Context-Sensitive Dynamic Partial Order Reduction

Elvira Albert[1], Puri Arenas[1], María García de la Banda[2,4], Miguel Gómez-Zamalloa[1], and Peter J. Stuckey[3,4]

[1] DSIC, Complutense University of Madrid, Spain
[2] DCIS, University of Melbourne, Australia
[3] Faculty of IT, Monash University, Australia
[4] IMDEA Software Institute

Abstract. Dynamic Partial Order Reduction (DPOR) is a powerful technique used in verification and testing to reduce the number of *equivalent* executions explored. Two executions are equivalent if they can be obtained from each other by swapping adjacent, non-conflicting (*independent*) execution steps. Existing DPOR algorithms rely on a notion of independence that is *context-insensitive*, i.e., the execution steps must be independent in all contexts. In practice, independence is often proved by just checking no execution step writes on a shared variable. We present context-sensitive DPOR, an extension of DPOR that uses *context-sensitive independence*, where two steps might be independent only in the particular context explored. We show theoretically and experimentally how context-sensitive DPOR can achieve exponential gains.

1 Introduction

A fundamental challenge in the verification and testing of concurrent programs arises from the combinatorial explosion that occurs when exploring the different ways in which processes/threads can interleave. Partial-order reduction (POR) [4,6,7] is a general theory that provides full coverage of all possible executions of concurrent programs by identifying equivalence classes of redundant executions, and only exploring one representative of each class. Two executions are said to be equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (i.e., independent) execution steps. POR-based approaches avoid the exploration of such equivalent executions thanks to the use of two complementary sets: *persistent sets* and *sleep sets*. Intuitively, the former contains the execution steps that must be explored (as they might lead to non-equivalent executions), while the latter contain the steps that should no longer be explored (as they lead to executions equivalent to others already explored).

In the state-of-the-art POR algorithm [5], called DPOR (*Dynamic POR*), persistent sets are computed dynamically by only adding a step to the persistent set (called *backtracking set* in DPOR terminology) if the step is proved to be dependent on another previously explored step. Refining dependencies thus improves POR verification methods [8, 15]. While very effective, DPOR is not optimal, as it sometimes explores equivalent executions. Optimality was later achieved by *optimal*-DPOR [1] thanks to the analysis of past explorations to build *source sets* and *wakeup trees*. Intuitively, the former is a relaxation of persistent sets that avoids exploring steps that will later be blocked by the sleep set. The latter stores fragments of executions that are known not to end up being blocked by the sleep set. Source sets and wakeup trees, respectively, replace persistent sets and enhance the performance of sleep sets. Together, they ensure the exploration of all equivalence classes with the minimum number of executions, regardless of scheduling decisions.

Our work stems from the observation that source sets, and their predecessors persistent sets, are computed dynamically based on a notion of *context-insensitive independence*, which requires two steps be independent in *all possible* contexts. While optimal-DPOR has indeed been proved to be optimal, it is so only under the assumption of context-insensitive independence. In existing implementations of both DPOR and optimal-DPOR [1, 5], context-insensitive independence is over-approximated by requiring global variables accessed by one execution step not to be modified by the other. The contribution of this paper is to extend the DPOR framework to take advantage of *context-sensitive independence*, that is, of two steps being independent for a given state encountered during the execution, rather than for all possible states. For example, steps $\{\mathit{if}\ (\mathit{cond})\ f = \mathit{0}\}$ and $\{f{+}= 3\}$ are independent for states where *cond* fails, but not for states where it holds.

Context-sensitiveness is a general, well-known concept that has been intensively studied and applied in both static analyses [13] and dynamic analyses [11]. The challenge is in incorporating this known concept into a sophisticated framework like DPOR. We do so by adding to the computation of the standard sleep sets any sequence of steps that are independent in the considered context, so that the exploration of such sequence is later avoided. Our extension is orthogonal to the previous improvements of source sets and wakeup trees, and can thus be used in conjunction with them.

Importantly, our method also provides an effective technique to improve the traditional over-approximation of context-insensitive independence. Consider, for example, the simple case where two steps add a certain amount to the value of a variable, or the more complex case of an agent-based implementation of merge sort, where each agent splits their input into two parts, gives them to child agents to sort, and then merges the result. Both cases will give the same result regardless of the execution order, and at least the merge case will be difficult to prove. Our context-sensitive approach can easily determine in both cases that the orders lead to the same result (for each particular input being tested) and, hence, only consider one execution order of the processes. Without this, the algorithm will need to consider an exponential number of executions for the merge case, even though they are all equivalent. Our experimental results confirm our method achieves exponential speedups.

2 Preliminaries

Following [1], we assume our concurrent system is composed of a finite set of *processes* (or threads) and has a unique *initial state* s_0. Each process is a sequence of atomic execution *steps* that are globally relevant, that is, might depend on and affect the global state of the system. Each such step represents the combined effect of a global statement and a finite sequence of local statements, ending just before the next global statement in the process. The set of processes enabled by state s (that is, that can perform an execution step from s) is denoted by $\mathit{enabled}(s)$.

An *execution sequence* E is a finite sequence of execution steps performed from the initial state s_0. For example, $q.r.r$ is an execution sequence that executes the first step of process q, followed by two steps of process r. The state reached by execution sequence E is unique and denoted by $s_{[E]}$. Executions sequences E and E' are *equivalent* if they reach the same state: $s_{[E]} = s_{[E']}$. An execution sequence is *complete* if it exhausts all processes (that is, there is no possible further step).

An *event* (p, i) denotes the i-th occurrence of process p in an execution sequence, and $\mathit{proc}(e)$ denotes the process of event e. The set of events in execution sequence E is denoted by $\mathit{dom}(E)$, and contains event (p, i) iff p appears at least i times in E. We use $e <_E e'$ to denote that event e occurs before event e' in E, and $E \leq E'$ to denote that sequence E is a prefix of sequence E'. Note that $<_E$ establishes a total order between events in E. Let $\mathit{dom}_{[E]}(w)$ denote the set of events in execution sequence $E.w$ that are in sequence w, that is, $\mathit{dom}(E.w) \backslash \mathit{dom}(E)$. If w is a single process p, we use $\mathit{next}_{[E]}(p)$ to denote $\mathit{dom}_{[E]}(p)$.

The core concept in POR is that of the *happens-before* partial order among the events in execution sequence E, denoted by $\rightarrow_E$. This relation defines a subset of the $<_E$ total order, such that any two sequences with the same happens-before order are equivalent. POR algorithms use this relation to reduce the number of equivalent execution sequences explored, with Optimal-DPOR ensuring that only one execution sequence in each equivalence class is explored. The happens-before partial order has traditionally been defined in terms of a *dependency* relation between the execution steps associated to those events [7]. Intuitively, two steps p and q are *dependent* if there is at least one execution sequence E for which they do not commute, either because one enables the other or because $s_{[E.p.q]} \neq s_{[E.q.p]}$. Instead, the Optimal-DPOR algorithm is based on a very general happens-before relation that is not defined in terms of a dependency relation [1]. It simply requires it to satisfy the following seven properties for all execution sequences E:

1. $\rightarrow_E$ is a partial order on $\mathit{dom}(E)$, which is included in $<_E$.
2. The execution steps of each process are totally ordered, i.e. $(p, i) \rightarrow_E (p, i{+}1)$ whenever $(p, i + 1) \in \mathit{dom}(E)$, as one enables the other.
3. If E' is a prefix of E, then $\rightarrow_E$ and $\rightarrow_{E'}$ are the same on $\mathit{dom}(E')$. That is, adding more events cannot change the order among previous events.
4. Any linearization E' of $\rightarrow_E$ on $\mathit{dom}(E)$ is an execution sequence with exactly the same happens-before relation $\rightarrow_{E'}$ as $\rightarrow_E$. Thus, $\rightarrow_E$ induces a set of

equivalent execution sequences, all with the same happens-before relation. We use $E \simeq E'$ to denote that E and E' are linearizations of the same happens-before relation, and $[E]_{\simeq}$ to denote the equivalence class of E.

5. If $E \simeq E'$, then $s_{[E]} = s_{[E']}$, thus ensuring equivalent sequences commute.
6. For any sequences E, E' and w, such that $E.w$ is an execution sequence, we have $E \simeq E'$ iff $E.w \simeq E'.w$.
7. If p, q, and r are different processes, then if $\mathit{next}_{[E]}(p) \rightarrow_{E.p.r} \mathit{next}_{[E.p]}(r)$ and $\mathit{next}_{[E]}(p) \not\rightarrow_{E.p.q} \mathit{next}_{[E.p]}(q)$, then $\mathit{next}_{[E]}(p) \rightarrow_{E.p.q.r} \mathit{next}_{[E.p.q]}(r)$. This ensures that if the next step of p happens-before the next step of r, this will still be the case if we add in the middle a step independent of p.

The above relation is used for defining the concept of a *race* between two events. Event e is said to be in race with event e' in execution E, if the events have different processes, e happens-before e' in E ($e \rightarrow_E e'$), and the two events are "concurrent", i.e. there exists an equivalent execution sequence $E' \simeq E$ where the two events are adjacent. We write $e \precsim_E e'$ to denote that e is in race with e' and that the race can be reversed (i.e., the events can be executed in reversed order).

3 The happens-before relation is not context-sensitive

One could think the generality of the above happens-before definition allows it to capture context-sensitive independence and, thus, there is no need to modify DPOR to achieve context-sensitivity. The following example shows this is not the case and explains why Optimal-DPOR might explore sequences avoided by our method. Consider three simple processes defined by:

p: `write(x=5)` q: `write(x=5)` r: `read(x)`

All three pairs of associated execution steps will usually be considered as dependent, which means traditional DPOR methods will process the 6 sequences resulting for all permutations of $\{p, q, r\}$. However, there are only 2 different resulting states, one where r is executed after q and/or p thus reading 5, and one where it is executed before the others, thus reading 0. Let us construct a minimal (i.e., least restrictive) happens-before partial order for all execution sequences of $\{p, q, r\}$. For sequences of length 2, the only properties that need to be considered are 1, 4 and 5 (all others deal with at least three events in the execution). A minimal partial order that satisfies these three properties is:

$$\begin{tikzcd}
(a) & (b) & (c) & (d) & (e) & (f) \\
p \arrow[d] & p \arrow[d] \arrow[d, dotted, bend left] & q \arrow[d] & q \arrow[d] \arrow[d, dotted, bend left] & r \arrow[d] \arrow[d, dotted, bend left] & r \arrow[d] \arrow[d, dotted, bend left] \\
q & r & p & r & p & q
\end{tikzcd}$$

where the dotted arrows indicate a happens-before order between the two process steps, and the continuous arrows indicate the $<_E$ total order within the execution sequence. That is, the following relations hold: (a) $p \not\rightarrow_{p.q} q$, (b) $p \rightarrow_{p.r} r$, (c) $q \not\rightarrow_{q.p} p$, (d) $q \rightarrow_{q.r} r$, (e) $r \rightarrow_{r.p} p$, (f) $r \rightarrow_{r.q} q$. Note that (b), (d), (e) and (f) are needed as, otherwise, properties (4) and (5) of the happens-before definition above would require (b) and (e) to be equivalent, as well as (d) and (f). As given, only (a) and (c) are equivalent.

For sequences of length 3, all properties need to be considered, although our example makes property 2 directly satisfied, as each process has only one execution step. Property (6) requires $p \rightarrow_{p.q.r} r$ to hold, due to (a) and (b). Similarly, $q \rightarrow_{q.p.r} r$ must hold due to (c) and (d). However, these cannot be the only happens-before relations for sequences $p.q.r$ and $q.p.r$, as this would contradict property (4): $q.p.r$ is a linearization of the happens-before for $p.q.r$ and, hence, must have identical happens-before relations. Hence, $q \rightarrow_{p.q.r} r$ and $p \rightarrow_{q.p.r} r$ must also hold. Consider now sequence $p.r.q$. By property (3), $p \rightarrow_{p.r.q} r$ must hold. Again, this cannot be the only relation for the sequence as, by (4), it would also be the only relation for sequence $p.q.r$. Hence, $r \rightarrow_{p.r.q} q$ must also hold. Similarly, $q \rightarrow_{q.r.p} r$ and $r \rightarrow_{q.r.p} p$ must also hold. A similar reasoning can be done for sequences $r.p.q$ and $r.q.p$, obtaining the following minimal happens-before relation for sequences of length 3:

$$\begin{tikzcd}
(g) & (h) & (i) & (j) & (k) & (l) \\
p \arrow[d] \arrow[dd, dotted, bend right] & p \arrow[d] \arrow[d, dotted, bend left] & q \arrow[d] \arrow[dd, dotted, bend right] & q \arrow[d] \arrow[d, dotted, bend left] & r \arrow[d] \arrow[d, dotted, bend left] \arrow[dd, dotted, bend right] & r \arrow[d] \arrow[d, dotted, bend left] \arrow[dd, dotted, bend right] \\
q \arrow[d] \arrow[d, dotted, bend left] & r \arrow[d] \arrow[d, dotted, bend left] & p \arrow[d] \arrow[d, dotted, bend left] & r \arrow[d] \arrow[d, dotted, bend left] & p \arrow[d] & q \arrow[d] \\
r & q & r & p & q & p
\end{tikzcd}$$

Since $g \simeq i$ and $k \simeq l$, Optimal-DPOR must explore at least 4 different sequences with this minimal happens-before relation. Furthermore, it would need to explore all 6 sequences using the traditional happens before over-approximation. In contrast, using context-sensitivity we can determine that $s_{[d]} = s_{[h]} = s_{[i]} = s_{[j]}$ and $s_{[k]} = s_{[l]}$. Hence, only two sequences must be explored. As we will see later, our algorithm explores 3 sequences when using the traditional happens-before relation.

4 Context-sensitivity can give exponential gains

Let us motivate the relevance of our work by means of a typical producer-consumer interaction where we can see that the gain of using context-sensitive independence can be exponential. Consider two processes, a producer (p) that stores results in a bounded buffer (a FIFO queue), and a consumer (c) that takes them from the buffer, defined as follows:

```
produce(Q,v)
    if $Q$ not full
        $Q$ := $Q ++ [v]$
```

```
consume(Q)
    if $Q$ not empty
        let $Q = [v] + +Q'$
        $Q$ := $Q'$
        return $v$
    else return $\bot$
```

Let us, for simplicity, assume that calls to produce and consume are atomic, that is, locks are used to prevent their concurrent execution. In any sequence E containing events (p, i) and (c, j), either $(p, i) \rightarrow_E (c, j)$ or $(c, j) \rightarrow_E (p, i)$ must hold, even in a minimal happens-before relation. However, as long as the buffer is neither empty, nor full, both orders lead to the same state.

Given n occurrences of the producer and n of the consumer, a context-insensitive algorithm will need to explore all their interleavings, since each occurrence happens-before the other. This means exploring $\binom{2n}{n}$ executions. However, most of these lead to the same state: if the size k of the buffer is $k \geq n$, the state is determined by the subset of consumers that read an empty buffer and, hence, there are exactly 2^n different states.

Consider an example where $n = 3$, $k = 5$, and p stores 1,2,3 in sequence. A DAG representing all execution sequences is given in Fig. 1. For clarity, edges for the consumer appear as dotted and labeled by the value consumed. Nodes represent states and are labeled by the number of elements in the buffer. For brevity, we denote events (p, i) and (c, j) as p_i and c_j, respectively.

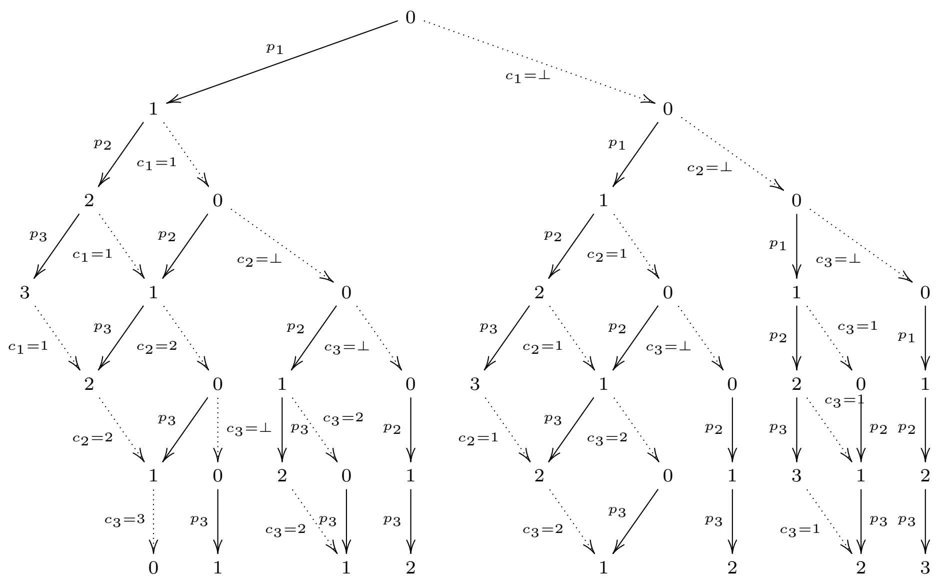

Fig. 1. All interleavings of consumers (dotted) and producers for $n = 3$ and $k = 5$.

Note that there are only $2^3 = 8$ non-equivalent execution sequences, rather than $\binom{6}{3} = 20$. The reduction given by context-sensitive independence is exponential, since each state labeled from 1 to $k-1$ has two paths leading to the same state, hence reducing the number of leaves of the resulting subtree by a factor of 2 (modulo some edge effects). Section 6.1 gives experimental results on the application of our context-sensitive method to this example.

5 Context-sensitive DPOR

We will use the Source-DPOR algorithm [1] both to explain and to implement our method. This is because Source-DPOR is usually faster than Optimal-DPOR in practice, and its algorithm (and thus our extension) is much easier to understand. Both the original algorithm and our extension are formulated in a general setting, which only assumes the existence of a happens-before relation between the events of an execution. It can thus be used both for computational models where dependency of concurrent threads is based on modifying shared variables, and for those where dependency of asynchronous message-passing processes is based on modifying shared messages. Most examples in the paper use shared variables, as traditional in the DPOR literature, while our implementation is developed for an asynchronous message-passing language (see Section 6).

5.1 The extended algorithm

Source-DPOR can be obtained from Algorithm 1 by removing lines 11-14 and line 16, which provide our extension. Note also that we have made the sleep

Algorithm 1 Context-sensitive DPOR

```
 1: procedure EXPLORE($E, Sleep$)
 2:     $sleep(E) := Sleep$;
 3:     if $(\exists p \in (enabled(s_{[E]}) \setminus Sleep))$ then
 4:         $backtrack(E) := \{p\}$;
 5:         while $(\exists p \in (backtrack(E) \setminus sleep(E)))$ do
 6:             for all $(e \in dom(E)$ such that $e \precsim_{E.p} next_{[E]}(p))$ do
 7:                 let $E' = pre(E, e)$;
 8:                 let $v = notdep(e, E).p$;
 9:                 if $(I_{[E']}(v) \cap backtrack(E') = \emptyset)$ then
10:                     add some $q'$ in $I_{[E']}(v)$ to $backtrack(E')$
11:                 let $u = dep(e, E)$
12:                 if $(\nexists w \in sleep(E')$ where $w \leq v.u)$ then
13:                     if $(s_{[E.p]} = s_{[E'.v.u]})$ then
14:                         add $v.u$ to $sleep(E')$;
15:             $Sleep' := \{v \mid v \in sleep(E), E \models p \diamond v\}$
16:                         $\cup \{v \mid p.v \in sleep(E)\}$;
17:             EXPLORE($E.p, Sleep'$);
18:             $sleep(E) := sleep(E) \cup \{p\}$;
```

set for each sequence E, $sleep(E)$, global in line 2, as our modifications require the addition of new elements to previous sleep sets. Let us first describe the behaviour of the original Source-DPOR algorithm. As shown in Algorithm 1, Source-DPOR extends an execution sequence E with current sleep set $Sleep$, which contains the set of processes that previous executions have determined do not need to be explored yet from E. Initially, the algorithm starts with an empty sequence and an empty sleep set. In general, the algorithm starts by selecting any process p that is enabled by the state reached after executing E and is not already in $Sleep$. If it does not find any such process p, it stops. Otherwise, it initiates the backtrack set of E (i.e., the set of processes that must be explored from E) to be the singleton $\{p\}$, and starts exploring every element p in this set that is not in $sleep(E)$ (which is the same as $Sleep$ in the original algorithm). Note that the backtrack set of E might grow as the loop progresses (due to later executions of line 10).

For each such p, Source-DPOR performs two phases: race detection (lines 6 to 10) and state exploration (lines 15, 17 and 18). The race detection starts by finding all events e in $dom(E)$ such that $e \precsim_{E.p} next_{[E]}(p)$. For each such e, it sets E' to $pre(E, e)$, i.e., to be the prefix of E up to, but not including e. It also sets v to $notdep(e, E).p$, where $notdep(e, E)$ is the subsequence of events of E that occur after e but do not "happen after" e (i.e., every e' such that $e <_E e'$ and $e \not\to_E e'$). It then checks whether there is any process in the backtrack set of E that appears also in $I_{[E']}(v)$, where $I_{[E']}(v)$ denotes the set of processes that perform events in $dom_{[E']}(v)$ that have no happens-before predecessors in $dom_{[E']}(v)$. If there is no such process, it adds any process in $I_{[E']}(v)$ to the backtrack set of E'. Note that this has the effect of adding new processes to the backtrack sets of earlier parts of the exploration tree (right before e was explored in E). After this, Source-DPOR continues with the state exploration phase for $E.p$, by retaining in its sleep set $Sleep'$ any element v in $sleep(E)$ that is independent of p in E (denoted as $E \models p \diamond v$), i.e., any v such that the next event $next_{[E]}(p)$ would not happen-before any event in $dom[E.p](v)$. After this, the algorithm explores $E.p$ with sleep set $Sleep'$, and finally it adds p to $Sleep$ to ensure p is not selected again.

Let us now explain the new lines added by our method. We start during the race detection phase, where event e has been detected in the original Source-DPOR to be in a reversible race with the next event $next_{[E]}(p)$. We first set u to be $dep(e, E)$, i.e., to be the sub-sequence of E that starts with e and contains all events that "happen after" e in E. We then simply need to check (line 13) whether inverting the sequences of events v and u after E' will lead to the same state and, if so, add $v.u$ to $sleep(E')$. However, none of this is needed if there is already something in $sleep(E')$ that will by itself prevent us from exploring the reversed sequence $v.u$. This is why we first check, in line 12, whether $v.u$ has a prefix w ($w \leq v.u$) in $sleep(E')$ and, if so, do nothing. The only other change occurs during the exploration phase. In the sleep set propagation step that computes $Sleep'$, any sequence $p.v$ in $sleep(E)$ that starts with the process p we are about to explore, is replaced by v in the initial sleep set of the new state. This is not needed in the original Source-DPOR, because its *Sleep* set only has processes, not sequences.

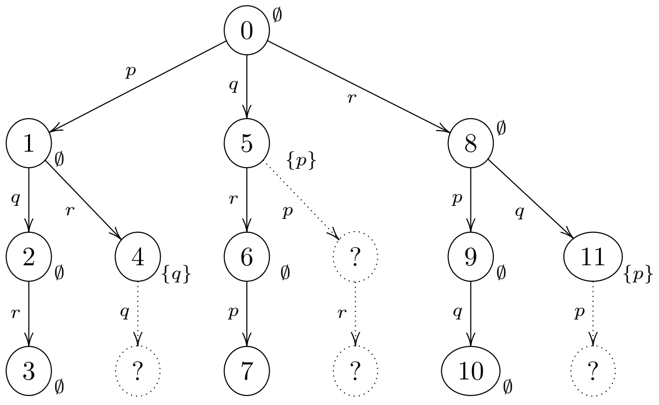

Fig. 2. Context-sensitive DPOR with initial sleep sets for each state. The dotted components would be visited by optimal-DPOR with the traditional happens-before.

Example 1. Let us follow the algorithm's execution on the example of Section 3 (Fig. 2) but using the traditional happens-before approximation where all p, q and r are dependent to each other. Since all processes have only one execution step, by an abuse of notation, we will refer to events by their process name. The algorithm starts with sequence ϵ and an empty sleep set, denoted as state 0 in Fig. 2. The execution first chooses p, detects no races and explores sequence p with an empty sleep set to state 1. The execution then chooses q, detects a reversible race with p, and adds q to $backtrack(\epsilon)$, i.e., state 0 in the figure. At this point our method confirms that $s_{[p.q]} = s_{[q.p]}$, and thus adds $q.p$ to $sleep(\epsilon)$, i.e., state 0, indicating there is no need to explore q from it. The execution proceeds by exploring sequence $p.q$ with an empty sleep set to state 2. Now only r can be chosen. The execution detects a reversible race with q, and adds r to $backtrack(p)$, i.e., state 1. Our method confirms that $s_{[p.q.r]} = s_{[p.r.q]}$, thus adding $r.q$ to $sleep(p)$, i.e., state 1. The execution then explores sequence $p.q.r$ to state 3 and finds the first solution, where r reads x as 5. It then backtracks to state 1, adding r to $sleep(p.q)$ and q to $sleep(p)$ on the way. Next, it chooses r, and finds a reversible race with p which adds r to $backtrack(\epsilon)$. Our method also realises $s_{[p.r]} \neq s_{[r.p]}$, which means nothing needs to be added to $sleep(\epsilon)$. The execution then explores $p.r$ to state 4 with the sleep set $sleep(p.r)$ initialized to q. Thanks to this, q cannot be selected at this point and the execution backtracks to state 0, adding r to $sleep(p)$ and p to $sleep(\epsilon)$ on the way. In the original source-DPOR algorithm q would not have been in $sleep(p.r)$, since $r.q$ would not have been in $sleep(p)$. Hence, it would have explored the full sequence $p.r.q$.

The execution then backtracks to state 0 and explores sequence q to state 5, with $\mathit{sleep}(q)$ initially set to $\{p\}$ (since $\mathit{sleep}(\epsilon)$ was $\{p, q.p\}$ at this point). The execution can then only choose r. It finds a reversible race but does not add anything to $\mathit{backtrack}(\epsilon)$ since r is already there. Our method also realises $s_{[q.r]} \neq s_{[r.q]}$. It then explores $q.r$ to state 6. The execution chooses p detects a reversible race with r at state 5, and adds p to $\mathit{backtrack}(q)$. Since p is already in $\mathit{sleep}(q)$, it does not check for equivalence. The method finds an equivalent solution at state 7. In the original Source-DPOR algorithm p would not have been in $\mathit{sleep}(q)$, since $q.p$ would not have been in $\mathit{sleep}(\epsilon)$. Hence, it would have explored the full sequence $q.p.r$.

The execution now backtracks to state 0 and explores sequence r to state 8 with $\mathit{sleep}(r)$ initially empty. The execution then chooses p, finds a reversible race with r, but does not update anything, as the tests in lines 9 and 12 fail. The execution explores $r.p$ to state 9 with $\mathit{sleep}(r.p)$ initially empty. It then chooses q, finds a reversible race with p, and adds q to $\mathit{backtrack}(r)$. Our method then confirms $s_{[r.p.q]} = s_{[r.q.p]}$, and adds $q.p$ to $\mathit{sleep}(r)$. The execution then explores sequence $r.p.q$ to state 10, and finds the second solution where r reads x as 0. The execution then backtracks to state 8, adding q to $\mathit{sleep}(r.p)$ and p to $\mathit{sleep}(r)$ on the way. The execution then chooses q and finds a reversible race with r which produce no effects. It then explores $r.q$ to state 11 with $\mathit{sleep}(r.q)$ initially set to $\{p\}$. Since nothing can be selected, the execution terminates. In the original Source-DPOR p would not have been in $\mathit{sleep}(r.q)$, since $q.p$ would not have been in $\mathit{sleep}(r)$. Hence, it would have explored the full sequence $r.q.p$.

The execution has explored 3 complete sequences rather than the minimal 2, whereas the original Source-DPOR would have explored 6 (rather than its minimal 4). Note that while some redundant executions are not detected until their last steps, others are detected earlier, as is the case for sequence $q.p.r$. □

The above example shows in detail how the context-sensitive DPOR algorithm works step by step, and how it can detect equivalent execution sequences. However, the example is too simple to show how the algorithm can make real reductions in the exploration. Note that the dotted derivations that the original Source-DPOR algorithm would have explored, have also been executed by our algorithm in order to do the context-sensitive checks in line 13. Hence, though the context-sensitive DPOR algorithm has obtained less solutions, it has not been able to reduce the exploration, and it has performed some recomputations. The following example illustrates how context-sensitive DPOR is able to achieve reductions while exploring execution sequences.

Example 2. Let us consider the execution of our algorithm on the producer-consumer example of Section 3, and let us assume it first explores the execution sequence $p_1.p_2.p_3$, shown as the leftmost sequence in Fig. 1. Up to this point no race has been detected. Now the algorithm can only select c_1 and detects a reversible race with p_3 adding c_1 to $\mathit{backtrack}(p_1.p_2)$. It then confirms $s_{[p_1.p_2.p_3.c_1]} = s_{[p_1.p_2.c_1.p_3]}$ and, hence, adds $c_1.p_3$ to $\mathit{sleep}(p_1.p_2)$. After exploring the complete leftmost branch, it backtracks to state $p_1.p_2$ and selects c_1. It then detects the race with p_2, adding c_1 to $\mathit{backtrack}(p_1)$. It then confirms

$s_{[p_1.p_2.c_1]} = s_{[p_1.c_1.p_2]}$ and, hence, adds $c_1.p_2$ to $sleep(p_1)$. It then executes c_1, reaching state $p_1.p_2.c_1$ with p_3 in its sleep set. At this point the algorithm can only select c_2. The important point to note is that the algorithm has been able to avoid exploring the equivalent sub-sequence $p_3.c_2.c_3$ that the original DPOR algorithm would have had to explore. The reduction is made more apparent if we continue some more steps. Let us assume the algorithm has already found the second solution and backtracks to state p_1 to select c_1. After managing the race with p_1, and executing c_1, it will reach state $p_1.c_1$ with p_2 in its sleep set. Importantly, this will prevent our algorithm from exploring the whole execution tree below $p_1.c_1.p_2$. As we will see later in Section 6.1, our algorithm is able to obtain the minimal number of 2^n solutions for this example and, more importantly, it is able to reduce exponentially the number of states explored. □

5.2 Soundness

Soundness relies on showing that any omitted Mazurkiewicz trace, i.e, the happens-before order of a complete execution sequence, is equivalent to an explored one in terms of the final state.

Lemma 1. *If the context-sensitive DPOR algorithm discovers that $s_{[E.p]} = s_{[E'.v.u]}$, for any complete sequence C of the form $C = E'.v.u.w$ there is a complete sequence $C' = E.p.w$ that defines a different Mazurkiewicz trace $T' = \rightarrow_{C'}$ and leads to the same final state.*

Proof. Let $C = E'.v.u.w$ be a complete execution sequence. Since $s_{[E.p]} = s_{[E'.v.u]}$, we have that $s_{[C]} = s_{[C']}$ where $C' = E.p.w$. Note that for C, $next_{[E]}(p) \rightarrow_C e$, while in C', $e \rightarrow_{C'} next_{[E]}(p)$. □

Theorem 1. *For each Mazurkiewicz trace T defined by the happens-before relation, $\mathsf{Explore}(\epsilon, \emptyset)$ explores a complete execution sequence that either implements T, or reaches the same state as one that implements T.*

Proof. Consider an execution of $\mathsf{Explore}(\epsilon, \emptyset)$ without the additions for context-sensitivity, and assuming we always choose an enabled process that would not be sleep set blocked in the extended algorithm, wherever possible. This is exactly the source-DPOR algorithm of [1] and, hence, is guaranteed to explore a complete execution sequence that implements each T [12].

Suppose that some Mazurkiewicz trace T is omitted by our context-sensitive DPOR, C is the complete execution sequence that implements T ($T = \rightarrow_C$) and is explored by the original source-DPOR algorithm. This sequence must be cut by our algorithm. Thus, it must be of the form $C = E'.v'.u'.y$, where our algorithm added $v.u$ to $sleep(E')$ after finding $s_{[E.p]} = s_{[E'.v.u]}$, and $v'.u'$ is $v.u$ possibly with some events added that do not depend on any event in $dom_{[E']}(v.u)$, as otherwise the sleep set entry would have been removed. Hence, there exists a complete execution sequence $E'.v.u.w.y$ with the same happens-before relation as C, obtained by moving events independent of $v.u$ (those with processes in w) after $v.u$. By Lemma 1 there is a different trace T' which leads to the same state as C. Since the source-DPOR tree explores a complete execution sequence for each Mazurkiewicz trace, it must include a complete execution sequence C' that implements T'. Note that C' has the same happens before relation as $E.p.w.y$.

We now show that C' appears to the left of C in the source-DPOR tree. Sequence $E.p$ clearly appears to the left of C in the source-DPOR tree, or it could not be used to add the sleep set entry that blocked C. Suppose to the contrary that C' appears to the right of C. Let E'' be the largest common prefix of C and C'. Now $C = E''.q.w'$ for some q. Since C' appears to the right of C, then q will be in the sleep sets for (the remainder of) sequence C' unless it is removed by some dependent event. Let $e' = next_{[E'']}(q)$.

Suppose that $E''.q \leq E'$ then the first change is above E'. The happens-before relation for C' must then have some event e'' (after $E''.q$) such that $e'' \rightarrow_{C'} e'$, but this cannot be the case since $\rightarrow_{C'} = \rightarrow_{E.p.w.y}$ where this does not occur.

Suppose that $E' \leq E''$ and, thus, the first change is at or after the place where $E.p$ and $E'.v.u$ differ. Clearly C' must appear to the right of $E'.proc(e)$ (otherwise it would be left of C). Hence, $proc(e)$ is in the sleep set (for the remainder) of C' after E' until removed by dependent events. Suppose event e'' removes $proc(e)$. Then, we have that $e'' \rightarrow_{C'} e$. This is a contradiction since this does not occur in $E.p.w.y$.

Hence, C' must appear to the left of C in the source-DPOR tree. If C' exists in the tree visited by context-sensitive DPOR we are done, since we have found an equivalent complete sequence. Otherwise, we can apply the same construction to discover an equivalent complete sequence that occurs to the left in the original source-DPOR tree. The procedure must terminate since, eventually, we reach the left most branch, which cannot be removed by the context-sensitive additions to the algorithm. $\square$

5.3 Optimizations

Let us now discuss two possible optimizations that are crucial to fully exploit the algorithm's potential, as our experiments in Section 6.3 show.

1. Anticipating cuts: Consider a very frequent situation, where E is an execution sequence with state s and enabled steps $p_1, p_2, q_1, \ldots, q_n$. Steps p_1 and p_2 are independent in the context of s, but considered as dependent, either because there is a context in which they are, or because of a loss of precision in the dependency over-approximation (e.g., they both increment the same variable). Steps $q_1, \ldots, q_n$ might have some dependencies among them but none is dependent with p_1 nor p_2. Let us assume our algorithm selects first p_1 and then p_2. At this point, p_2 is added to the backtrack set of E (line 10), and the sequence $p_2.p_1$ added to the sleep set of E (line 14). When the algorithm backtracks to E, the sleep set contains p_1 (due to line 18) and $p_2.p_1$. Let us assume it selects p_2. The sleep set is updated to include p_1, since line 15 removes p_1 but line 16 puts it there again. Thus, our algorithm reaches a sequence E' with enabled steps $p_1, q_1, \ldots, q_n$ and p_1 in the sleep set. If none of the steps $q_1, \ldots, q_n$ transitively generates a step that is dependent on p_1, then all execution sequences coming from this point will be sleep set blocked (since p_1 will always remain in the sleep set) and many useless computations will be performed. If we can compute a set O that over-approximates the set of steps that can arise in any future execution from the current state, and none of these depend on p, we can then block sequence E'. In general, whenever a sequence in the sleep set (added by line 14 due to a context-sensitive check) is consumed except for its last step l (by the successive executions of line 16), if no step in O is dependent on l, we block the execution at this point. Section 6 describes the analysis we implemented to compute such O for actors.

2. Guiding with sleep sequences: The algorithm makes three arbitrary selections: the first step to explore (line 3); the next step to backtrack with (line 5); and a step for the backtrack set (line 10). Implementations should make these selections such that the shortest sequences in the sleep set are explored first. This allows context-sensitive equivalent explorations to be discarded as soon as possible. Otherwise, potentially good sleep sequences (i.e. those that will be responsible for important exploration reductions) could be discarded.

6 Implementation and experimental evaluation

We have implemented and experimentally evaluated our method for actor programs within the tool SYCO [3], a systematic testing tool for ABS programs [10]. SYCO can be used online through its web interface available at `http://costa.ls.fi.upm.es/syco`.

6.1 Producer-consumer with actors

Actor programs consist of computing entities called actors, each with its own local state and thread of control, that communicate by exchanging messages asynchronously. The actor concurrency model [2,9] has been regaining popularity lately and is used in many systems such as Go, ActorFoundry, Asynchronous Agents, Charm++, E, ABS, Erlang, and Scala. It is also influencing commercial practice, with Twitter using actors for scalability and Microsoft using them in the development of its asynchronous agents library.

An actor configuration consists of the local state of the actors and a set of pending *tasks*. In response to receiving a message (or task), an actor can update its local state, send messages (tasks) to another actor or itself (using the ! function), or create new actors (using the instruction new). Actor languages often have instructions to await for an asynchronous call to terminate. The actor model is characterized by inherent concurrency of computation within and among actors (note that tasks within each actor work on a locally shared memory), dynamic creation of actors, and interaction only through direct asynchronous message passing with no restriction on message arrival order. In the computation of an actor system, there are two non-deterministic choices: selecting an actor and scheduling one of its pending tasks.

```
{ /* main Block */
  List<Data> d = D0;
  Buffer b = new Buffer();
  Producer p = new Producer(b);
  Consumer c = new Consumer(b);
  p ! produceN(d);
  c ! consumeN(size(d));
}
class Consumer(Buffer b){
  Data consumeN(Int n){
    while (n > 0){
      await b!take();
      n = n − 1;
    }
  }
}
class Producer(Buffer b){
  Unit produceN(List<Data> d){
    while (not empty(d)){
      await b!store(head(d));
      d = tail(d);
    }
  }
}
```

```
class Buffer {
  List⟨Int⟩ buffer = Nil;
  Int n = 0;
  Int max = MAX;
  Unit store(Data d){
    if (n < max){
      buffer = append(buffer, d);
      n = n + 1;
    }
  }
  Data take(){
    if (n > 0){
      Data d = head(buffer);
      buffer = tail(buffer);
      n = n − 1;
      return d;
    }
    else
     return Nil;
  }
}
```

Fig. 3. Actor-based producer-consumer program

Fig. 3 shows an actor-based version of the producer-consumer program provided in Section 4. The execution starts from an (initially empty) actor that executes the main block, shown at the top, to create three concurrent actors representing the buffer of size MAX, the consumer and the producer. The last two receive a reference to the buffer b used for the communication. The main block then performs two asynchronous calls to add tasks on the producer and consumer to execute the corresponding methods. These tasks will in turn make asynchronous calls on the buffer to create the tasks that consume and produce data on it. The search tree that results from the execution of the main block has the same shape as the one in Fig. 1. Basically, the actor program for one producer and one consumer creates 4 tasks: consumeN, produceN, store and take. As consumeN and produceN do not modify the shared data (i.e., the buffer), they are trivially independent from all others. In contrast, store and take are detected as conflicting due to their write accesses to the buffer. As in the thread-based version, most steps lead to the same state, i.e., they are context-sensitive independent.

Table 1 experimentally compares Source-DPOR and context-sensitive DPOR (CDPOR) on the producer-consumer problem. Column *Execs* gives the number of complete executions sequences explored, *Time* the total time taken in mil-

	Source-DPOR			**CDPOR**			**Red. gains**	
N	**Execs**	**Time**	**States**	**Execs**	**Time**	**States**	**Execs**	**Time**
3	20	5	69	8	6	52	2.5x	0.9x
5	252	100	923	32	58	324	7.9x	1.8x
7	3432	1663	12869	128	357	1712	26.9x	4.7x
9	48620	30856	184755	512	2284	8428	95.0x	13.6x

Table 1. Reduction gains on consumer-producer

liseconds, and *States* the number of explored states obtained when executing the above example with our context-sensitive DPOR algorithm and with the original Source-DPOR algorithm, for an increasing number N of elements produced and consumed, and a buffer of size $\texttt{MAX} \geq N$. The last two columns show the reduction gains in *Execs* and *Time* obtained by our algorithm. Times are obtained on an Intel Core I7 at 3.4Ghz with 16GB of RAM (Linux Kernel 4.4). Our algorithm is able to obtain the exact number (2^N) of non-equivalent executions. Furthermore, it is able to detect the equivalent sequences of executions as soon as they happen. For instance, in the example of Fig. 1, it detects that sequence $p_1.c_1.p_2$ leads to the same state as the previously explored $p_1.p_2.c_1$, and thus blocks it at this point. We therefore observe the claimed exponential reductions, not only in number of explored sequences, but also in the number of explored states and execution time.

6.2 Implementation Details

Computing the over-approximation for optimization 1: Our analysis computes the over-approximation O of possibly reachable tasks from the current state s as follows: Using the flow graph of the program, we compute the set of task names reachable from the enabled tasks in s. We also compute the set of references of *alive* actors in s, which includes the references of actors with pending tasks, actors in parameters of pending tasks, and actors stored in fields. The set O of reachable tasks from s is obtained by combining each task name with each compatible alive actor.

Avoiding recomputations: Our algorithm recomputes sub-sequences due to the context-sensitive equivalence check between the current sequence $E.p$ and the one $E'.v.u$ that reverses the race (line 13). When the algorithm later backtracks to E', it may eventually recompute the same sequence $E'.v.u$, except for the last step if the check succeeded. Our implementation avoids these recomputations as follows: In line 13, instead of checking the context sensitive equivalence of $v.u$, it adds $v.u$ together with the state $S_{[E.p]}$ to the sleep set of E'. For efficiency, in our implementation this is done right after executing event p, so that the state stored is the current one. Hence, a sequence t in the sleep set with attached state s is interpreted as "if we execute t and reach state s, then we block the sequence and add an enabled event to the backtrack set of the previous state if possible". This guarantees we do not recompute any single step due to context-sensitive checks. Note that in actor based systems the shared state between actors is typically small, therefore we never store full states, only local ones. Also, our experiments show that the peak numbers of stored local states remains quite low (see column M in Table 2). Alternatively, the check can be implemented by recording the state changes from E' to $E.p$ and comparing them against those from E' to $E'.v.u$. This would require a bounded amount of memory which can be reused for every equivalence check.

6.3 Experimental evaluation

Table 2 shows our experimental results, which compare the performance of the original source-DPOR algorithm with three versions of our context-sensitive approach: CDPOR1 is the basic algorithm without any of the optimizations in Section 5.3, CDPOR2 applies the first optimization, while CDPOR3 applies both optimizations. The comparison is performed on 6 classical concurrent actor programs, borrowed from [14], each executed with 3 (size increasing) input parameters. All benchmarks can be found at the SYCO web interface. The data shown in each set of columns (*Execs*, *Time*) is computed as before. For CDPOR3 we include an additional column (M) to show the peak amount of additional memory used (measured in number of stored local states) due to the *avoiding recomputations* approach mentioned above.

The last three columns show the gains in time obtained by each version of our algorithm over the original source-DPOR algorithm. A timeout of 120s is used and, when reached, we write $>X$ to indicate that for the corresponding measure we encountered X units up to that point. Thus, $>X$ indicates that the measure is at least X.

Table 2 shows that the less optimized implementation CDPOR1 is at least 1.3 times faster (Reg(5)) than Source-DPOR, and can be almost 3 orders of magnitude (PSort(5)) faster. The gain is much larger using the optimizations, in which case we achieve up to 4 orders of magnitude speedups. In some cases, the main reduction is achieved by the first optimization (e.g., compare G2 and G3 in PSort), while in most cases it is achieved by the second one (e.g., see Reg). The most important observation, however, is that the gain increases exponentially in all examples with the size of the input, in all three versions of our implementation. This experimentally justifies our claims about the exponential gains made in Section 4.

7 Conclusions

We have presented a novel technique that can be incorporated to state-of-the-art DPOR algorithms [1,7,14] to further reduce the number of redundant sequences explored. The crux of our method is the dynamic detection and use of context-sensitive independence, which allows proving independence of execution steps

	Source-DPOR		**CDPOR1**		**CDPOR2**		**CDPOR3**			**Reduction gains**		
Bench.	**Execs**	**Time**	**Execs**	**Time**	**Execs**	**Time**	**Execs**	**Time**	**M**	**G1**	**G2**	**G3**
Fib(5)	94	93	26	64	26	50	1	39	7	1.5x	1.9x	2.4x
Fib(6)	2148	2935	256	1407	256	683	1	39	12	2.1x	4.3x	75.3x
Fib(7)	56735	>120s	7929	>120s	11924	43637	1	124	20	-	>2.7x	>967.7x
QSort(9)	84	99	13	57	7	20	1	23	7	1.7x	5.0x	4.3x
QSort(12)	280	356	26	176	26	68	1	24	9	2.0x	5.2x	14.8x
QSort(15)	3166	3940	177	1132	87	249	1	40	12	3.5x	15.8x	98.5x
MSort(9)	256	259	14	84	14	32	1	18	8	3.1x	8.1x	14.4x
MSort(12)	912	1187	33	470	23	98	1	37	11	2.5x	12.1x	32.1x
MSort(15)	15872	36653	135	2051	135	374	1	51	14	17.9x	98.0x	718.7x
Pi(5)	120	83	9	26	9	17	9	15	21	3.2x	4.9x	5.5x
Pi(6)	720	556	24	51	24	60	24	43	35	10.9x	9.3x	12.9x
Pi(7)	5040	4673	74	146	74	150	74	149	53	32.0x	31.2x	31.4x
PSort(4)	288	109	8	14	2	12	2	4	13	7.8x	9.1x	27.2x
PSort(5)	34560	11921	64	128	8	15	8	15	28	93.1x	794.7x	794.7x
PSort(6)	275358	>120s	1224	2598	72	128	72	129	53	>46.2x	>937.5x	>930.2x
Reg(4)	384	214	148	178	71	68	1	4	11	1.2x	3.1x	53.5x
Reg(5)	3840	2357	1047	1465	449	498	1	6	16	1.6x	4.7x	392.8x
Reg(6)	46080	39769	7920	13916	3145	4337	1	7	22	2.9x	9.2x	5681.3x

Table 2. Experimental evaluation

for the particular context encountered. As our experiments show, our method achieves exponential gains in a message-passing concurrency model. Although we have not yet evaluated it, we believe the benefits of our method for shared-memory programs with synchronized blocks of code should be similar as for message passing.

While our extension was performed on the Source-DPOR algorithm, in practice Optimal-DPOR is usually slower than Source-DPOR. Hence, we expect our context-sensitive algorithm to also be significantly faster than Optimal-DPOR.

Note that our context-sensitive extension could be applied directly to Optimal-DPOR. However, Optimal-DPOR only checks races at leaf nodes, which is unsuitable for our context sensitive check, since its too late to gain benefit. Efficiently combining them is not straightforward and it is left as future work. Further, we have shown our context-sensitive DPOR algorithm can achieve exponential gains over Source-DPOR (e.g., for the producer-consumer example).

Acknowledgements This work was funded partially by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R and by the CM project S2013/ICE-3006.

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. of POPL'14*, pages 373–384. ACM, 2014.
2. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

3. Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A Systematic Testing Tool for Concurrent Objects. In *Proc. of CC'16*, pages 269–270. ACM, 2016.
4. Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State Space Reduction Using Partial Order Techniques. *STTT*, 2(3):279–287, 1999.
5. Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proc. of POPL'05*, pages 110–121. ACM, 2005.
6. Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proc. of CAV'91*, volume 531 of *LNCS*, pages 176–185. Springer, 1991.
7. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.
8. Patrice Godefroid and Didier Pirottin. Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract). In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 438–449. Springer, 1993.
9. Philipp Haller and Martin Odersky. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theoretical Computer Science*, 410(2-3):202–220, February 2009.
10. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.
11. Shmuel Katz and Doron A. Peled. Defining Conditional Independence Using Collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.
12. Antoni W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course*, volume 255 of *LNCS*, pages 279–324. Springer, 1987.
13. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.
14. Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Proc. of FMOODS/FORTE'12*, volume 7273 of *LNCS*, pages 219–234. Springer, 2012.
15. Antti Valmari. On-the-Fly Verification with Stubborn Sets. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 397–408. Springer, 1993.