MASTER RESEARCH INTERNSHIP

BIBLIOGRAPHIC REPORT

# Bounded Unfolding-based Dynamic Partial Order Reduction

**Domain: Data Structures and Algorithms - Logic in Computer Science**

*Author:*
Grégory GOBIN

*Supervisor:*
Thierry JÉRON
Martin QUINSON
SUMO, MYRIADS

**Abstract:** Increasingly difficult and heavy computational tasks are now frequently handled using message passing applications. However, distributed programming and more generally, dealing with concurrency is quite a challenge. It follows an increasing demand for efficient tools to ensure correctness of such applications. Model checking is one of these tools. Its goal is to perform an exhaustive exploration of every paths in the program to search for bugs. This method suffers from the state space explosion problem: the size of the whole system makes it impractical to verify. Several methods exist to alleviate this problem, some are suitable for distributed systems. In this document, we present a state of the art on such a method (Dynamic Partial Order Reduction) and its possible crossover with bounded search, a method (based on numerical bounds) to drive the verification towards particular executions and prune the others. In particular, we focus on the adaptation of a work by Coons et al. [3] on a bounded extension of DPOR to its state-of-the-art version called Unfolding-based DPOR (UDPOR) [13].

## Contents

# Introduction

Message passing applications become increasingly important in the scientific and IT landscapes as they allow to use of more complex architectures, pulling higher performances from these than a single computer can achieve. Modern high-performance computing applications (e.g. physics simulation, logical solvers) rely on the exploitation of the computational power of these architectures to handle the increasingly heavy computational tasks. However, the design of such applications is difficult and concurrency itself gives rise to a lot of correctness challenges. In particular, exhaustive bug testing on all execution paths is practically infeasible for these applications and many bugs are not visible during a simple code review. Formal techniques are thus mandatory to ensure correctness.

Formal methods are mathematically rigorous techniques for modeling and analyzing computer hardware and software. These methods, ranging from abstract interpretation to automated theorem proving through model checking, are more and more used to detect bugs in hardware and software systems as early as possible. Among these methods, one is of particular interest: model checking. It consists in the systematic verification of a specification (a well chosen logical formula) on a model of the program (in general, modelled as a transition system). Model checking has been introduced by Clarke and Emerson in [2] and Queille and Sifakis in [12]. Model checking can be decomposed in three steps, each of which with its inherant difficulties.

Modeling is the technique through which a software system (in this case, only software) is precisely and compactly described. Typically, a high-level and abstract modeling language is employed (for example, Promela, SysML, or TLA). The output of this process is a model of the software system that the target model checker can understand (*i.e.*, the model checker that will be used to check the model). The model can be created manually or automatically from the source code of the software. Model compactness is a desirable feature. Additionally, it is necessary to preserve critical qualities that are relevant to the verification process. For instance, if one wishes to test deadlocks in the system, deadlocks must be kept in the model. Modeling a huge, complicated system is difficult, requiring experience and a thorough grasp of the system. As a result, users of model checking face a challenge in ensuring that the model appropriately represents the actions of the system. There are only a few efficient formal methods (abstractions-refinement like the B Method) for determining whether a model accurately represents the behavior of the system.

The property specification step requires languages that are understandable by the model checker to specify the properties. In general, a property is a requirement specified in the system's specification. Temporal logics (such as Linear Temporal Logic, Computation Tree Logic) are frequently used to define properties. The specification of the attributes should be sound, which means that it should precisely explain the properties that are to be verified.

Finally, the verification begins. The model checker determines whether or not the provided property is satisfied. Essentially, three outcomes are possible:

- The property is satisfied by the model,
- The property is unsatisfied, and the model checker produces a counterexample in the form of a (partial) execution trace violating the property,
- The verification stage failed (e.g., it runs out of time or memory) because the number of states is too large to investigate or keep in computer memory. The model could be reduced in some way in this scenario.

It is worth noting that, if the property is violated, this does not necessarily indicate that the system contains a corresponding mistake. The verification stage may be restarted with an enhanced (more accurate and / or compact) model and an exact specification property.

The whole process described above suffers from a phenomenon called "state-space explosion". The number of states in the model becomes quickly unmanageable, even for small and simple systems, especially for distributed systems. This phenomenon comes from the fact that the number of states grows exponentially in the number of processes. In order to handle larger and more realistic systems without verification failure (*i.e.*, without timeout nor out-of-memory problems), several methods to alleviate the state-space explosion have been developed:

- Abstraction: simplifying the model to keep as few features as possible, ideally only the critical ones. The risk here is to oversimplify the model, to the point where the feature of interest are not captured.
- Symbolic model checking: this method relies on memory-efficient structures like Binary Decision Diagrams, representing Boolean formulas (or functions), to encode states and transitions.
- Partial Order Reduction (POR): based on the observation that some (independent) transitions can be taken in different orders while still leading to equivalent states. We say that these transitions commute. This technique aims at reducing the state space by exploring only one such order of the transitions. POR is considered one of the most efficient techniques to handle verification on concurrent and distributed programs.
- Bounded Model Checking: a bounded model checker only tests whether or not a particular property is satisfied within that number of steps. Essentially, instead of an exhaustive verification of all execution paths, a bounded model checker provides a guarantee on partial executions.

In this work, we are interested in Partial Order Reduction and more specifically, Dynamic POR, its stateless version as well as its interactions with bounded search, a state-space pruning method. We discuss the adaptation of a bounded extension of DPOR (Coons et al. [3]) to a state-of-the-art DPOR algorithm, based on a concurrency specific semantics of programs called Prime Event Structure (PES). This algorithm, namely Unfolding-based DPOR, introduced by Rodriguez et al. in [13] uses a labelled PES to dynamically explore the reduced state-space and thus, avoid multiple exploration of equivalent executions.

We begin, in Section 1, by a description of the representations (semantics) of programs employed by the different methods. In Section 2, we present the chosen methods: an introduction to POR techniques, a focus on UDPOR algorithm and a concise presentation of bounded search approaches. Section 3 serves as a discussion on the interactions between these methods, by presenting a first approach to merge the two. Finally, we conclude and present possible extensions to the current work.

## 1 Formal methods for message passing applications

In a distributed asynchronous system (collection of computers hosting parts of a global computational task), agents (computers) are autonomous: there is no shared memory and no common clock to synchronize them. Hence, each agent in the system acts independently from the others. However, the realisation of a global task requires communication between agents, either to share data or to synchronize actions. Thus, the whole system should be treated as a single component. We now present several semantics of distributed programs, *i.e.*, formal ways to represent the behavior of such programs that are suitable to work on.

## 1.1 Interleaving semantics of distributed programs

As applications are composed of several processes that perform local actions and communicate through messages (asynchronously) to share data and ensure synchronisation, one can begin by modelling a single agent by a labelled transition system.

**Definition 1.1.1** - *Labelled Transition System***.** A labelled transition system (LTS) is a tuple $\mathcal{T} := (S, I, \Sigma, \rightarrow)$ where $S$ is the set of states, $I \subseteq S$ the set of initial states, $\Sigma$ the set of actions and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.
We usually write $s \overset{a}{\rightarrow} s'$ instead of $(s, a, s') \in \rightarrow$. States are labelled by atomic propositions through the labelling function $L$.

We, usually, consider that actions are deterministic, *i.e.*, when an action $a$ is performed in a state $s$, the resulting state $s'$ is unique (if it exists). Transition systems for which this property holds in all states and for any action are qualified as *action-deterministic*.

**Definition 1.1.2** - *Action-deterministic Transition System***.** A system $\mathcal{T} = (S, s_I, \Sigma, \rightarrow)$ is said *action-deterministic* iff for any $s \in S, a \in \Sigma$, there is *at most* one state $s'$ such that $s \overset{a}{\rightarrow} s'$.

Equivalently, an action $a$ can be seen as a function $a : S \rightarrow S$ such that $a(s) = s'$ iff $(s, a, s') \in \rightarrow$. We denote $a(s)$ the state reached by $a$ from $s$ when it exists (what we often suppose).

A transition $a$ is *enabled* at a state $s$ if there exists a state $s'$ such that $s \overset{a}{\rightarrow} s'$. We denote *enabled*$(s)$ the set of transitions enabled at a state $s$. Given two states $s, s' \in S$ and a sequence $w = t_1 ... t_n \in \Sigma^*$, we say that $w$ reaches $s'$ from $s$ (denoted $s \overset{w}{\Rightarrow} s'$) if there exist states $s = s_0, s_1, ..., s_n = s' \in S$ such that for all $i \in [\![ 1, n ]\!]$, $s_{i-1} \overset{t_i}{\rightarrow} s_i$. We denote *state*$(w)$ the state $s$ reached by $w$ from the initial state $s_I$. When the transition system is action-deterministic, this state is uniquely determined by the sequence $w$.

A state $s$ in $\mathcal{T}$ is reachable if $s = \mathit{state}(w)$ for some sequence $w = t_1 ... t_n \in \Sigma^*$. We denote *Reach*$(\mathcal{T})$ the reachable part of the transition system. A run (or execution, interleaving) of $\mathcal{T}$ is any sequence $w$ such that $s_I \overset{w}{\Rightarrow} s$ for some $s \in S$. We denote *runs*$(\mathcal{T})$ the interleaving space of system $\mathcal{T}$.

Now, the parallel nature of these applications requires a suitable modelling scheme to represent agent interactions. Usually, the whole system is built from the agents by considering every possible series of actions (of all agents). Such a sequence of actions is called an interleaving.

**Definition 1.1.3** - *Interleaving of Transition Systems***.** Let $\mathcal{T}_i = (S_i, I_i, \Sigma_i, \rightarrow_i)$ for $i = 1, 2$, be two transitions systems. The transition system $\mathcal{T}_1 \mid\mid\mid \mathcal{T}_2$ is defined by:

$$\mathcal{T}_1 \mid\mid\mid \mathcal{T}_2 = (S_1 \times S_2, I_1 \times I_2, \Sigma_1 \cup \Sigma_2, \rightarrow)$$

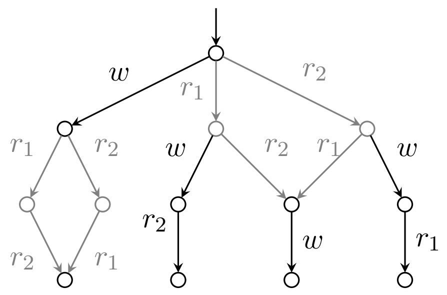
The transition relation is defined by the rules:

$$\frac{s_1 \overset{\alpha}{\rightarrow}_1 s'_1}{\langle s_1, s_2 \rangle \overset{\alpha}{\rightarrow} \langle s'_1, s_2 \rangle} \text{ and } \frac{s_2 \overset{\alpha}{\rightarrow}_2 s'_2}{\langle s_1, s_2 \rangle \overset{\alpha}{\rightarrow} \langle s_1, s'_2 \rangle}$$

The labelling function $L$ is defined by: $L(\langle s_1, s_2 \rangle) = L(s_1) \cup L(s_2)$.

| Process $P_1$ | Process $P_2$ | Process $P_3$ |
|---|---|---|
| $w: \mathit{Write}(x)$ | $r_1: \mathit{Read}(x)$ | $r_2: \mathit{Read}(x)$ |

(a) Program $\mathcal{P}$: $r_1$ and $r_2$ are independent but both dependent with $w$.



(b) Full state-space: all interleavings of $P_1$, $P_2$, $P_3$. Independent transitions are pictured in gray.

Figure 1: Interleaving semantics of a simple program $\mathcal{P}$ composed of three processes with a single instruction each.

**Example 1.1.1** - *Interleaving semantics***.** Figure 1 shows a program $\mathcal{P}$ (Fig 1a) and its interleaving semantics (Fig 1b). As each process $(P_1, P_2, P_3)$ can execute its first (and only) instruction, thus, a transition for each instruction leaves the initial state. The instruction $w : \mathit{Write}(x)$ of process $P_1$ changes the value of $x$. Hence, performing the $\mathit{Read}(x)$ instructions $r_1, r_2$ of processes $P_2$ and $P_3$ before or after action $w$ do not produce the same result. For this reason, the sequences $r_1w$ and $wr_1$ do not lead to the same state. On the contrary, before (or after) executing $w$, both actions $r_1, r_2$ read the same value of $x$ and the sequences $r_1r_2$ and $r_2r_1$ lead to the same state, producing the diamond shapes in the $LTS$ (transitions in gray on Fig 1b).

In the remaining part of the document, we will use $\mathcal{T}$ to designate an arbitrary action-deterministic transition system.

The transition systems are widely use but they are not the most suitable way to handle concurrency. The object of the next section is an alternative semantics for concurrency, the *event structures*.

## 1.2 Event structures: a semantics for concurrency

To capture the concurrency of a program, one can rely on event structures, another mathematical concept used to model computer programs. Such a structure consists in a set of events $E$ labelled by actions (or instructions of the original program) equiped with two relations: causality (also called "happens-before") and conflict. Usually, the causality relation captures dependencies between instructions: read-after-write, write-after-write but also simply the order of instructions of a single thread.

**Definition 1.2.1** - *Prime Event Structures***.** A prime event structure (PES in short) on a set of labels $\Sigma$ is a tuple $\mathcal{E} := (E, <, \#, h)$ where E is a set of events, $< \subseteq E \times E$ is a strict partial order on E, called causality relation, $h : E \to \Sigma$ is the labelling function, and $\# \subseteq E \times E$ is the symmetric, irreflexive conflict relation, compatible with the causality relation, *i.e.*, conflicts are inherited: for all $e, e', e'' \in E$, if $e \# e'$ and $e' < e''$, then $e \# e''$.
An event $e$ has only finitely many causes: $\lceil e \rceil := \{e' \in E \mid e' < e\}$.

**Definition 1.2.2** - *Configuration (of a PES)***.** A configuration $C$ of a prime event structure $\mathcal{E} := (E, <, \#, h)$ is any causally closed and conflict free set of events, *i.e.*, $C \subseteq E$ and:

- Causally closed: if $e \in C$, then $\lceil e \rceil \subseteq C$;
- Conflict Free: if $e, e' \in C$, then $\neg(e\#e')$.

Instead of keeping a collection of equivalent totally-ordered executions, a $PES$ allows us to keep partially-ordered executions, represented by configurations. In particular, the local configuration $[e]$ of an event $e$ is the $\subseteq$-minimal configuration that contains $e$ and its causes, *i.e.*, $[e] := \lceil e \rceil \cup e$. The set of configurations of a $PES$ $\mathcal{E}$ is denoted $conf(\mathcal{E})$. Two events are in immediate conflict (denoted $e\#^i e'$) if their respective histories are compatible: if and only if $e\#e'$ and both $\lceil e \rceil \cup [e']$ and $[e] \cup \lceil e' \rceil$ are configurations. Given a configuration $C$, we denote $ex(C)$ the set of extensions of $C$, *i.e.*, $ex(C) := \{e \in E \setminus C \mid \lceil e \rceil \subseteq C\}$. One can order event structures through the "prefix" relation. A $PES$ $\mathcal{E} := (E, <, \#, h)$ is a prefix of $\mathcal{E}' := (E', <', \#', h')$, denoted $\mathcal{E} \trianglelefteq \mathcal{E}'$ if and only if $E \subseteq E'$, $h$, $<$ and $\#$ are the projections of $<'$ and $\#'$ to $E$ and $\{e' \in E' \mid e' < e, e \in E\} \subseteq E$.

This concurrency semantics will serve as a guide for the exploration of the state-space of a program. Thus, we consider an incremental method to build a $PES$ from a transition system. Such a structure is called an *unfolding* of the system and its construction based on an independence relation approximating the fact that two actions commute (unformally: describe a diamond-like shape 1.1.1 and Fig 1).

**Definition 1.2.3** - *Commutation, Independence of actions***.** Two actions $a_1, a_2 \in \Sigma$ of a transition system $\mathcal{TS}$ commute in a state $s$ if (1) executing one does not enable nor disable the other and (2) if the state reached after executing both actions is the same no matter the execution order. Formally:

1. $(a_1 \in enabled(s) \wedge s \overset{a_1}{\to} s') \implies (a_2 \in enabled(s) \iff a_2 \in enabled(s'))$, and similarly for $a_2$

2. $(a_1, a_2 \in enabled(s) \implies ((s \overset{a_1 \cdot a_2}{\to} s_{1,2} \wedge s \overset{a_2 \cdot a_1}{\to} s_{2,1}) \implies s_{1,2} = s_{2,1}))$

An independence relation $I \subseteq \Sigma \times \Sigma$ is an under-approximation of the commutation relation, *i.e.*, if two actions are independent $\forall a_1, a_2, (a_1, a_2) \in I$ (also written $I(a_1, a_2)$), then they commute in all states. Conversely $a_1, a_2$ are dependent and we note $D(a_1, a_2)$ when $\neg(I(a_1, a_2))$

Using this independence relation, one can *unfold* the $LTS$ of a program into a *prime event structure* (Definition 1.2.4). In an unfolding, events are pairs $e := \langle a, C \rangle \in \Sigma \times conf(\mathcal{E})$ allowing to distinguish different occurences of the same action $a$ via the (partially-ordered) history $C$ preceding this action. Such a structure allows us to represent concurrency in a program in an exponentially more compact manner than using interleaving semantics.

**Definition 1.2.4** - *Unfolding***.** *The* unfolding of a transition system $\mathcal{T}$ under an independence relation $I$ is the $\Sigma$-PES $\mathcal{E} = (E, <, \#, \lambda)$ built incrementally by the following rules until saturation (no new event is added):

0. Start from the initial $\Sigma$-PES: $\mathcal{E} := (\emptyset, \emptyset, \emptyset, \emptyset)$

1. For any configuration $C$ of $\mathcal{E}$, any action $a \in enabled(state(C))$, add a new event $e := \langle a, C \rangle$ if for each $<$-maximal event $e'$ of $C$, $\neg I(a, \lambda(e'))$.

2. For any such new event $e = \langle a, C \rangle$, set $\lambda(e) := a$ and for every event $e' \in E \setminus \{e\}$, update $<$, $\#$ as follows:

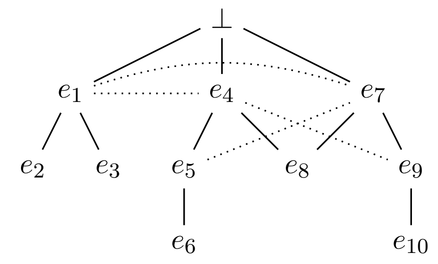
(a) if $e' \in C$, then $e' < e$,

(b) if $e' \notin C$ and $\neg I(a, \lambda(e'))$, then $e' \# e$,

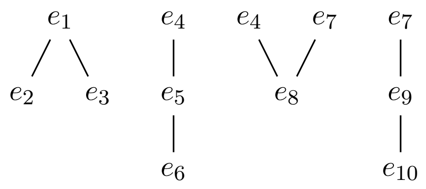
(c) if $e' \notin C$ and $I(a, \lambda(e'))$, then $e'$ and $e$ are concurrent.

**Example 1.2.1** - *Concurrency Semantics (Unfolding)***.** Figure 2 depicts the unfolding (Fig 2a) and its maximal configurations (Fig 2b). An event $\perp$ is inserted as a root of the unfolding and all other events depend causally of it. The events $e_1, e_4$ and $e_7$ are labelled $w$, $r_1$ and $r_2$, respectively. The conflicts reflect the dependences of the underlying actions: $e_4, e_7$ are concurrent ($r_1, r_2 : Read(x)$ actions are independent) whereas they are in conflict with $e_1$ (both actions are dependent with $w : Write(x)$). These conflicts turn the $PES$ into a tree-like structure of partial orders.

One can identify four maximal configurations in this unfolding (Fig 2b). These configurations are partial orders of events. Replacing events by their labels (corresponding actions), these configurations represent partially-ordered executions. We give the associated totally ordered executions (considering configurations from left to right): $\{wr_1r_2, wr_2r_1\}, \{r_1wr_2\}, \{r_1r_2w, r_2r_1w\}, \{r_2wr_1\}$.



(a) Unfolding of program $\mathcal{P}$. Solid lines express causality and dotted lines are immediate conflicts.



(b) Maximal configurations of the unfolding. Events are occurences of actions and each configuration represents partially ordered executions.

Figure 2: Concurrency semantics (unfolding) of program $\mathcal{P}$ and its maximal configurations (representing partially ordered executions).

We now have two semantics at our disposal: interleaving and concurrency semantics. These are models on which verification is performed by a model checker. However, exploration methods of the interleaving semantics face several difficulties. In the next section, we present these difficulties and introduce the adopted solutions.

## 1.3 Stateful and stateless model checking

As presented, model checking is a rather clear procedure. It performs an exhaustive search, traversing all states of the model that are reachable from the initial state. A naïve approach consists in using a depth-first search approach to recursively follow each transition from each state. To avoid repetitive exploration of the same state, one can store each visited state. Such an approach is called stateful model checking. However, such a method faces two non-trivial problems: the need in memory and the capacity to check whether two states of the program are equal.

Comparing two states is a difficult task, since a state relates to the values of variables, program counters, and many other factors. In distributed programs, the difficulty is even higher since such a program is composed of many processes and communication channels. Solutions have been imagined to associate a unique identifier to a state. However, P. Godefroid proved in [5] that computing such an identifier is hard.

These challenges have lead some researchers to develop a new approach: stateless model checking. In stateless model checking, the checker only stores the states of the currently explored path. It drastically reduces the memory usage but also introduces the inability to check whether or not a given state has been previously discovered. This inability leads to redundant explorations of the same state. Partial Order Reduction (POR) techniques which are presented and discussed in the next part (Section 2) are precisely meant to mitigate such redundancy. If the state space does not contain cycles, some of these techniques are even optimal, thus exploring as few executions as possible. In case of a cyclic state space, these techniques rely on state equality checking methods to prune the execution through specific cut-off methods.

We now present two orthogonal methods to handle the state space explosion problem: POR techniques and bounded search.

## 2 Alleviating state space explosion

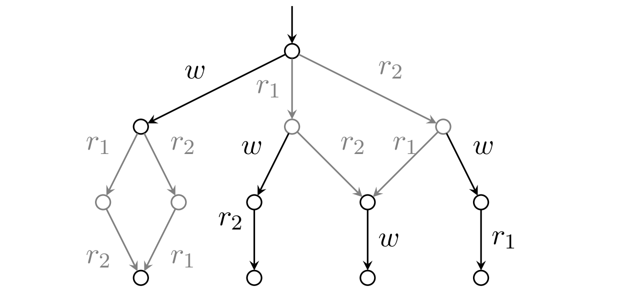
As presented in previous parts, model checking faces a phenomenon: the state space explosion problem. The number of states of a model increases exponentially in various components such as the number of variables and, for distributed programs, the number of processes. Several methods are considered to (partially) alleviate this phenomenon. In this section, we focus on partial order reduction techniques and bounded model checking.

### 2.1 Partial order reduction techniques

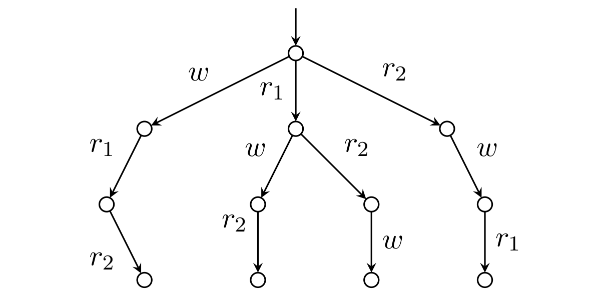
In the context of concurrent (and distributed) applications, partial order reduction (POR) is often considered to be among the most efficient approaches for alleviating the state space explosion. These techniques are widely used to detect deadlocks. If a deadlock exists, a Mazurkiewicz trace (Definition 2.1.1) leads to it and will be discovered. More generally, safety-property violations in any acyclic state space can be detected by POR algorithms. These methods attempt to minimize the size of the explored part of a system's state space by taking advantage of the commutativity of independent activities. It is possible that the outcome of concurrent actions is not dependent on the sequence in which they are performed. As a result, identifying only one arbitrary order, rather than investigating all possible orders, may be a viable strategy for verifying safety properties. POR takes advantage of this insight by choosing at each visited state, a subset of the enabled actions to either visit or avoid, based on independence of actions.

**Example 2.1.1** - *Partial Order Reduction***.** On Figure 3, we present the full state-space of program $\mathcal{P}$ (Fig 3a) and a state-space resulting from a search exploiting POR techniques (Fig 3b). Given an independence relation (Definition 1.2.3) such that $I(r_1, r_2)$, a search with partial order reduction will explore a single interleaving of $r_1$ and $r_2$ (e.g. $r_1r_2$). This choice leads to the reduced state-space of Fig 3b, where half the diamond-shape transitions patterns (in gray on the figure) are pruned. This way, the search avoids revisiting states.

As mentioned, POR relies on the exploration of executions that can be grouped in equivalence classes built on top of independence relations. Given an independence relation (either determined by static analysis, or dynamically during verification) as defined in Definition 1.2.3, one can define an equivalence relation of complete executions and the associated classes called Mazurkievicz traces [8].



(a) Full state-space: all interleavings of $P_1$, $P_2$, $P_3$.



(b) Reduced state space: a single interleaving of $r_1$ and $r_2$ is explored.

Figure 3: Partial Order Reduction applied to program $\mathcal{P}$

**Definition 2.1.1** - *Trace equivalence, Mazurkievicz trace***.** Given a set of actions $\Sigma$ and an independence relation $I \in \Sigma \times \Sigma$, the *trace equivalence* $\equiv_I$ is defined as the least congruence in the monoid $(\Sigma^*, \cdot, \epsilon)$ such that:

$$(a_1, a_2) \in I \implies a_1 a_2 \equiv_I a_2 a_1.$$

Equivalently, $w, w' \in \Sigma^*$ are equivalent $w \equiv_I w'$ if and only if there exists a finite sequence $w = w_0, w_1, ..., w_n = w'$ $(n \geq 0)$ such that for each $i \in [\![ 1, n ]\!]$, one can find two strings $u_i, v_i \in \Sigma^*$ and decompose $w_{i-1}$ and $w_i$ as follow:

$$w_{i-1} = u_i a_1 a_2 v_i \quad \text{and} \quad w_i = u_i a_2 a_1 v_i$$

Equivalence classes of $\equiv_I$ are called *Mazurkiewicz traces.* A trace contains equivalent complete executions (or interleavings) of an LTS $\mathcal{T}$. All these interleavings reach a unique state.

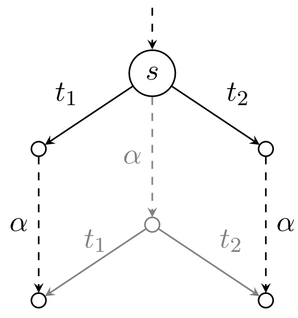
From now on, a POR algorithm is qualified as optimal if it explores one and only one execution per Mazurkievicz trace while still exploring each Mazurkievicz trace. It provides a full coverage without redundant explorations.

There are a variety of approaches for computing subsets of actions to visit. Valmari developed stubborn sets [14], while Peled produced ample sets [11]. The final method, persistent sets, was developed by Godefroid [6], and it is the one that has received the most attention in the literature.

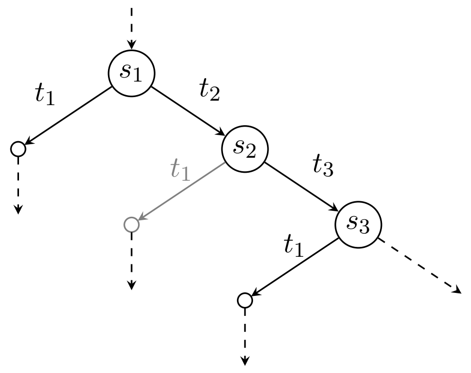
A persistent set is a provably-sufficient subset of the set of enabled transitions such that unselected transitions are guaranteed not to interfere with the execution of those being selected. Intuitively, a subset $P$ of the set of transitions enabled in a state $s$ is called persistent in $s$ if whatever one does from $s$, while remaining outside of $P$, does not interact with $P$.

**Definition 2.1.2** - *Persistent set (at a state s)***.** A set $P$ of transitions is *persistent in s* if and only if, for all nonempty sequences $\alpha = a_0 a_1 ... a_n$ such that $s = s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} s_1 ... \overset{a_n}{\to} s_{n+1}$ from $s$, such that $a_i \notin P$ $(0 \leq i \leq n)$ then, $a_n$ is independent in $s_n$ with all transitions in $P$.

**Example 2.1.2** - *Persistent set***.** Figure 4a shows how a *persistent set* allows to reduce the state-space. At state $s$, a persistent set $P$ is computed, containing transitions $t_1$ and $t_2$. As $P$ is persistent in $s$, $t_1$ and $t_2$ do not interfere with other transitions outside of $P$ (they are independent of any other action). More formally, if a state is reachable from $s$, it is also reachable by a sequence starting by an action of $P$. It is thus sufficient to explore transitions of $P$ and, consequently, prune the remaining part of the state-space (in gray on Figure 4a).



(a) Persistent set at state $s$: $t_1, t_2 \in P$, $\alpha = a_0a_1...a_n$ is a sequence of unselected actions, *i.e.*, for $0 \leq i \leq n$, $a_i \notin P$



(b) Sleep set: $I(t_1, t_2)$ but $\neg I(t_1, t_3)$, thus $t_1$ is in the sleep set at $s_2$ but not at $s_3$.

Figure 4: *Persistent set* and *Sleep set*: parts in gray may be pruned during the selective search.

A naïve method to compute a persistent set consists in accumulating all dependent transitions in the persistent set until saturation. It ensures that any action outside the persistent set is independent of all actions of the persistent set. The *persistent-set technique* alone can still lead to the exploration of multiple interleavings of independent transitions. When the algorithm cannot avoid the selection of such independent transitions, *the sleep-set technique* is a complementary method to circumvent the wasteful exploration of the interleavings of these transitions. This technique exploits information about the past of the search to block the future exploration of transitions.

**Example 2.1.3** - *Sleep set*. Figure 4b depicts a sleep-set pruned transition. Transitions $t_1$ and $t_2$ are independent and it is not necessary to explore $t_1$ after $t_2$ (at state $s_2$) as the part of the state-space reachable through $t_1$ has been explored at state $s_1$. Thus, after exploring $t_1$ at state $s_1$, $t_1$ is added to the sleep set of $s_2$ and all transitions dependent with $t_1$ are discarded from the sleep set of $s_2$. At state $s_2$, $t_1$ is pruned ("sleeping" transitions are not explored). However, $t_1$ and $t_3$ are dependent and new states are reachable after executing $t_3$ and $t_1$. Therefore, $t_1$ is not passed in the sleep-set at state $s_3$ and the exploration continues.

A *selective search* algorithm, using *persistent-set* and *sleep-set* techniques in conjunction, is presented on Figure 5. The properties of this selective search are studied by Godefroid [4].

This algorithm performs a depth-first traversal of the system by computing, at each visited state, a persistent set of actions to explore and by propagating the information through sleep sets. Once a transition $t_1$ has been explored from a state $s$ (reaching state $s_1$), transitions independent with $t_1$ are passed to the sleep set at $s_1$. Once another transition $t_2$ has been explored (reaching $s_2$), the sleep set at $s_2$ is the sleep set at $s$ with a few modifications: $t_1$ is added to it and every transition dependent with $t_2$ are removed from it. The algorithm continues the exploration, thus building the sleep set at a state $s'$ reached from $s$ through an action $a$ by enriching the sleep set with all actions already executed from $s$ and purging all actions dependent with $a$ in $s$.

The principle of all POR approaches is precisely to reduce the state space exploration while covering at least one execution per Mazurkiewicz trace. If a deadlock exists, a Mazurkiewicz trace leads to it and will be discovered. More generally, safety-property violations in any acyclic state space can be detected by POR algorithms.

```
Data: $Stack := \{s_0\}$, $Hash := \emptyset$, $s_0.Sleep := \emptyset$
while $Stack \neq \emptyset$ do
    Pop $s$ from $Stack$
    if $s \notin Hash$ then
        Add $s$ to $Hash$
        $T = PersisentSet(s) \setminus s.Sleep$
    else
        $T = \{a \in \Sigma \mid a \in Hash(s).Sleep \wedge a \notin s.Sleep\}$
        $s.Sleep = s.Sleep \cap Hash(s).Sleep.$
        $Hash(s).Sleep = s.Sleep$
    end
    forall $a \in T$ do
        $s' = a(s)$ // action $a$ is executed
        $s'.Sleep = \{b \in s.Sleep \mid (a, b)$ are independent in $s\}$
        Push $s'$ onto $Stack$
        $s.Sleep = s.Sleep \cup \{a\}$
    end
end
```

Figure 5: Selective search using persistent sets and sleep sets

A recent upgrade in POR methods, UDPOR, uses an unfolding of the program into a prime event structure to guide the state space exploration and make the reduction implicit. We will now explain its behavior.

## 2.2 Unfolding-based Dynamic Partial Order Reduction (UDPOR)

In this section, we present a state-of-the-art POR method. This method (UDPOR) exploits an alternative semantics for concurrency (defined in Section 1.2) and achieves optimality by exploring a single execution by Mazurkievicz traces.

The key concepts behind UDPOR are the notions of partial orders and the structure in which they are agglomerated: an unfolding. In their article, Rodriguez et al. [13], use the fact that one can associate a partial order to each Mazurkievicz trace. This partial order of action occurences (events) is treated as a part of a richer structure: a *prime event structure* (Definition 1.2.1).

Figure 6 presents the UDPOR exploration algorithm (Rodriguez et al. [13]). Here, Mazurkievicz traces are explicitly represented by configurations and UDPOR explores a single linearization of each configuration.

The algorithm starts with an initially empty set $U$. All events encountered during the exploration will be appended to it. Then the procedure *Explore* is called with empty sets as parameters: $C$, the first parameter is a configuration encoding the currently explored execution, $D$ for *disabled*, the second parameter, is a set of events to *avoid* (acting as a sleep set) and $A$, the last one, contains events conflicting with $D$ and is used as a guide to find alternative executions. The procedure begins by computing all extensions of the configuration $C$ and appending them to $U$ (line 4). The algorithm backtracks (at line 6) if, either each enabled action should be avoided ($en(C) \subseteq D$) or

```
1  Set $U := \emptyset$
2  Call Explore($\emptyset, \emptyset, \emptyset$)
3  Procedure Explore($C, D, A$)
4      Compute $ex(C)$, and add all events in $ex(C)$ to $U$
5      if $en(C) \subseteq D$ then
6          Return // Backtrack: sleep-set blocked or deadlocking execution
7      end
8      if $(A = \emptyset)$ then
9          chose $e$ from $en(C) \setminus D$
10     else
11         choose e from $A \cap en(C)$
12     end
13     Explore($C \cup e, D, A \setminus e$) // Left Call: Explore an extension of configuration
14     if $\exists J \in Alt(C, D \cup e)$ then
15         Explore($C, D \cup e, J \setminus C$) // Right Call: Explore an alternative configuration
16     end
17     $U := U \cap Q_{C,D}$
```

Figure 6: Unfolding-based POR exploration

there are no such action ($en(C) = \emptyset$). The former case corresponds to a redundant call and thus, to a sleep-set blocked execution while the latter case corresponds to a deadlock or the end of the program. If UDPOR continues (does not backtrack), an enabled event $e$ is chosen (line 8-12), in guiding set $A$ if possible ($A \neq \emptyset$) and a recursive exploration $Explore(C \cup \{e\}, D, A \setminus \{e\})$ of the extended configuration $C \cup \{e\}$ is called (the "left" call, line 13). It continues avoiding the events in $D$, following the guide $A$ from which $e$ is removed. Once the "left" call completed, all configurations containing $C \cup \{e\}$ have been explored. Thus, the algorithm checks if there exists a configuration containing $C$ but not $e$ by computing *alternatives* $Alt(C, D \cup \{e\})$ (line 14). If such a configuration exists, a recursive call is performed (the "right" call, line 15), this time $e$ is added to $D$ (the search avoids $e$) and the alternative $J \setminus C$ is used as a guide. The alternatives are the UDPOR counterparts to the backtracking sets of the original DPOR.

Formally, an *alternative* to $D' = D \cup \{e\}$ after $C$ in $U$ is a set of events $J \subseteq U$ that does not contain events in $D'$ ($J \cap D' = \emptyset$), constitutes a configuration $C \cup J$ after $C$ and such that any event in $D'$ conflicts with some event in $J$. The computation of alternatives is a costly operation: in order to avoid redundant explorations and thus, sleep-set blocked executions, function $Alt(C, D \cup \{e\})$ has to solve an NP-complete problem [3]. As shown by Coti et al. in [10], Optimal DPOR [1] faces the same problem. In [10], the authors present another approach called *k-partial alternatives* in which the *Alt* function computes sets of events $J$ conflicting with only $k$ events in $D$ instead of all of them. It allows to tune between an optimal and a quasi-optimal search that might be more efficient. It is important to note that, experiments [1] show promising results: even for small values of $k$, optimality is achieved.

These POR techniques provide a full coverage of the transition system to verify safety properties. However, complex systems are still hard to handle through these methods. In the next section, we introduce an orthogonal technique that allows the verification of a selected part of the system:

bounded search.

## 2.3 Bounded search

Despite their efficiency, POR techniques still struggle with large complex systems: it can still be necessary to reduce the number of executions to take into account and thus, the number of visited states. An interesting way to filter out several executions is to use bounds to push the search towards executions of particular interest and discard the undesired ones. The intuition behind bounded search is that, when searching for concurrency bugs or safety issues, only a few interactions (the nature of these interactions may vary) between the processes are necessary to cause these issues. This intuition relies on the observation (and supposition) that a bug requiring less actions to set up is more likely to happen, and conversely. Bounded search does not visit all relevant reachable states but only those that are reachable within the bound, thus pruning the state-space. A search provides the *bounded coverage* if it explores all relevant states reachable within the bound.

For a bounded search, the bound may be any property of an execution. One can, for example, choose to monitor the occupation of the processor by each process (by counting the difference of processor release / yield operations). This way, one can block processes monopolizing the processor for too long in order to force processes to take turns. This example is called a fair bounded search: it drives the verification towards (more typical) executions in which processes share the resources. Fair stateless model checking can thus be seen as bounded search. Previous works also include bounded search using the depth, the number of context switches [9], the number of preemptive context switches [9] (preemption bounding).

The bounds are formally defined in [3] as a pair $(Bv, c)$ where $Bv : \Sigma^* \to \mathbb{N}$ is a bound evaluation function and $c \in \mathbb{N}$ is the threshold (the bound itself). These data are inputs to the bounded search algorithm. In particular, two bound evaluation functions are presented in [3]: preemption bound and fair bound.

In [3], Coons et al. present BPOR, an algorithm that takes advantage of POR in preemption-bounded and fair-bounded search while maintaining the bounded coverage.

# 3 Interaction between DPOR and Bounds

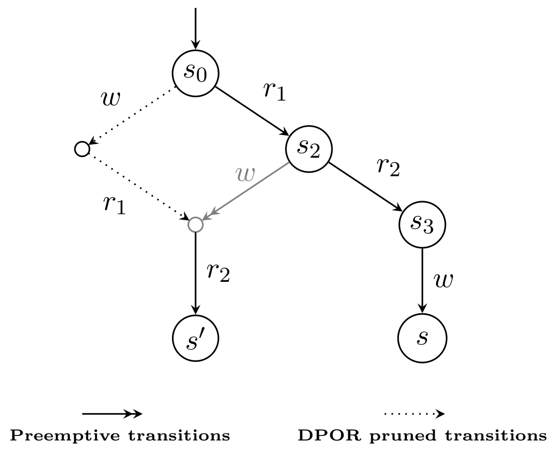
In this section, we present the approach of Coons et al. [3] to merge DPOR exploration with a bounded search. In particular, the team points out that a naïve combination of DPOR and bounded search exposes the difficulties of combining the two methods: bounds create dependences.

## 3.1 Artificial Dependences

POR techniques are sensible to the independence relation used to approximate commutation of transitions (see Definition 1.2.3): a finer relation (closer to commutation relation, with more independences) entails a more efficient reduction. The introduction of bounds can disturb partial-order reduction: take a bound $(Bv, c)$ and two independent transitions $t_1, t_2$ such that $Bv(t_1t_2) < c$ but $c \leq Bv(t_2t_1)$. Then, despite $I(t_1, t_2)$, the order of execution matters. In particular, if DPOR explores $t_2$ first, it will not be able to explore $t_1$, thus leaving states unvisited while reachable within the bound. The bound creates an artificial dependency between $t_1$ and $t_2$. Because it leaves reachable states unvisited, this artificial dependence leads to unsoundness of the search.

| Process $P_1$ | Process $P_2$ |
|---|---|
| $w:\ Write(y)$ | $r_1:\ Read(x)$ |
| | $r_2:\ Read(y)$ |

(a) Program $Pr$: $w$ and $r_1$ are independent (operations on different variables), $w$ and $r_2$ are dependent (same variable involved).



(b) State-space of the program $Pr$ illustrating a preemption-bounded search with DPOR. The transition pruned by the bounded search appears in gray.

Figure 7: Bounded search with DPOR. The preemption bound makes $w$ and $r_1$ dependent, thus leaving $s'$ unvisited whereas it is reachable within the bound.

**Example 3.1.1** - *Artificial Dependence***.** Figure 7 shows how naïve preemption-bounded search with POR leads to unvisited state $s'$ and thus, unsoundness of the search. The program $Pr$ (Fig 7a) is a simple two processes (or threads) program, executed on the same computing unit and thus sharing resources. Process $P_1$ changes the value of shared variable $y$ (action $w : Write(y)$) while process $P_2$ reads the variables $x$ and $y$ (actions $r_1$ and $r_2$). Involving different variables, the instructions $w$ (of $P_1$) and $r_1$ (of $P_2$) are independent but $w$ and $r_2$ are dependent. Figure 7b shows the result of a naïve preemption-bounded search with POR by restricting the number of preemptions to 0. At state $s_0$, DPOR prunes transition $w$ (and the following $r_1$) and chooses to explore $r_1$, *i.e.*, among the two possible interleavings of independent transitions $w$, $r_1$, it will explore $r_1w$. However, at state $s_2$, the transition $w$ is a preemption: it interrupts process $P_2$ during its execution to execute an instruction of $P_1$. This transition is, thus pruned by the bound and the exploration continues until it reaches state $s$. Here, state $s'$ is never visited (although it is reachable within the bound) because the artificial dependence between $w$ and $r_1$ is not taken into account. However, the execution $wr_1r_2$ is valid within the bound: after $w$, $P_1$ ends and $r_1$ is not preemptive. Hence, if, instead of executing $r_1$ at state $s_0$, DPOR choosed to explore $w$, state $s'$ would have been visited.

This example shows the importance of identifying artificial dependences to maintain both bounded coverage and POR efficiency. In their article [3], Coons et al. study interesting properties of the bounds to keep POR efficiency and maintain bounded coverage. These properties are explained in the next section.

## 3.2 Properties for compatibility

As shown in previous section, combining bounds and partial order reduction is not trivial: it introduces artificial dependences. In their article [3], Coons et al. identified two properties that should be satisfied by the bounds to allow a good interface with POR. In this section, we present these two properties (stability and extensibility) as well as the bounds (not) satisfying these properties.

The first property is called *stability*. With stable bounds, two sequences leading to the same global state have the same bound value (the same cost). Stable bounds are desirable because independent transitions remain commutative and POR exploits its commutativity to reduce the state space.

**Definition 3.2.1** - *Stable Bound Function.* A bound function $Bv$ is *stable* if all executions of a Mazurkievicz trace have the same cost, *i.e.*, for $\omega, \omega' \in runs(\mathcal{T})$,

$$\omega \in [\omega'] \implies Bv(\omega) = Bv(\omega')$$

Any two equivalent sequences (thus leading to the same global state) have the same bound value.

When the bound under consideration is not stable, a part of the state-space may appear unreachable within the bound whereas it is reachable. In fact, the first path explored by the search may not be the cheapest one, thus leading to redundant executions (of the same trace) to ensure visiting each state reachable within the bound. A selective search will prune the state space most efficiently if it explores the cheapest sequence of transitions first.

**Definition 3.2.2** - *Extensible Bound Function.* A bound function $Bv$ is *extensible* if independent transitions do not affect each other cost, *i.e.*, for any $t \in \Sigma$, $w \in \Sigma^*$ and $w' = a_0 a_1 ... a_n$ such that $I(t, a_i)$ for all $i \in [\![ 0, n ]\!]$,

$$Bv(wtw') = max(Bv(wt), Bv(ww'))$$

The second property, *extensibility*, ensures that independent transitions do not affect each other cost. An extensible bound preserves local state reachability: states that were previously reachable are still reachable after exploring independent transitions. It is not the case when the bound is not extensible: independent transitions affect each other cost, thus the order of execution of these transitions matters. Bounds that are not extensible sacrifice partial-order reduction by forcing the exploration of independent transitions, leading to many redundant states.

Most of the meaningful bounds are either not stable, not extensible or neither. For example, the depth of exploration used as a bound is stable but not extensible, preemption bounding is neither stable nor extensible and the same goes for fair bounding (see Section 3.3). There is at least a (trivial) stable and extensible bound: the constant bound returning zero. Indeed, this is equivalent to an unbounded search and allows for full exploitation of partial order reduction. Outside of this trivial bound, the usual ones impose artificial dependences that must be treated to reduce their impact on the effiency of partial-order reduction and ensure bounded coverage.

In [3], the authors present a modification of persistent sets to accomodate artificial dependences from preemption and fair bounding. Their approach is detailed in the next section.

## 3.3 A first approach: preemption and fair bounding

In order to compensate for artificial dependences for preemption and fair bounding, Coons et al. produced an adaptation of DPOR to bounded search called BPOR [3]. Informally, BPOR relies on the computation of suitable sets of transitions to visit at each state, allowing to reduce the visited state-space while maintaining bounded coverage. The key concept behind BPOR is the concept of *bound sufficient set.*

**Definition 3.3.1** - *Sufficient Set.* A set $T \subseteq \Sigma$ of actions is *sufficient* at a state $s$ if and only if any state $s'$ reachable from $s$ (i.e., $s \overset{w}{\Rightarrow} s'$ for some $w \in \Sigma^*$) is also via at least on action in $T$: $s'$ is reachable by a sequence $w_T = aw'$ with $a \in T$.

The authors then provide algorithm to compute bound sufficient sets for preemption-bounded and fair-bounded search and prove that these sets are sufficient to explore the bounded state space soundly. The core idea is to build the sufficient sets as sets of backtracking points. The search inserts backtracking points conservatively to reverse the order of exploration of dependent transitions. This method allows to compensate for the artificial dependences imposed by the bounding constraints, thus providing bounded coverage. However, the conservative aspect of the method (it adds possibly useless backtracking points) limits partial order reduction.

## 4 Conclusion

As message passing applications become more and more prominent in the scientific landscape, the need for guarantees of security become increasingly important. Among available tools to provide these guarantees, one is particularly suitable for finding bugs: model checking. Indeed, model checking is a prolific formal method consisting in the systematic verification of a property on a model of the program. However, this method has a limited range: as soon as systems become too complex, model checkers are no more able to handle the size of the state space. In order to mitigate this problem, several solutions, ranging from partial order reduction to bounded search, have been developed. In this document, we gave a detailed presentation of available partial order reduction techniques which allow for state-space reduction by exploiting concurrency in programs. We (briefly) presented bounded search, a method using bounds to drive the exploration to executions susceptible to contain bugs. This technique thus speeds up bug discovery. Lastly, we presented the difficulties to combine the approaches and the solution adopted by Coons et al. [3] to accommodate partial order reduction and bounded search. It is known that combining bounded search and POR techniques is not trivial: artificial dependences may limit POR efficiency and even lead to unsound exploration. Coons et al. made it for DPOR, producing the BPOR algorithm but POR techniques have been improved with a state-of-the-art method: UDPOR. It remains, to our knowledge, to study the same combination of bounded search and UDPOR. Several authors conjecture (or produce) a possible extension of POR techniques (UDPOR included) to *conditional* (or local) independence relation [4, 13, 7]. More specifically, an example of such an independence relation is presented in [4] in which the author bounds the size of queues during the search. Without being properly presented as a bounded search, the idea of encoding bounds in the independence relation looks promising.

## References

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *SIGPLAN Not.*, 49(1):373–384, jan 2014.

[2] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[3] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. *SIGPLAN Not.*, 48(10):833–848, October 2013.

[4] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[5] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97*, pages 174–186, Paris, France, 1997. ACM Press.

[6] Patrice Godefroid, Gerhard Goos, Juris Hartmanis, and Jan Leeuwen, editors. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[7] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.

[8] Antoni Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[9] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 446–455. ACM, 2007.

[10] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 354–371, Cham, 2018. Springer International Publishing.

[11] Doron Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification*, pages 409–423, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[12] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

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

[14] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pages 491–515, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.