Partial Order Reduction for Event-driven Multi-threaded Programs

Pallavi Maiya[1], Rahul Gupta[1], Aditya Kanade[1], and Rupak Majumdar[2]

[1] Indian Institute of Science
[2] MPI-SWS

Abstract. Event-driven multi-threaded programming is fast becoming a preferred style of developing efficient and responsive applications. In this concurrency model, multiple threads execute concurrently, communicating through shared objects as well as by posting asynchronous events. In this work, we consider partial order reduction (POR) for this concurrency model. Existing POR techniques treat event queues associated with threads as shared objects and reorder every pair of events handled on the same thread even if reordering them does not lead to different states. We do not treat event queues as shared objects and propose a new POR technique based on a backtracking set called the *dependence-covering set*. Our POR technique reorders events handled by the same thread only if necessary. We prove that exploring dependence-covering sets suffices to detect all deadlock cycles and assertion violations defined over local variables. To evaluate effectiveness of our POR scheme, we have implemented a dynamic algorithm to compute dependence-covering sets. On execution traces of some Android applications, we demonstrate that our technique explores many fewer transitions —often orders of magnitude fewer— compared to exploration based on persistent sets, in which event queues are considered as shared objects.

1 Introduction

Event-driven multi-threaded programming is fast becoming a preferred style of concurrent programming in many domains. In this model, multiple threads execute concurrently, and each thread may be associated with an event queue. A thread may post events to the event queue of a target thread. For each thread with an event queue, an event-loop processes the events from its event queue in the order of their arrival. The event-loop runs the handler of an event only after the previous handler finishes execution but interleaved with the execution of all the other threads. Further, threads can communicate through shared objects; even event handlers executing on the same thread may share objects. This style of programming is a staple of developing efficient and responsive smartphone applications [22]. A similar programming model is also used in distributed message-passing applications, high-performance servers, and many other settings.

Stateless model checking [12] is an approach to explore the reachable state space of concurrent programs by exploring different interleavings systematically but without storing visited states. The scalability of stateless model checking depends crucially on partial order reduction (POR) techniques [32,27,11,8]. Stateless search with POR

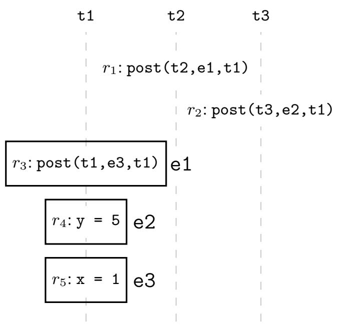


Fig. 1: A partial trace of an event-driven multi-threaded program

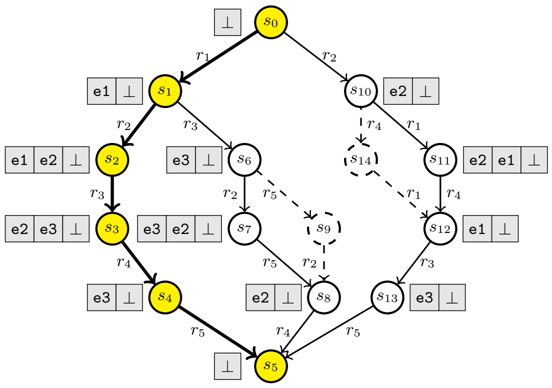


Fig. 2: The state space reachable through all valid permutations of operations in the trace in Figure 1

defines an equivalence class on interleavings, and explores only a representative interleaving from each equivalence class (called a Mazurkiewicz trace [21]), while still providing certain formal guarantees *w.r.t.* exploration of the complete but possibly much larger state space. Motivated by the success of model checkers based on various POR strategies [14,13,29,26,5,6,33,28], in this work, we propose an effective POR strategy for event-driven multi-threaded programs.

1.1 Motivation

We first show why existing POR techniques may not be very effective in the combined model of threads and events. Consider a partial execution trace of an event-driven multi-threaded program in Figure 1. The operations are labeled r_1 to r_5 and execute from top to bottom. Those belonging to the same event handler are enclosed within a box labeled with the corresponding event ID. An operation `post(t,e,t′)` executed by a thread `t` enqueues an event `e` to the event queue of a thread `t′`. In our trace, threads `t2` and `t3` respectively post events `e1` and `e2` to thread `t1`. The handler of `e1` posts an event `e3` to `t1`, whereas, those of `e2` and `e3` respectively write to shared variables `y` and `x`.

Figure 2 shows the state space reachable through all valid permutations of operations in the trace in Figure 1. Each node indicates a state of the program. An edge is labeled with an operation and indicates the state transition due to that operation. The interleaving corresponding to the trace in Figure 1 is highlighted with bold lines and shaded states. For illustration purposes, we explicitly show the contents of the event queue of thread `t1` at some states. Events in a queue are ordered from left to right and a box containing $\perp$ indicates the rear end of the queue.

Existing POR techniques (e.g., [11,10,29,31,5]) recognize that r_2 and r_5 (also r_1 and r_4) are independent (or non-interfering) and that it is sufficient to explore any one of them at state s_6 (respectively, s_{10}). The dashed edges indicate the unexplored transitions. However, existing POR-based model checkers will explore all other states and transitions. Since no two handlers executed on `t1` modify a common object, all the interleavings reach the same state s_5. Thus, existing techniques explore two redundant

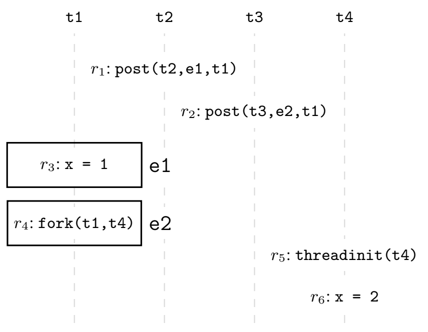


Fig. 3: A partial trace w of a program involving a multi-threaded dependence

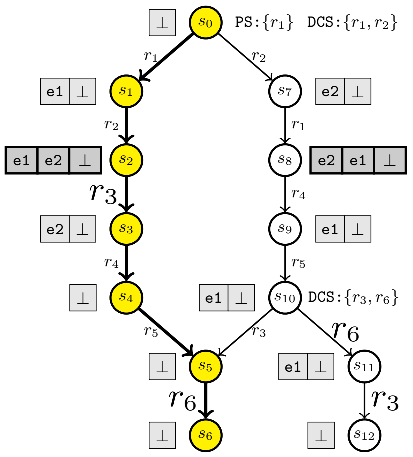


Fig. 4: A transition system for some valid permutations of operations in the trace in Figure 3

interleavings because they treat *event queues as shared objects* and so, mark any two `post` operations that enqueue events to the event queue of the same thread as dependent. Consequently, they explore both r_1 and r_2 at state s_0, and r_2 and r_3 at state s_1. These result in unnecessary reorderings of events. More generally, if there are n events posted to an event queue, these techniques may explore $O(n!)$ permutations among them, even if exploring only one of them may be sufficient.

1.2 Contributions of This Paper

Based on the observation above, we do *not* consider event queues as shared objects. Equivalently, we treat a pair of `posts` even to the same thread as *independent*. This enables more reductions. For example, for the state space in Figure 2, our approach explores only the initial trace (the leftmost interleaving).

Since we shall not reorder every pair of events posted to the same thread by default, the main question is "How to determine which events to reorder and how to reorder them selectively?". Surely, if two handlers executing on the same thread contain dependent transitions then we must reorder their `post` operations, but this is not enough. To see this, consider a partial trace w in Figure 3. The transitions r_3 and r_6 are dependent and belong to *different* threads. Figure 4 shows a transition system depicting a partial state space explored by different orderings of r_3 and r_6. The contents of thread `t1`'s queue are shown next to each state. As can be seen in the rightmost interleaving, executing r_6 before r_3 requires posting `e2` *before* `e1` even though their handlers do not have dependent transitions. Thus, operations posting events to the same thread may have to be reordered even to reorder some multi-threaded dependences! Our first contribution is to define a relation that captures both single-threaded and multi-threaded dependences.

We now discuss what the implications of (1) treating `posts` as independent and (2) only selectively reordering them are. For multi-threaded programs, or when `posts` are considered dependent, reordering a pair of adjacent independent transitions in a transition sequence does not affect the reachable state. Hence, the existing dependence relation [11] induces equivalence classes where transition sequences differing only in the order of executing independent transitions are in the same Mazurkiewicz trace [21]. However, our new dependence relation (where `posts` are considered independent) may not induce Mazurkiewicz traces on an event-driven multi-threaded program. First, reordering `posts` to the same thread affects the order of execution of the corresponding handlers. If the handlers contain dependent transitions, it affects the reachable state. Second, one cannot rule out the possibility of *new* transitions (not present in the given transition sequence) being pulled in when independent `posts` are reordered, which is not admissible in a Mazurkiewicz trace. We elaborate on this in Section 2.3.

Our second contribution is to define a notion of *dependence-covering sequence* to provide the necessary theoretical foundation to reason about reordering `posts` selectively. Intuitively, a transition sequence u is a dependence-covering sequence of a transition sequence u' if the relative ordering of all the pairs of dependent transitions in u' is preserved in u. While this sounds similar to the property of any pair of transition sequences in the same Mazurkiewicz trace, the constraints imposed on a dependence-covering sequence are more relaxed (as will be formalized in Definition 4), making it suitable to achieve better reductions. For instance, u is permitted to have *new* transitions, that is, transitions that are not in u', under certain conditions.

Given a notion of POR, a model checking algorithm such as DPOR [10] uses persistent sets [11] to explore representative transition sequences from each Mazurkiewicz trace. As we show now, DPOR based on persistent sets is *unsound* when used with the dependence relation in which `posts` are independent. Let us revisit Figure 4. The set $\{r_1\}$ is persistent at state s_0 because exploring any other transition from s_0 does not reach a transition dependent with r_1. This set is tagged as `PS` in the figure. A selective exploration using this set explores only one ordering between r_3 and r_6.

Our final contribution is the notion of *dependence-covering sets* as an alternative to persistent sets. A set of transitions L at a state s is said to be dependence-covering if for any sequence u' executed from s, a dependence-covering sequence u starting with some transition in L can be explored. We prove that selective state-space exploration based on dependence-covering sets is sufficient to detect all deadlock cycles and violations of assertions over local variables. The dependence-covering sets at certain states are marked in Figure 4 as `DCS`. In contrast to `PS`, `DCS` at state s_0 contains *both* r_1 and r_2. Let u' be the transition sequence along the rightmost interleaving in Figure 4. The sequence w (the leftmost interleaving) is not a dependence-covering sequence of u' since the dependent transitions r_3 and r_6 appear in a different order. We therefore require r_2 to be explored at s_0. Note that, $\{r_2\}$ is another dependence-covering set at s_0 as both the orderings of r_3 and r_6 can be explored from a state s_{10} reached on exploring r_2.

We have implemented a proof-of-concept model checking framework called EM-Explorer which simulates the non-deterministic behaviour exhibited by Android applications given individual execution traces. We implemented a dynamic algorithm to compute dependence-covering sets and a selective state-space exploration based on these sets in EM-Explorer. For comparison, we also implemented an exploration based on DPOR, where `posts` to the same thread are considered dependent. We performed experiments on traces obtained from 5 Android applications. Our results demonstrate that our POR explores many fewer transitions —often orders of magnitude fewer— compared to DPOR using persistent sets.

1.3 Related Work

Mazurkiewicz traces induced by an independence relation form the foundation for most existing work on POR, and most prior work in the event-driven setting consider operations on event queues to be dependent. For example, Sen and Agha [29] and Tasharofi et al. [31] describe dynamic POR techniques for distributed programs with actor semantics where processes (actors) communicate only by asynchronous message passing (and do not have shared variables). Both techniques explore all possible interleavings of messages sent to the same process. Basset [17], a framework for state space exploration of actor systems, uses the DPOR algorithm described in [29], resulting in exploration of all valid configurations of messages sent to the same actor. In comparison, we explore only a subset of event orderings at each thread, and doing so requires relaxing Mazurkiewicz traces to dependence-covering sequences.

Recent algorithms guarantee optimality in POR [5,28], *i.e.*, they explore at most one transition sequence per Mazurkiewicz trace. For example, POR using source sets and wakeup trees [5] enables optimal exploration. However, the notion of source sets and the corresponding algorithm assume total ordering between transitions executed on the same thread. Hence, integrating our new dependence relation with source sets will involve significant changes to the definitions and algorithms presented in [5]. Rodríguez et al. [28] describe unfolding semantics parametrized on the commutativity based classical independence relation [11], and present an unfolding based optimal POR algorithm. The unfolding semantics identifies dependent transitions with no ordering relation between them to be in conflict. Their POR algorithm backtracks and explores a new transition sequence w from a state s only if every prior transition explored from s is in conflict with some transition in w. This is problematic in our setting where posts are considered independent and hence trivially non-conflicting, causing unfolding based POR to miss reordering `posts` when required. Establishing optimality in our setting is an interesting but non-trivial future direction.

R^4 [16] is a stateless model checker for event-driven programs like client-side web applications. R^4 adapts persistent sets and DPOR algorithm to the domain of single-threaded, event-driven programs with multiset semantics. Each event handler is atomically executed to completion without interference from other handlers, and thus an entire event handler is considered a single transition. In contrast, our work focuses on POR techniques for multi-threaded programs with event queues, and thus needs to be sensitive to interference from other threads while reordering dependent transitions.

In many domains, the event-loop works in FIFO order, as is also considered in this work. For example, Android [15,19], TinyOS [4], Java AWT [3], and Apple's Grand Central Dispatch [2], provide a default FIFO semantics. Abstracting FIFO order by the multiset semantics, as in [30,9], can lead to false positives. There is a lot of recent work on concurrency analysis for smartphone environments. For example, [15,19,7] provide algorithms for race detection for Android applications. Our work continues this line by describing a general POR technique. We note that the event dispatch semantics can be diverse in general. For example, Android applications permit posting an event with a timeout or posting a specific event to the front of the queue. We over-approximate the effect of posting with timeout by forking a new thread which does the `post` non-deterministically but do not address other variants in this work. We leave a more general POR approach that allows such variants to event dispatch to future work.

2 Formalization

We consider event-driven multi-threaded programs comprising the usual sequential and multi-threaded operations such as assignments, conditionals, synchronization through locks, and thread creation. In addition, the operation $\texttt{post}(t_1, e, t_2)$ posts an asynchronous event e from the source thread t_1 to (the event queue of) a destination thread t_2. Each event has a handler, which runs to completion on its thread but may interleave with operations from other threads. An operation is *visible* if it accesses an object shared between at least two threads or two event handlers (possibly running on the same thread). All other operations are *invisible*. We omit the formal syntax and semantics of these operations; they can be found in [19].

The *local state of an event handler* is a valuation of the stack and the variables or heap objects that are modified only within the handler. The local state of a thread is the local state of the currently executing handler. A *global state* of the program A is a valuation to the variables and heap objects that are accessed by multiple threads or multiple handlers. Even though event queues are shared objects, we do not consider them in the global state (as defined above). Instead, we define a *queue state of a thread* as an ordered sequence of events that have been posted to its event queue but are yet to be handled. This separation allows us to analyze dependence more precisely. Event queues are FIFO queues with unbounded capacity, that is, a `post` operation never blocks. For simplicity, we assume that each thread is associated with an event queue.

2.1 Transition System

Consider an event-driven multi-threaded program A. Let L, G, and Q be the sets of local, global and queue states respectively. Let T be the set of all threads in A. A *state* s of A is a triple (l, g, q) where (1) l is a partial map from T to L, (2) g is a global state and (3) q is a total map from T to Q. A *transition* by a thread t updates the state of A by performing one visible operation followed by a finite sequence of invisible operations ending just before the next visible operation; all of which are executed on t. Let R be the set of all transitions in A. A transition $r_{t,\ell}$ of a thread t at its local state ℓ is a partial function, $r_{t,\ell} : G \times Q \mapsto L \times G \times Q$. A transition $r_{t,\ell} \in R$ is enabled at a state $s = (l, g, q)$ if $\ell = l(t)$ and $r_{t,\ell}(g, q)$ is defined. Note that the first transition of the handler of an event e enqueued to a thread t is *enabled* at a state s, if e is at the front of t's queue at s and t is not executing any other handlers. We may use $r_{t,\ell}(s)$ to denote application of a transition $r_{t,\ell}$, instead of the more precise use $r_{t,\ell}(g, q)$.

We formalize the state space of A as a *transition system* $\mathcal{S}_G = (S, s_{init}, \Delta)$, where S is the set of all states, $s_{init} \in S$ is the initial state, and $\Delta \subseteq S \times S$ is the transition relation such that $(s, s') \in \Delta$ iff $\exists r \in R$ and $s' = r(s)$. We also use $s \in \mathcal{S}_G$ instead of $s \in S$. Two transitions r and r' *may be co-enabled* if there may exist some state $s \in S$

where they both are enabled. Two events e and e' handled on the same thread t *may be reordered* if $\exists s, s' \in S$ reachable from s_{init} such that $s = (l, g, q)$, $s' = (l', g', q')$, $q(t) = e \cdot w \cdot e' \cdot w'$ and $q'(t) = e' \cdot v \cdot e \cdot v'$. In Figure 2, events `e1` and `e2` may be reordered but not `e1` and `e3`.

In our setting, if a transition is defined for a state then it maps the state to a successor state deterministically. For simplicity, we assume that all threads and events in A have unique IDs. We also assume that the transition system is *finite* and *acyclic*. This is a standard assumption for stateless model checking [10]. The transition system $\mathcal{S}_G$ collapses invisible operations and is thus already reduced when compared to the transition system in which even invisible operations are considered as separate transitions. A transition system of this form is sufficient for detecting deadlocks and assertion violations [12].

Notation. Let $next(s, t)$ give the next transition of a thread t in a state s. Let $thread(r)$ return the thread executing a transition r. If r executes in the handler of an event e on thread t then the *task* of r is $task(r) = (t, e)$. A transition r on a thread t is *blocked* at a state s if $r = next(s, t)$ and r is not enabled in s. We assume that only visible operations may block. Function $nextTrans(s)$ gives the set of next transitions of all threads at state s. For a transition sequence $w : r_1.r_2 \ldots r_n$ in $\mathcal{S}_G$, let $dom(w) = \{1, \ldots, n\}$. Functions $getBegin(w, e)$ and $getEnd(w, e)$ respectively return the indices of the first and the last transitions of an event e's handler in w, provided they belong to w.

Deadlock cycles and assertion violations. A pair $\langle DC, \rho \rangle$ in a state $s \in S$ is said to form a *deadlock cycle* if $DC \subseteq nextTrans(s)$ is a set of n transitions blocked in s, and ρ is a one-to-one map from $[1, n]$ to DC such that each $\rho(i) \in DC$, $i \in [1, n]$, is blocked by some transition on a thread $t_{i+1} = thread(\rho(i + 1))$ and may be enabled only by a transition on t_{i+1}, and the transition $\rho(n) \in DC$ is blocked and may be enabled by two different transitions of thread $t_1 = thread(\rho(1))$. A state s in $\mathcal{S}_G$ is a *deadlock* state if all threads are blocked in s due to a deadlock cycle.

An *assertion* α is a predicate over local variables of a handler and is considered visible. A state s *violates* an assertion α if α is enabled at s and evaluates to *false*.

2.2 Dependence and Happens-before Relations

The notion of dependence between transitions is well-understood for multi-threaded programs. It extends naturally to event-driven programs if event queues are considered as shared objects, thereby, marking two `posts` to the same thread as dependent. To enable more reductions, we define an alternative notion in which two `post` operations to the same thread are *not* considered dependent. One reason to selectively reorder events posted to a thread is if their handlers contain dependent transitions. This requires a new notion of dependence between transitions of event handlers executing on the same thread, which we refer to as *single-threaded dependence*.

In order to explicate single-threaded dependences, we first define an event-parallel transition system which over-approximates the transition system $\mathcal{S}_G$. The *event-parallel transition system* $\mathcal{P}_G$ of a program A is a triple $(S_P, s_{init}, \Delta_P)$. In contrast to the transition system $\mathcal{S}_G = (S, s_{init}, \Delta)$ of Section 2.1 where events are dispatched in their order of arrival and execute till completion, a thread with an event queue in $\mathcal{P}_G$ removes *any* event in its queue and spawns a fresh thread to execute its handler. This

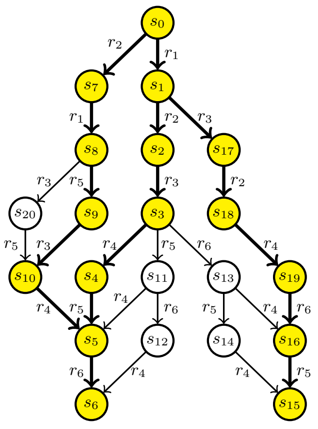
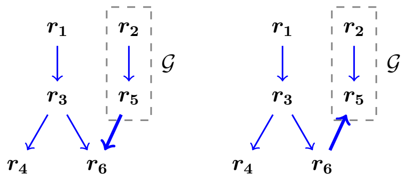


Fig. 6: Partial event-parallel transition system for the program in Figure 5.

```
r1: post(t1,e1,t); // runs on thread t1
r2: post(t2,e2,t); // runs on thread t2
h1 := {r3: post(t,e3,t); r4: y = 2;}
h2 := {r5: x = 5;}
h3 := {r6: x = −5;}
```

Fig. 5: Pseudo code of an event-driven program.



(a) w_1: $r_1.r_2.r_3.r_4.r_5.r_6$, w_2: $r_2.r_1.r_5.r_3.r_4.r_6$ (b) w_3: $r_1.r_3.r_2.r_4.r_6.r_5$

Fig. 7: Dependence graphs of some sequences in $\mathcal{S}_G$ of the program in Figure 5.

enables concurrent execution of events posted to the same thread. Rest of the semantics remains the same. Let T and T_P be the sets of all threads in $\mathcal{S}_G$ and $\mathcal{P}_G$ respectively. For each state $(l, g, q) \in S$, there exists a state $(l', g', q') \in S_P$ such that (1) for each thread $t \in T$, if $l(t)$ is defined then there exists a thread $t' \in T_P$ where $l'(t') = l(t)$, (2) $g = g'$, and (3) for each thread $t \in T$, $q(t) = q'(t)$. Let R_P be the set of transitions in $\mathcal{P}_G$ and $ep : R \to R_P$ be a total function which maps a transition $r_{t,\ell} \in R$ to an equivalent transition $r'_{t',\ell'} \in R_P$ such that $\ell = \ell'$ and either $t' = t$ or t' is a fresh thread spawned by t in $\mathcal{P}_G$ to handle the event to whose handler $r_{t,\ell}$ belongs in $\mathcal{S}_G$.

We illustrate the event-parallel transition system for the example program in Figure 5. Here, `x` and `y` are shared variables. The transitions r_1 and r_2 respectively run on threads `t1` and `t2`. The last three lines in Figure 5 give definitions of handlers of the events `e1`, `e2` and `e3` respectively. Figure 6 shows a partial state space of the program in Figure 5 according to the event-parallel transition system semantics. The edges are labeled with the respective transitions. The shaded states and thick edges indicate part of the state space that is reachable in the transition system semantics of Section 2.1 as well, under the mapping between states and transitions described above.

Definition 1. Let R_P be the set of transitions in the event-parallel transition system $\mathcal{P}_G$ of a program A. Let $D_P \subseteq R_P \times R_P$ be a binary, reflexive and symmetric relation. The relation D_P is a valid ***event-parallel dependence relation*** iff for all $(r_1, r_2) \in R_P \times R_P$, $(r_1, r_2) \notin D_P$ implies that the following conditions hold for all states $s \in S_P$:

1. If r_1 is enabled in s and $s' = r_1(s)$ then r_2 is enabled in s iff it is enabled in s'.
2. If r_1 and r_2 are both enabled in s then there exists $s' = (l', g', q') = r_1(r_2(s))$ and $s'' = (l'', g'', q'') = r_2(r_1(s))$ such that $l' = l''$ and $g' = g''$.

This definition is similar to the definition of dependence relation in [12] except that we do *not* require equality of the event states q' and q'' in the second condition above.

Clearly, any pair of `post` transitions, even if posting to the same event queue, are *independent* according to the event-parallel dependence relation.

Definition 2. Let R be the set of transitions in the transition system $\mathcal{S}_G$ of a program A. Let D_P be a valid event-parallel dependence relation for A and $D \subseteq R \times R$ be a binary, reflexive and symmetric relation. The relation D is a valid ***dependence relation*** iff for all $(r_1, r_2) \in R \times R$, $(r_1, r_2) \notin D$ implies that the following conditions hold:

1. If r_1 and r_2 are transitions of handlers of two different events e_1 and e_2 executing on the *same thread* then the following conditions hold:
   (A) Events e_1 and e_2 may be reordered in $\mathcal{S}_G$.
   (B) $ep(r_1)$ and $ep(r_2)$ are independent in D_P, *i.e.*, $(ep(r_1), ep(r_2)) \notin D_P$.
2. Otherwise, conditions 1 and 2 in Definition 1 hold for all states $s \in S$.

Condition 1 above uses D_P to define single-threaded dependence between transitions of two handlers in $\mathcal{S}_G$. Condition 2 applies the constraints in Definition 1 to states in $\mathcal{S}_G$ to define (1) dependence among transitions of the same handler and (2) multi-threaded dependence. Hence, all `posts` are considered *independent* of each other in $\mathcal{S}_G$.

Example 1. The transitions r_5 and r_6 in Figure 5 run in two different event handlers but on the same thread `t`. Since the handlers execute concurrently in the event-parallel transition system, we can inspect the effect of reordering r_5 and r_6 on a state where they are co-enabled. In particular, at state s_3 in Figure 6, the sequence $r_6.r_5$ reaches state s_{14}, whereas, $r_5.r_6$ reaches s_{12} which differs from s_{14} in the value of `x`. Therefore, $(r_5, r_6) \in D_P$ and by condition 1.B of Definition 2, $(r_5, r_6) \in D$.

The condition 1.A of Definition 2 requires that the ordering between e_1 and e_2 should not be fixed. Suppose the handler of e_1 posts e_2 but the two handlers do not have any pair of transitions that are in D_P. Nevertheless, since a `post` transition in e_1's handler *enables* e_2, the transitions in the two handlers should be marked as dependent. This requirement is met through condition 1.A.

If $(r_i, r_j) \in D$, we simply say that r_i and r_j are *dependent*. In practice, we over-approximate the dependence relation, for example, by considering all conflicting accesses to shared objects as dependent. We now extend the happens-before relation defined in [10] with the FIFO rule in [15,19].

Definition 3. The ***happens-before relation*** $\rightarrow_w$ for a transition sequence $w : r_1.r_2 \ldots r_n$ in $\mathcal{S}_G$ is the smallest relation on $dom(w)$ such that the following conditions hold:

1. If $i < j$ and r_i is dependent with r_j then $i \rightarrow_w j$.
2. If r_i and r_j are transitions posting events e and e' respectively to the same thread, such that $i \rightarrow_w j$ and the handler of e has finished and that of e' has started in w, then $getEnd(w, e) \rightarrow_w getBegin(w, e')$. This is the FIFO rule.
3. $\rightarrow_w$ is transitively closed.

The relation $\rightarrow_w$ is defined over transitions in w. We overload $\rightarrow_w$ to also relate transitions in w with those in the $nextTrans$ set in the last state, say s, reached by w. For a task (t, e) having a transition in $nextTrans(s)$, $i \rightarrow_w (t, e)$ if either (a) $task(r_i)$ $= (t, e)$ or (b) $\exists k \in dom(w)$ such that $i \rightarrow_w k$ and $task(r_k) = (t, e)$.

2.3 Dependence-covering Sets

Mazurkiewicz trace [21] forms the basis of POR for multi-threaded programs and event-driven programs where `posts` are considered dependent. Two transition sequences belong to the same Mazurkiewicz trace if they can be obtained from each other by reordering adjacent independent transitions. The objective of POR is to explore a representative sequence from each Mazurkiewicz trace. As pointed out in the Introduction, the reordering of `posts` (independent as per Definition 2) in a transition sequence w may not yield another sequence belonging to the same Mazurkiewicz trace (denoted $[w]$) for two reasons: (1) it may reorder dependent transitions from the corresponding handlers and (2) some new transitions, not in w, may be pulled in.

We elaborate on the second point. Suppose in w, a handler h_1 executes before another handler h_2, both on the same thread, such that h_2 is executed only partially in w. Let us reorder the `post` operations for these two and obtain a transition sequence w'. Since the handlers run to completion, in order to include all the transitions of h_1 (executed in w) in w', we must complete execution of h_2. However, as h_2 is only partially executed in w, this results in including *new* —previously unexplored— transitions of h_2 in w'. This renders w and w' inequivalent by the notion of Mazurkiewicz equivalence.

We therefore propose an alternative notion, suitable to correlate two transition sequences in event-driven multi-threaded programs, called the *dependence-covering sequence*. The objective of our reduction is to explore a dependence-covering sequence u at a state s for any transition sequence w starting at s.

Let $w : r_1.r_2 \ldots r_n$ and $u : r'_1.r'_2 \ldots r'_m$ be two transition sequences from the same state s in $\mathcal{S}_G$ reaching states s_n and s'_m respectively. Let $R_w = \{r_1, \ldots, r_n\}$ and $R_u = \{r'_1, \ldots, r'_m\}$.

Definition 4. The transition sequence u is called a ***dependence-covering sequence*** of w if (i) $R_w \subseteq R_u$ and (ii) for each pair of dependent transitions $r'_i, r'_j \in R_u$ such that $i < j$, any one among the following conditions holds:

1. r'_i and r'_j are executed in w and their relative order in u is consistent with that in w.
2. r'_i is executed in w and $r'_j \in nextTrans(s_n)$.
3. r'_i is not executed in w, $r'_j \in nextTrans(s_n)$ and w can be extended in $\mathcal{S}_G$ such that r'_i executes before r'_j.
4. Irrespective of whether r'_i is executed in w or not, r'_j is not in $R_w \cup nextTrans(s_n)$.

The condition (i) above allows new transitions, that are not in w, to be part of u. The condition (ii) restricts how the new transitions may interfere with the dependences exhibited in w and also requires all the dependences in w to be maintained in u. These conditions permit a dependence-covering sequence of w to be a relaxation of Mazurkiewicz trace $[w]$, making it more suitable for stateless model checking of event-driven multi-threaded programs where `posts` may be reordered selectively.

As an example, let w_1, w_2 and w_3 (listed in Figure 7) be the three transition sequences in Figure 6 which correspond to valid sequences in the transition system $\mathcal{S}_G$ of the program in Figure 5. To illustrate dependence-covering sequences, we visualize the dependences in these sequences as dependence graphs. The nodes in the dependence graph of a sequence w represent transitions in w. If a transition r_i executes before another transition r_j in w such that r_i and r_j are dependent then a directed edge is drawn from r_i to r_j. Figure 7 depicts the dependence graphs of w_1, w_2 and w_3. Sequences w_1 and w_2 are dependence-covering sequences of each other, as their dependence graphs are identical (Figure 7(a)). Consider a sequence $w_4 = r_2.r_5$ whose dependence graph is the subgraph $\mathcal{G}$ in Figure 7. A dependence graph $\mathcal{G}'$ of any dependence-covering sequence u of w_4 contains $\mathcal{G}$ as a subgraph, with no incoming edge into $\mathcal{G}$ from any node in $\mathcal{G}'$ which is not in $\mathcal{G}$. However, there are no restrictions on dependences between nodes in $\mathcal{G}'$ which are not in $\mathcal{G}$. Hence, w_1 and w_2 (see Figure 7(a)) are dependence-covering sequences of w_4 even though w_4 and w_1 (or w_2) do not belong to the same Mazurkiewicz trace, whereas w_3 is not a dependence-covering sequence of w_4 due to the edge r_6 to r_5 (see Figure 7(b)).

Definition 5. A non-empty subset L of transitions enabled at a state s in $\mathcal{S}_G$ is a ***dependence-covering set*** in s iff, for all non-empty sequences of transitions $w : r_1 \ldots r_n$ starting at s, there exists a dependence-covering sequence $u : r'_1 \ldots r'_m$ of w starting at s such that $r'_1 \in L$.

Example 2. All the transition sequences connecting state s_0 to state s_5 in Figure 2 are dependence-covering sequences of each other. Thus, the dependence-covering set in s_0 can be $\{r_1\}$, $\{r_2\}$ or $\{r_1, r_2\}$. Even if we take a prefix σ of any of these sequences, the shaded sequence in Figure 2 is a dependence-covering sequence of σ.

In Figure 4, $\{r_2\}$ and $\{r_1, r_2\}$ are individually dependence-covering sets in state s_0, whereas, $\{r_1\}$ is not a dependence-covering set at s_0.

For efficient stateless model checking of event-driven multi-threaded programs, we can explore a reduced state space using dependence-covering sets.

Definition 6. A ***dependence-covering state space*** of an event-driven multi-threaded program A is a reduced state space $\mathcal{S}_R \subseteq \mathcal{S}_G$ obtained by exploring only the transitions in a dependence-covering set at each state in $\mathcal{S}_G$ reached from s_{init}.

The objective of a POR approach is to show that even while exploring a reduced state space, no concurrency bug is missed *w.r.t.* the complete but possibly much larger state space. The exploration of a dependence-covering state space satisfies this objective. The following theorem states the main result of this paper.

Theorem 1. Let $\mathcal{S}_R$ be a dependence-covering state space of an event-driven multi-threaded program A with a finite and acyclic state space $\mathcal{S}_G$. Then, all deadlock cycles in $\mathcal{S}_G$ are reachable in $\mathcal{S}_R$. If there exists a state v in $\mathcal{S}_G$ which violates an assertion α defined over local variables then there exists a state v' in $\mathcal{S}_R$ which violates α.

The proof follows from the appropriate restrictions on allowed dependences in a dependence-covering sequence u compared to the dependences in w where w is required to reach a deadlock cycle or an assertion violation in the complete state space. We provide a complete proof of the above theorem in Appendix A in [18].

The set $\{r_1, r_2\}$ is both a persistent set and a dependence-covering set in state s_0 in Figure 2. We observe that in general, a persistent set P at a state $s \in \mathcal{S}_G$ is also a dependence-covering set at s. Here, persistent set is defined using the dependence relation where `posts` to the same event queue are dependent, whereas, dependence-covering set is defined using the dependence relation where they are not (more formally, using Definition 2). We present a proof of this claim in Appendix B in [18]. Note that a dependence-covering set need not be a persistent set. As seen in Example 2, $\{r_1\}$ and $\{r_2\}$ are both dependence-covering sets at s_0 in Figure 2 but they are not persistent sets.

3 Implementation

This section provides a high-level sketch of our algorithm to dynamically compute dependence-covering sets. Due to lack of space, we do not present the complete algorithm and its soundness proof. We refer the readers to [18] for these details.

Android applications form a very popular class of event-driven multi-threaded programs. We further discuss the implementation of EM-Explorer, a proof-of-concept model checking framework which simulates the Android concurrency semantics given an execution trace of an Android application. We have implemented our algorithm in EM-Explorer for experimental evaluation.

3.1 Dynamic Algorithm for Computing Dependence-covering Sets

DPOR [10] is an effective algorithm to dynamically compute backtracking choices for selective state-space exploration. It performs DFS traversal of the transition system of a program, but instead of exploring all the enabled transitions at a state, it only explores transitions added as backtracking choices by the steps of the algorithm. DPOR guarantees that it explores a persistent set at each of the visited states. Our algorithm, called *EM-DPOR*, extends DPOR to compute dependence-covering sets for event-driven multi-threaded programs. It differs from DPOR in the aspects of computing backtracking choices as well as the states to which the algorithm backtracks.

Let $backtrack(s)$ refer to the backtracking set computed by EM-DPOR for a state s. We say a task (t, e) is *executable* at a state s if the first transition of event e's handler is enabled in s (see Section 2.1) or the task is being executed in state s. Similar to DPOR, on exploring a sequence w reaching a state s', our algorithm identifies the nearest transition r in w executed at a state s which is dependent with a transition $r' \in nextTrans(s')$. If r and r' belong to two different threads then similar to DPOR, we require that they may be co-enabled. In addition, if they belong to two different handlers on the same thread then we require that they may be reordered (see Section 2.1). In the latter case, we first identify a pair of `post` operations executed on *different* threads which need to be reordered so as to reorder r and r' (see [18] for the details).

We now discuss how to reorder two transitions from two different threads. Let r and r' be such transitions where r executes before r' in w and s be the state from which r executes. In order to compute backtracking choices to reorder them, EM-DPOR computes a set of candidate tasks whose respective threads are enabled in state s, such that each *candidate task* contains a transition executed after r that has happens-before relation with r'. We select a thread t as a backtracking choice if (t, e) is a candidate task and t is not already explored from state s. These steps are similar to the steps of DPOR except that we use the happens-before relation of Definition 3.

A challenging case arises if the threads corresponding to all the candidate tasks are already explored from s. As will be illustrated in Example 3, this does not imply that r

and r' cannot be reordered in future. Unless all candidate tasks are executable at state s, this also does not imply that they have been reordered in the already explored state space. Therefore, the algorithm selects some candidate task (t', e') (if any) that is not executable at s and attempts to reorder e' with the event, say e'', enqueued to the same thread t' and executable at s. This results in a *recursive* call to identify backtracking choices as well as backtracking state. Due to this, in a future run, e'' and e' may be reordered. Suppose r'' is the transition in (t', e') that has happens-before with r'. The resulting reordering of the handlers of e'' and e' subsequently enables exploring r'' prior to r which in turn, leads to the desired reordering of r and r'. Example 3 illustrates the essential steps of EM-DPOR.

Example 3. We explain how EM-DPOR computes dependence-covering sets to explore the state space in Figure 4 starting with transition sequence w in Figure 3. On exploring a prefix of w and reaching state s_5, r_3 and r_6 are identified to be dependent. When attempting to compute backtracking choices at state s_2 (the state where r_3 is explored) to reorder r_3 and r_6, r_4 is found to have a happens-before ordering with r_6 and thus r_4's task (`t1,e2`) is identified as the candidate task. However, thread `t1` is already explored from state s_2 and task (`t1,e2`) is not executable (see the event queue shown at s_2). Hence, the algorithm tries to reorder `e2` with event `e1` whose task is executable at state s_2. This is achieved by recursively starting another backward search to identify the backtracking choices that can reorder `e1` and `e2`. In this case, `post` operations r_1 and r_2 can be reordered to do so. Therefore, r_2 is added to the backtracking set at s_0, exploring which leads to s_8 where `e2` *precedes* `e1` in the event queue as required. Thus, the algorithm computes $\{r_1, r_2\}$ as a dependence-covering set at s_0 and eventually reaches state s_{10} where r_3 and r_6 are co-enabled and can be ordered as desired.

Note that even while considering the dependence between transitions r_3 and r_6, EM-DPOR is able to identify a seemingly unrelated state s_0 (much prior to state s_2 where r_3 is explored) as an appropriate state to backtrack to. Also, r_3 and r_6 are transitions from two *different threads*. Even then, to reorder them, EM-DPOR had to reorder the two `post` transitions r_1 and r_2 to the same thread `t1` at s_0.

Similar to DPOR, we have implemented our algorithm using a vector clocks datastructure [20]. In a multi-threaded setting where all the operations executed on the same thread are totally ordered, a clock is assigned to each thread and the components of a vector clock correspond to clock values of the threads of the program. In an event-driven program, the operations from different handlers on the same thread need not be totally ordered and hence, we assign a clock to each task in the program. In addition to the vector clock computations described in [10], we establish an order between event handlers executed on the same thread if their corresponding `posts` have a happens-before ordering, so as to respect the FIFO ordering of events.

3.2 EM-Explorer Framework

Building a full-fledged model checker for Android applications is a challenge in itself but is not the focus of this work. Tools such as JPF-Android [24] and AsyncDroid [25] take promising steps in this direction. However, presently they either explore only a

Table 1: Statistics on execution traces from Android apps and their model checking runs using different POR techniques. Android apps: (A) Remind Me, (B) My Tracks, (C) Music Player, (D) Character Recognition, and (E) Aard Dictionary

Apps	Trace	Threads/	Memory	**DPOR**			**EM-DPOR**		
	length	Events	Locations	Traces	Transitions	Time	Traces	Transitions	Time
A	444	4/9	89	24	1864	0.18s	3	875	0.05s
B	453	10/9	108	1610684*	113299092*	4h*	405013	26745327	101m 30s
C	465	6/24	68	1508413*	93254810*	4h*	266	34333	4.15s
D	485	4/22	40	1284788	67062526	199m 28s	756	39422	6.58s
E	600	5/30	30	359961*	14397143*	4h*	14	4772	1.4s

limited number of sources of non-determinism [25] or require a lot of framework libraries to be modeled [24,23]. We have therefore implemented a prototype exploration framework called EM-Explorer, which emulates the semantics of visible operations like `post` operation, memory read/write, and lock acquire/release.

EM-Explorer takes an execution trace generated by an automated testing and race detection tool for Android applications, called DroidRacer [19], as input. As DroidRacer runs on real-world applications, we can experiment on real concurrency behaviors seen in Android applications and evaluate different POR techniques on them. DroidRacer records all concurrency relevant operations and memory reads and writes. EM-Explorer *emulates* such a trace based on their operational semantics and explores all interleavings of the given execution trace permitted by the semantics. Android permits user and system-generated events apart from program-generated events. EM-Explorer only explores the non-determinism between program and system generated events while keeping the order of user events fixed. This is analogous to model checking *w.r.t.* a fixed data input. EM-Explorer does not track variable values and is incapable of evaluating conditionals on a different interleaving of the trace.

Android supports different types of component classes, e.g., Activity class for the user interface, and enforces a happens-before ordering between handlers of lifecycle events of component classes. EM-Explorer seeds the happens-before relation for such events in each trace before starting the model checking run, avoiding exploration of invalid interleavings of lifecycle events. We remove handlers with no visible operations from recorded traces before model checking.

4 Experimental Evaluation

We compare the performance of (1) EM-DPOR which computes dependence-covering sets with (2) DPOR which computes persistent sets. Both the algorithms are implemented in the EM-Explorer framework described in Section 3 and use vector clocks. The implementation of DPOR uses the dependence relation in which `post`s to the same thread are considered dependent.

We evaluated these POR techniques on execution traces generated by DroidRacer on 5 Android applications obtained from the Google Play Store [1]. Table 1 presents statistics like trace length (the number of *visible* operations), and the number of threads, events and (shared) memory locations in an execution trace for each of these applications. We only report the threads created by the application, and the number of events excluding events with no visible operations in their handlers.

We analyzed each of the traces described in Table 1 using both the POR techniques. Table 1 gives the number of interleavings (listed as *Traces*) and distinct transitions explored by DPOR and EM-DPOR. It also gives the time taken for exploring the reduced state space for each execution trace. If a model checking run did not terminate within 4 hours, we force-kill it and report the statistics for 4 hours. The statistics for force-killed runs are marked with $*$ in Table 1. Since EM-Explorer does not track variable values, it cannot prune executions that are infeasible due to conditional sequential execution. However, both DPOR and EM-DPOR are implemented on top of EM-Explorer and therefore operate on the same set of interleavings. The difference in their performance thus arises from the different POR strategies.

In our experiments, DPOR's model checking run terminated only on two execution traces among the five, whereas, EM-DPOR terminated on all of them. Except for one case, EM-DPOR finished state space exploration within a few seconds. As can be seen from Table 1, DPOR explores a much larger number of interleavings and transitions, often *orders of magnitude* larger compared to EM-DPOR. While this is a small evaluation, it does show that significant reduction can be achieved for real-world event-driven multi-threaded programs by avoiding unnecessary reordering of events.

Performance. Both the techniques used about the same memory and the maximum peak memory consumed by EM-DPOR across all traces, as reported by Valgrind, was less than 50MB. The experiments were performed on a machine with Intel Core i5 3.2GHz CPU with 4GB RAM, and running Ubuntu 12.04 OS.

5 Conclusions and Future Work

The event-driven multi-threaded style of programming concurrent applications is becoming increasingly popular. We considered the problem of POR-based efficient stateless model checking for this concurrency model. The key insight of our work is that more reduction is achievable by treating operations that post events to the same thread as independent and only reordering them if necessary.

We presented POR based on dependence-covering sequences and sets. Exploring only dependence-covering sets suffices to provide certain formal guarantees. Our experiments provide empirical evidence that our dynamic algorithm for computing dependence-covering sets explores orders of magnitude fewer transitions compared to DPOR for event-driven multi-threaded programs. While we evaluate our algorithm on Android applications, the general idea of dependence-covering sets is more widely applicable. As future work, we aim to achieve better reductions by defining a notion of sleep sets suitable for this concurrency model and combining it with dependence-covering sets as well as explore optimality of POR in this context.

Acknowledgements. P. Maiya thanks Google India for its support through a PhD Fellowship in Programming Languages and Compilers. This research was also funded in part by the Indo-German Max Planck Center in Computer Science (IMPECS).

References

1. Google Play. https://play.google.com/store/apps
2. Grand Central Dispatch. https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref
3. Java AWT EventQueue. http://docs.oracle.com/javase/7/docs/api/java/awt/EventQueue.html
4. TinyOS. http://www.tinyos.net
5. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal Dynamic Partial Order Reduction. In: POPL. pp. 373–384. ACM (2014)
6. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.F.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015)
7. Bielik, P., Raychev, V., Vechev, M.T.: Scalable race detection for android applications. In: OOPSLA. pp. 332–348. ACM (2015)
8. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. STTT 2(3), 279–287 (1999)
9. Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: SIGSOFT FSE. pp. 48:1–48:11. ACM (2012)
10. Flanagan, C., Godefroid, P.: Dynamic Partial-Order Reduction for Model Checking Software. In: POPL. pp. 110–121. ACM (2005)
11. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer, Heidelberg (1996)
12. Godefroid, P.: Model Checking for Programming Languages using Verisoft. In: POPL. pp. 174–186. ACM Press (1997)
13. Godefroid, P.: Software model checking: The verisoft approach. Formal Methods in System Design 26(2), 77–101 (2005)
14. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
15. Hsiao, C.H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C.L., Pokam, G.A., Chen, P.M., Flinn, J.: Race Detection for Event-driven Mobile Applications. In: PLDI. pp. 326–336. ACM (2014)
16. Jensen, C.S., Møller, A., Raychev, V., Dimitrov, D., Vechev, M.T.: Stateless model checking of event-driven applications. In: OOPSLA. pp. 57–73. ACM (2015)
17. Lauterburg, S., Dotta, M., Marinov, D., Agha, G.A.: A framework for state-space exploration of java-based actor programs. In: ASE. pp. 468–479. IEEE Computer Society (2009)
18. Maiya, P., Gupta, R., Kanade, A., Majumdar, R.: A partial order reduction technique for event-driven multi-threaded programs. CoRR abs/1511.03213 (2015)
19. Maiya, P., Kanade, A., Majumdar, R.: Race detection for Android applications. In: PLDI. pp. 316–325. ACM (2014)
20. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms. pp. 215–226. Elsevier (1989)
21. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1986)
22. Mednieks, Z., Dornin, L., Meike, G.B., Nakamura, M.: Programming Android. O'Reilly Media, Inc. (2012)
23. van der Merwe, H.: Verification of Android Applications. In: ICSE. vol. 2, pp. 931–934. IEEE (2015)
24. van der Merwe, H., van der Merwe, B., Visser, W.: Verifying android applications using Java PathFinder. ACM SIGSOFT Software Engineering Notes 37(6), 1–5 (2012)

25. Ozkan, B.K., Emmi, M., Tasiran, S.: Systematic asynchrony bug exploration for android apps. In: Kroening, D., Pasareanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 455–461. Springer, Heidelberg (2015)
26. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs. In: PADTAD. pp. 43–53. ACM (2007)
27. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer (1993)
28. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: CONCUR. LIPIcs, vol. 42, pp. 456–469. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
29. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer (2006)
30. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
31. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE 2012. LNCS, vol. 7273, pp. 219–234. Springer, Heidelberg (2012)
32. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) Advances in Petri Nets 1990. LNCS, vol. 483, pp. 491–515. Springer (1989)
33. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI. pp. 250–259. ACM (2015)