.

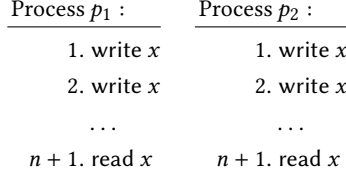
*Example 3.4 (Mazurkiewicz-based vs observation exploration).* Figure 4 illustrates the difference between the Mazurkiewicz and observation trace equivalence on the example of Figure 3. Every execution of the system starts with an initialization trace  $t^I$  that initializes the lock  $l$  to False, and the initial value  $\text{desposit} = 4$ . Consider that  $p_1$  is executed with parameter  $\text{amount} = 1$  and  $p_2$  is executed with parameter  $\text{amount} = 2$ , (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

Fig. 5. An architecture of two processes with  $n + 1$  events each.Fig. 6. An architecture of  $k$  processes with two events each.

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

*Example 3.6 (Many processes of small size).* We now turn our attention to a system  $\mathcal{P}$  of  $k$  identical processes  $p_1, \dots, p_k$  with two events each, in Figure 6. There is only one global variable  $x$ , and each process performs a read and then a write to  $x$ . There are  $O(k^k)$  realizable observation functions, by choosing for each one among  $k$  read events, one among  $k$  write events it can observe. Hence  $\mathcal{T}_{\mathcal{P}}/\sim_O$  has size  $O(k^k)$ . In contrast, the size of  $\mathcal{T}_{\mathcal{P}}/\sim_M$  is  $\Omega((k!)^2)$ . This holds as there are  $k!$  ways to order the  $k$  write events, and for each such permutation there are  $k!$  ways to assign each of the  $k$  read events to the write event that it observes. To see this second part, let  $w_1, \dots, w_k$  be any permutation of the write events, and let  $r_i$  be the read event in the same process as  $w_i$ . Then  $r_i$  can be placed right after any  $w_j$  with  $i \leq j$ . Observe that  $\mathcal{T}_{\mathcal{P}}/\sim_O$  is exponentially more succinct than  $\mathcal{T}_{\mathcal{P}}/\sim_M$ , as

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

### 3.3 Solution Overview

Traditional DPOR algorithms exploit the Mazurkiewicz equivalence, and use various techniques such as persistent sets and sleep sets to explore each Mazurkiewicz class by few representative traces. Our goal is to develop an analogous DPOR that utilizes the observation equivalence, which by Theorem 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*  $A^+ : \mathcal{R} \rightarrow \mathcal{W}$ , and
- (2) a *negative annotation*  $A^- : \mathcal{R} \rightarrow 2^{\mathcal{W}}$

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  $w : g \leftarrow \text{write } f(v_1, \dots, v_{n_i})$ , we have  $\text{val}_{A^+}(w) = f(\alpha_1, \dots, \alpha_{n_i})$ , 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  $e_r \in \mathcal{L}^R$  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  $e_a^1, e_a^2 \in \mathcal{E}(\tau_i) \cap \mathcal{L}^A$  such that  $\text{in}_{\tau_i}(e_a^1) < \text{in}_{\tau_i}(e_a^2)$  and  $\text{loc}(e_a^1) = \text{loc}(e_a^2)$  there exists a lock release event  $e_r \in \mathcal{E}(\tau_i) \cap \mathcal{L}^R$  such that  $\text{in}_{\tau_i}(e_a^1) < \text{in}_{\tau_i}(e_r) < \text{in}_{\tau_i}(e_a^2)$  and  $\text{loc}(e_r) = \text{loc}(e_a^1) = \text{loc}(e_a^2)$ .
  - (b)  $\bigcup_i \mathcal{R}(\tau_i) = \text{dom}(A^+)$  and  $\text{img}(A^+) \subseteq \bigcup_i \mathcal{W}(\tau_i)$ , 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  $X_i = X \cap \mathcal{E}_i$  the subset of events of  $X$  from process  $p_i$ . The following assertions hold:

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

#### 4.1 The Hardness of Realizing Positive Annotations

A core step in our data-centric DPOR algorithm is constructing a trace that realizes a positive annotation. That is, given a positive annotation  $A^+$ , the goal is to obtain a trace  $t$  (if one exists) such that  $O_t = A^+$ , 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



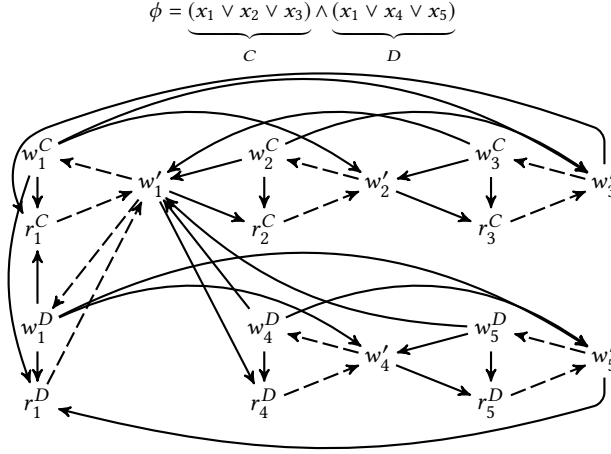


Fig. 7. The reduction of 3SAT over  $\phi$  to ACYCLIC EDGE ADDITION over  $(G, H)$ . The nodes and solid edges represent the graph  $G$ . The dashed edges represent the triplets in  $H$ .

realizing a positive annotation under sequential consistency semantics. We first show that ACYCLIC EDGE ADDITION is NP-hard, and afterwards that the problem is polynomial-time reducible to realizing a positive annotation.

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

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

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

LEMMA 4.2. *ACYCLIC EDGE ADDITION is NP-hard.*

*Sketch.* 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  $C = (x_{C_1} \vee x_{C_2} \vee x_{C_3})$ , we introduce a pair of nodes  $w_{C_j}^C, r_{C_j}^C \in V$  and an edge  $(w_{C_j}^C, r_{C_j}^C) \in E$ , where  $j \in \{1, 2, 3\}$ . Additionally, we introduce an edge  $(w_{C_j}^C, w_{C_l}^C) \in E$  for every pair  $j, l \in \{1, 2, 3\}$  such that  $j \neq l$ , and an edge  $(w_{C_j}^C, r_{C_l}^C)$  for each  $j \in \{1, 2, 3\}$ , where  $l = (j + 1) \bmod 3 + 1$ . 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  $(w_\ell^C, r_\ell^D), (w_\ell^D, r_\ell^C) \in E$ . The set  $H$  consists of triplets of nodes  $(w_{C_j}', w_{C_j}^C, r_{C_j}^C)$  for every clause  $C$  and  $j \in \{1, 2, 3\}$ . Figure 7 illustrates the construction.

**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 iff it does so in the reduced graph. The construction proceeds in two steps.

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

## 4.2 Realizing Positive Annotations in Acyclic Architectures

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

**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  $(x_{e_1, e_2} \Rightarrow x_{e_1, e_3})$ , 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  $\mathcal{P}$  be an acyclic architecture, and  $A^+$  a positive annotation over  $\mathcal{P}$ . We describe a procedure  $\text{Realize}(A^+)$  which returns a trace  $t$  that realizes  $A^+$ , or  $\perp$  if  $A^+$  is not realizable. The procedure works in two phases. In the first phase,  $\text{Realize}(A^+)$  uses Lemma 4.1 to extract a basis  $(\tau_i)_i$  of  $A^+$ . In the second phase,  $\text{Realize}(A^+)$  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  $x_{e_1, e_2}$ , where  $e_1, e_2 \in \bigcup_i \mathcal{E}(\tau_i)$ . Setting  $x_{e_1, e_2}$  to True corresponds to making  $e_1$  happen before  $e_2$  in the witness trace  $t$ . The clauses of the 2SAT instance capture four properties that each such ordering needs to meet, namely that

- (1) the resulting assignment produces a total order (totality, antisymmetry and transitivity) between all of the events that appear in adjacent processes in the communication graph  $G_{\mathcal{P}}$ ,
- (2) the produced total order respects the positive annotation, i.e., every write event  $w'$  that conflicts with an annotated read/write pair  $(r, w) \in A^+$  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. The following theorem summarizes the results of this section.

**THEOREM 4.4.** *Consider any architecture  $\mathcal{P} = (p)_i$  and let  $A^+$  be any well-formed positive annotation over a basis  $(\tau)_i$ . Deciding whether  $A^+$  is realizable is NP-complete. If  $\mathcal{P}$  is acyclic, the problem can be solved in  $O(n^3)$  time, where  $n = \sum_i |\tau_i|$ .*

## 5 DATA-CENTRIC DYNAMIC PARTIAL ORDER REDUCTION

In this section we develop our data-centric DPOR algorithm called DC-DPOR and prove its correctness and compactness, namely that 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  $r \in \text{Past}_t(e)\mathcal{R}$ , 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  $e' \rightarrow_t e$ , i.e., every event in the causal past cone of  $e$  also happens before  $e$  in  $t$ . However, the inverse is not true in general, as e.g. for some read  $r$  we have  $O_t(r) \rightarrow_t r$  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  $e$ .

**Algorithm 1:** Realize( $A^+$ )

---

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

---

```

1 Construct a directed graph  $G = (V, E)$  where
2   -  $V = \bigcup_i \mathcal{E}(\tau_i)$ , and
3   -  $E = \{(e_1, e_2) : (e_2, e_1) \in A^+ \text{ or } \text{PS}(e_1, e_2)\}$ 
4  $G^* = (V, E^*) \leftarrow$  the transitive closure of  $G$ 
   // A set  $C$  of 2SAT clauses over variables  $V_C$ 
5  $C \leftarrow \emptyset$ 
6  $V_C \leftarrow \{x_{e_1, e_2} : e_1, e_2 \in V \text{ and } e_1 \neq e_2 \text{ and either } \text{proc}(e_1) = \text{proc}(e_2) \text{ or } (\text{proc}(e_1), \text{proc}(e_2)) \in E_P\}$ 
   // 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
17 end
   // 3. Annotation clauses
18 foreach  $(r, w) \in A^+$  and  $w' \in V \cap \mathcal{W}$  s.t.  $\text{Confl}(r, w')$  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  $f : V_C \rightarrow \{\text{False}, \text{True}\}^{|V_C|}$  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  $G'$ 

```

---

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  $t_2$ .

## 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  $\text{ConflRW}(r, w)$ . 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.

---

**Algorithm 2:** DC-DPOR( $t, A$ )
 

---

**Input:** A maximal trace  $t$ , an annotation pair  $A = (A^+, A^-)$

```

// Iterate over reads not yet mutated
1 foreach  $r \in \mathcal{E}(t) \setminus \text{dom}(A^+)$  in increasing index in  $t(r)$  do
    // Find conflicting writes allowed by  $A^-$ 
2   foreach  $w \in \mathcal{E}(t)$  s.t.  $\text{Confl}(r, w)$  and  $w \notin A^-(r)$  do
3      $A_{r,w}^+ \leftarrow A^+ \cup \{(r, w)\}$ 
    // Attempt mutation and update  $A^-$ 
4     Let  $t' \leftarrow \text{Realize}(A_{r,w}^+)$ 
5     if  $t' \neq \perp$  then
6        $t'' \leftarrow$  a maximal extension of  $t'$ 
7        $A^-(r) \leftarrow A^-(r) \cup \{w\}$ 
8        $A_{r,w} \leftarrow (A_{r,w}^+, A^-)$ 
9       Call DC-DPOR( $t'', A_{r,w}$ )
10  end
11 end

```

---

We say that DC-DPOR *explores* a class of  $\mathcal{T}_{\mathcal{P}}/\sim_O$  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 4.4). 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  $A_1^+ \neq A_2^+$ .*

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

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  $O(|\mathcal{T}_{\mathcal{P}}/\sim_O| \cdot n^5)$  time.*

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 4.4, procedure *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.

Intuitively, the new locks have the effect that when calling the procedure *Realize* for realizing a positive annotation  $A^+$ , all write events protected by these locks are totally ordered by  $A^+$  (via the lock-acquire and lock-release events). Hence, *Realize* needs only to resolve orderings between read/write events to variables that (by the choice of  $X$ ) create no cycles in the communication graph.

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

- (1) Let  $Y = \bigcup_{(p_i, p_j) \in X} \lambda_{\mathcal{P}}(p_i, p_j)$  be the set of variables that appear in edges of the set  $X$ . We introduce a set of new locks  $\mathcal{L}^O$  in  $\mathcal{P}^X$  such that we have exactly one new lock  $l_g \in \mathcal{L}^O$  for each variable  $g \in Y$ .
- (2) For every process  $p_i$ , every write event  $w \in \mathcal{W}_i$  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  $\mathcal{P}$  and  $X$  an edge set of the underlying communication graph  $G_{\mathcal{P}}$ . We define a new equivalence on the trace space  $\mathcal{T}_{\mathcal{P}}$  as follows. Two traces  $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$  are *observationally equivalent refined by  $X$* , denoted by  $\sim_O^X$ , if the following hold:

- (1)  $t_1 \sim_O t_2$ , and

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

Clearly,  $\sim_O^X$  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  $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$ , if  $t_1 \sim_M t_2$  then  $t_1 \sim_O^X t_2$ .*

**Exponential succinctness of  $\sim_O^X$  in cyclic architectures.** Here we present a very simple cyclic architecture where the observation equivalence  $\sim_O^X$  refined by an all-but-two cycle set  $X$  is exponentially more succinct than the Mazurkiewicz equivalence  $\sim_M$ . Consider the architecture  $\mathcal{P}$  in

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

Fig. 8. A cyclic architecture of three processes.

Figure 8, which consists of three processes and two single global variables  $x$  and  $y$ . We choose an edge set as  $X = \{(p_1, p_2)\}$ , and  $X$  is an all-but-two cycle set of  $G_{\mathcal{P}}$ . We argue that  $\sim_O^X$  is exponentially more succinct than  $\sim_M$  by showing exponentially many traces which are pairwise equivalence under  $\sim_O^X$  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  $|T| = \binom{2 \cdot n}{n}$  as there are  $(2 \cdot n)!$  ways to order the  $2 \cdot n$  write  $y$  events of the two processes, but  $n! \cdot n!$  orderings are invalid as they violate the program structure. All traces in  $T$  have the same observation function, yet they are inequivalent under  $\sim_M$  since every pair of them orders two write  $y$  events differently. Finally,  $\mathcal{T}_{\mathcal{P}} / \sim_O^X$  is only exponentially large, and since

$$|(\mathcal{T}_{\mathcal{P}} / \sim_M) \setminus (\mathcal{T}_{\mathcal{P}} / \sim_O^X)| \geq |T| - 1$$

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



- (1) Consider the graph  $G = (V, E)$  constructed in Line 1 of Realize (Algorithm 1). For every pair of write events  $w, w'$  protected by some of the new locks  $\ell \in \mathcal{L}^O$ , 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, Realize returns  $\perp$ .
- (3) In Line 6 we use the edge set  $E_{\mathcal{P}X} \setminus X$ . 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  $(\text{proc}(e_1), \text{proc}(e_2)) \in E_{\mathcal{P}X} \setminus X$ .

We arrive at the following theorem.

**THEOREM 6.2.** *Consider a concurrent architecture  $\mathcal{P}$  of processes on an acyclic state space, and  $n = \max_{t \in \mathcal{T}_{\mathcal{P}}} |t|$  the maximum length of a trace of  $\mathcal{P}$ . Let  $X$  be an all-but-two cycle set of the communication graph  $G_{\mathcal{P}}$ . The algorithm DC-DPOR-Cyclic explores each class of  $\mathcal{T}_{\mathcal{P}} / \sim_0^X$  exactly once, and requires  $O(|\mathcal{T}_{\mathcal{P}} / \sim_0^X| \cdot n^5)$  time.*

Theorem 6.2 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<sup>1</sup>. 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.

**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 `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  $p_2, \dots, p_k$

<sup>1</sup><https://github.com/nidhugg/nidhugg>

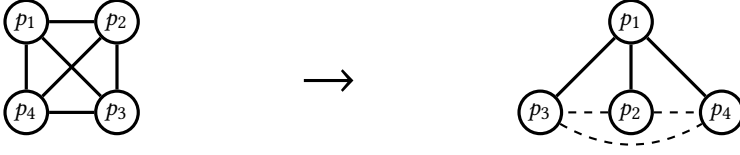


Fig. 9. Converting a cyclic architecture to a star architecture which is acyclic. On the star, solid edges correspond to observation equivalence interleavings, and dashed edges correspond to Mazurkiewicz equivalence interleavings.

are the leaves. Recall that 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.

---

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


---

```

Globals: int array[ $n + 1$ ]
// ----- Process  $j = 0$  -----
Locals : int  $i$ 
1  $i \leftarrow n$ 
2 while array[ $i$ ]  $\neq 0$  do
3   |  $i \leftarrow i - 1$ 
4 end

// ----- Process  $1 < j \leq n$  -----
5 array[ $j$ ]  $\leftarrow$  array[ $j - 1$ ] + 1

```

---



---

***opt\_lock*( $n$ ): 2 processes and  $n$  attempts**


---

```

Globals: int last_id,  $x$ 
// ----- Process  $0 < j < 2$  -----
Locals : int  $i$ 
1  $i \leftarrow 0$ 
2 while  $i < n$  do
3   |  $i \leftarrow i + 1$ 
4   | last_id  $\leftarrow j$ 
5   |  $x \leftarrow$  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

Table 3. Experimental results on two synthetic benchmarks.

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

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
<code>lastzero(4)</code>	38	2,118	0.21	0.84
<code>lastzero(5)</code>	113	53,172	0.34	19.29
<code>lastzero(6)</code>	316	1,765,876	0.63	856
<code>lastzero(7)</code>	937	-	1.8	-
<code>lastzero(8)</code>	3,151	-	9.32	-
<code>lastzero(9)</code>	12,190	-	47.97	-
<code>lastzero(10)</code>	52,841	-	383.12	-

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

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

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

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

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
<code>fib_bench(4)</code>	1,233	19,605	0.93	3.03
<code>fib_bench(5)</code>	8,897	218,243	7.41	37.82
<code>fib_bench(6)</code>	70,765	2,364,418	85.71	463.52

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

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
<code>pthread_demo(8)</code>	256	12,870	0.37	3.17
<code>pthread_demo(10)</code>	1,024	184,756	1.23	49.51
<code>pthread_demo(12)</code>	4,096	2,704,156	5.30	884.99

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

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
<code>sigma_false(6)</code>	16	10,395	0.22	2.57
<code>sigma_false(7)</code>	22	135,135	0.26	38.41
<code>sigma_false(8)</code>	29	2,027,025	0.28	658.27
<code>sigma_false(9)</code>	37	-	0.38	-
<code>sigma_false(10)</code>	46	-	0.44	-

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

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
<code>parker(8)</code>	1,254	3,343	1.52	1.33
<code>parker(10)</code>	2,411	6,212	5.03	3.96
<code>parker(12)</code>	4,132	10,361	8.09	5.62
<code>parker(14)</code>	6,529	16,022	11.96	6.86
<code>parker(16)</code>	9,714	23,427	19.89	10.85

worst case (Theorem 4.4). 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 reproducing bugs by testing, due to scheduling non-determinism, makes model checking a very relevant approach [Algave 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 process is  $p_1 = w_y^1 r_x^1$ , and the second process is  $p_2 = w_x^2 r_y^2$ . Additionally, we let  $w_x^0$  and  $w_y^0$  denote the two initialization write events.

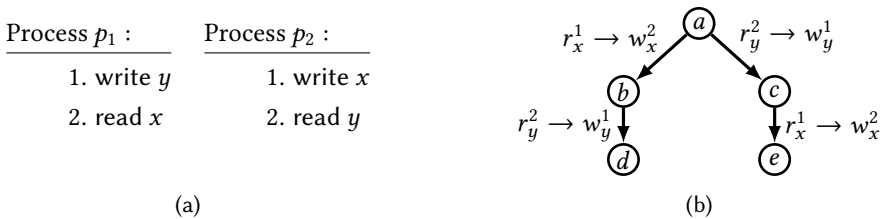


Fig. 10. A simple concurrent system of two processes, and the corresponding MCR exploration.

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  $r_x^1$  to observe the value of  $w_x^2$ , and  $r_y^2$  to observe the value of  $w_y^1$ , respectively.
- (b) When in  $b$ , the process will create the seed interleaving  $\{d\}$ , forcing  $r_y^2$  to observe the value of  $w_y^1$ .
- (c) When in  $c$ , the process will create the seed interleaving  $\{e\}$ , forcing  $r_x^1$  to observe the value of  $w_x^2$ .

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 contract, 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 Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction (POPL).
- Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *TACAS*.
- Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV*.
- Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. 2004. Zing: A Model Checker for Concurrent Software. In *CAV*.
- Jean-Marie Cadiou and Jean-Jacques Lévy. 1973. Mechanizable proofs about parallel processes. In *SWAT*.

- Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. *Data-centric Dynamic Partial Order Reduction*. Technical Report. IST Austria. <https://repository.ist.ac.at/id/eprint/872>
- E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. 1999b. State space reduction using partial order techniques. *STTT* 2, 3 (1999), 279–287.
- E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (1986).
- Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999a. *Model Checking*. MIT Press, Cambridge, MA, USA.
- Brian Densky and Patrick Lam. 2015. SATCheck: SAT-directed Stateless Model Checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 20–36. <https://doi.org/10.1145/2814270.2814297>
- Azadeh Farzan and Zachary Kincaid. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. In *CAV*.
- Azadeh Farzan and P. Madhusudan. 2009. The Complexity of Predicting Atomicity Violations. In *TACAS*.
- Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *POPL*.
- Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.
- P. Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Secaucus, NJ, USA.
- Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *POPL*.
- Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *FMSD* 26, 2 (2005), 77–101.
- Patrice Godefroid, Gerard J. Holzmann, and Didier Pirotin. 1995. State-space Caching Revisited. *FMSD* 7, 3 (1995), 227–241.
- Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *PLDI*.
- Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. 2012. Using Unfoldings in Automated Testing of Multithreaded Programs. In *ACSD*.
- Vineet Kahlon, Chao Wang, and Aarti Gupta. 2009. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV*.
- Shmuel Katz and Doron Peled. 1992. Defining Conditional Independence Using Collapses. *Theor. Comput. Sci.* 101, 2 (1992), 337–359.
- Akash Lal and Thomas Reps. 2009. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. *FMSD* 35, 1 (2009), 73–97.
- Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.
- L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (1979), 690–691.
- Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. 2010. Evaluating Ordering Heuristics for Dynamic Partial-order Reduction Techniques. In *FASE*.
- Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.
- Tom Ball Madan Musuvathi, Shaz Qadeer. 2007. *CHESS: A systematic testing tool for concurrent software*. Technical Report.
- Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.
- A Mazurkiewicz. 1987. Trace Theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer-Verlag New York, Inc., 279–324.
- K. L. McMillan. 1995. A Technique of State Space Search Based on Unfolding. *FMSD* 6, 1 (1995), 45–65.
- Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. *SIGPLAN Not.* 42, 6 (2007), 446–455.
- Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*.
- Doron Peled. 1993. All from One, One for All: On Model Checking Using Representatives. In *CAV*.
- Carl Adam Petri. 1962. *Kommunikation mit Automaten*. Ph.D. Dissertation. Universität Hamburg.
- César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based Partial Order Reduction. In *CONCUR*.
- Olli Saarikivi, Kari Kahkonen, and Keijo Heljanko. 2012. Improving Dynamic Partial Order Reductions for Concolic Testing. In *ACSD*.
- Koushik Sen and Gul Agha. 2006. Automated Systematic Testing of Open Distributed Programs. In *FASE*.
- Koushik Sen and Gul Agha. 2007. A Race-detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs. In *HVC*.



- Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A Novel Dynamic Partial-order Reduction Technique for Testing Actor Programs. In *FMOODS/FORTE*.
- Antti Valmari. 1991. Stubborn Sets for Reduced State Space Generation. In *Petri Nets*.
- Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole Partial Order Reduction. In *TACAS*.