Comparing Source Sets and Persistent Sets for Partial Order Reduction *

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas

Department of Information Technology, Uppsala University, Uppsala, Sweden
{parosh,stavros.aronis,bengt,kostis}@it.uu.se

Abstract. Partial order reduction has traditionally been based on persistent sets, ample sets, stubborn sets, or variants thereof. Recently, we have presented a strengthening of this foundation, using source sets instead of persistent/ample/stubborn sets. Source sets subsume persistent sets and are often smaller than they are. We introduced source sets as a basis for Dynamic Partial Order Reduction (DPOR), in a framework which assumes that processes are deterministic and that all program executions are finite. We also show how to use source sets for partial order reduction in a framework which does not impose these restrictions. Finally, we compare source sets with persistent sets, providing some insights into conditions under which source sets and persistent sets do or do not differ.

1 Introduction

Verification and systematic testing of concurrent programs are difficult, since they must consider all the different ways in which processes/threads can interact. *Model checking* [19, 6] addresses this problem by systematically exploring the state space of a given program and verifying that each reachable state satisfies a given property. A serious hindrance to the applicability of model checking is the *state-space explosion* problem, i.e., that the number of possible interleavings grows exponentially with the length of program execution. There are several approaches that limit the number of explored interleavings. Among them, *Partial Order Reduction* (POR) [27, 18, 9, 7] stands out, as it provides full coverage of all behaviours that can occur in *any* interleaving, even though it explores only a representative subset. POR is based on the observation that two interleavings can be regarded as equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (independent) execution steps. POR exploits this observation by guaranteeing that for each possible interleaving, it explores one that is equivalent to it. This is sufficient for checking many interesting safety properties, including race freedom, absence of global deadlocks, and absence of assertion violations [27, 9, 7].

* This work was carried out within the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center), partly supported by the Swedish Research Council.

Partial order reduction approaches are based on reducing the set of process steps that are explored at each scheduling point. This set of process steps has been given different names, including *stubborn sets* [27], *persistent sets* [9], and *ample sets* [7]. These approaches are rather similar: their differences are mainly due to the considered model of computation and the class of properties to be checked. In the following, we will consider the approach based on persistent sets. Recently, we have proposed an improvement over the persistent set technique, which is based on a new class of sets called *source sets* [2]. Source sets subsume persistent sets (i.e., any persistent set is also a source set), and source sets are often smaller than persistent sets. Moreover, source sets are provably minimal, in the sense that the set of explored processes from some scheduling point must be a source set in order to guarantee exploration of all equivalence classes. This implies that techniques based on source sets have the potential to produce better reduction than techniques based on persistent sets.

In our previous work [2], we introduced source sets in the context of stateless model checking, as a basis for *Dynamic Partial Order Reduction* (DPOR). We used a framework, which assumes as restrictions on analyzed programs that processes (or threads) are deterministic and that all program executions are finite. We demonstrated the power of source sets by using them as the basis for two DPOR algorithms:(i) *Source-DPOR*, which minimally adapts the original DPOR algorithm due to Flanagan and Godefroid [8] to use source sets instead of persistent sets, leading to notably better reduction, and (ii) *Optimal-DPOR*, which is provably optimal in the sense that it explores exactly one representative execution in each equivalence class.

In this paper, we consider how to use source sets for partial order reduction in a framework which does not impose the restrictions of deterministic processes and terminating computations. We therefore consider a framework of arbitrary finite-state concurrent programs, in which processes can be non-terminating and exhibit control non-determinism. This is essentially the class of programs that is considered in classical works [27, 18, 9, 7] on POR that employ ample/persistent/stubborn sets, and is used, e.g., in the Promela language of the widely used SPIN model checker [11]. For this model, we show in this paper how source sets can be used as a basis for POR in enumerative state-space exploration. We also show how the theory of POR can equally well be based on source sets, and that the underlying theory is at least as simple as when using persistent sets.

We first consider the basic principles for partial order reductions, as they have been formulated for non-deterministic finite-state concurrent programs in standard texts; for example, in the survey by Clarke *et al.* [7] or the textbook by Baier and Katoen [3, Chapter 8]. For simplicity, we limit the exposition to the detection of local assertion violations and deadlocks. We show that the use of ample/persistent/stubborn sets can be replaced by source sets.

We thereafter consider the framework of stateless model checking for terminating programs, but allowing control non-determinism, thereby generalizing the standardly used restriction to deterministic processes [8, 2]. We present a generalization of the Source-DPOR algorithm to this setting.

Finally, we make a direct comparison between source sets and persistent sets, with the purpose to characterize under which conditions source sets provide a strict advantage over persistent sets. We provide examples of situations in which sources sets are guaranteed to be strictly smaller than corresponding persistent sets, and also situations in which this is not the case.

To keep the presentation simple, we do not include complementary techniques, most notably sleep sets [10]. Sleep sets are complementary to persistent sets and source sets; especially in stateless model checking they are essential for reduction. Including sleep sets, as we did in our prior paper [2], makes the algorithms, and particularly their correctness proofs, more complex. For simplicity, we have omitted them from this paper, but the observed differences between source sets and persistent sets remain essentially the same also in the presence of sleep sets.

Related Work Early persistent set techniques [27, 9, 7] relied on static analysis to compute persistent sets. Sleep set techniques [10] were also used to dynamically prevent explorations that would be redundant. The DPOR algorithm of Flanagan and Godefroid [8] showed how to construct persistent sets on-the-fly "by need", leading to better reduction. Similar techniques have been combined with dynamic symbolic execution, which is also known as *concolic testing*, where new test runs are initiated in response to detected races by flipping these races using postponed sets [24]. Since then, several variants, improvements, and adaptations of DPOR for stateless model checking [14, 26] and concolic testing [23, 21] have appeared, all based on persistent sets. In 2014, we introduced source sets, which are provably more succinct than persistent sets, as a basis for two DPOR algorithms [2]. We subsequently applied one of them, namely the Source-DPOR algorithm, to programs executing under the TSO and PSO memory model [1].

Other techniques for reducing state-space explosion in model checking include, unfoldings [16], which can in principle achieve better reduction than POR in number of interleavings. However, such techniques [13, 20] have larger cost per explored execution than techniques based on POR, and one of them [13] also needs an additional post-processing step for checking non-local properties such as races and deadlocks. Another line of work exploits a weaker form of equivalence, the *maximal causal model* for a concurrent computation from a given execution trace, a notion defined by Serbanuta *et al.* [25]. This was used recently [12] for a stateless model checking algorithm, which also explores fewer traces than classical DPOR techniques. The corresponding algorithm, called Maximal Causality Reduction, relies on an offline constraint analyzer to formulate constraints that are then solved using an SMT solver.

For the analyses of timed systems, such as those performed by the UPPAAL model checker [5], partial order reduction techniques have been proposed [4, 22], but they have not achieved reductions that are comparable in magnitude to those observed for untimed concurrent programs.

Organization In the next section, we introduce our computational model. In Sect. 3 we formulate a partial-order view of program executions, the partial-order framework, along with the definition of source sets. In Sect. 4 we show that

source sets can be used as a basis for POR in state-space exploration. In Sect. 5 we consider how to actually construct source sets, and consider two settings: static computation of source sets, as performed in model checkers such as SPIN, and dynamic computation, as performed in DPOR. We present algorithms for constructing source sets in both these settings. In Sect. 6 we recall the definition of persistent sets, and show how, under various conditions, source sets are strictly smaller than persistent sets, but that the strictness may disappear in some settings. Finally, in Sect. 7 we summarize the findings of the paper.

2 Framework

Let us introduce the technical background material. First, we present our model of concurrent programs, and thereafter the concepts of independence, races, and the happens-before relation. To keep the exposition simple, we assume a simple model of concurrent programs as composed of a finite set of processes, each of which is finite-state, can be non-terminating, and exhibit control non-determinism. Processes communicate by reading from and writing to a set of shared variables. Locks can be seen as shared variables that are manipulated in a certain way.

2.1 Computation Model

We model a concurrent program as a transition system $TS = \langle \mathcal{P}, \mathcal{X}, S, s_0, T \rangle$, where $\mathcal{P}$ is a finite set of *processes* (or *threads*), $\mathcal{X}$ is a finite set of shared variables, S is a finite set of *global states*, each of which consists of a valuation of the shared variables and a local state of each process, $s_0 \in S$ is the *initial global state*, and T is a finite set of transitions. Each transition t belongs to a unique process, denoted $\widehat{t}$. We assume that each transition can be represented as a guarded command over the shared variables and the local state of its processes. If the guard of transition t evaluates to true in s, we say that t is *enabled* in s; we denote this by $s \vdash t$ and write $t(s)$ for the unique state that results after t is executed from state s. We let *enabled*(s) be the set of transitions that are enabled in s. A process is said to be *blocked* in some state s if none of its transitions is enabled in s. Often, the local state of a process will include a control location, which is updated by transitions in the usual way.

For a sequence $w = t_1 \dots t_k$ of transitions, we write $s \vdash w$ to denote that the sequence w can be executed from s, i.e., that there are states $s_1 \dots s_k$ such that with $s_0 = s$ we have $s_{i-1} \vdash t_i$ and $s_i = t_i(s_{i-1})$ for $i = 1, \dots, k$, and write $w(s)$ for s_k. If $s \vdash w$ we say that w is an *execution sequence from* s. An *execution sequence* of TS is an execution sequence from its initial state. We will use w and v with sub- and superscripts for execution sequences from some state, and E with sub- and superscripts for execution sequences from the initial state. Since transitions are deterministic, an execution sequence w from s takes the program to a unique state $w(s)$. When s is the initial state, we will use $s_{[E]}$ to denote $E(s_0)$. We sometimes use execution sequences instead of states in concepts and notations; e.g., we sometimes write $E \vdash w$ to denote $s_{[E]} \vdash w$. For a sequence of

transitions w, we let $w \setminus t$ denote w with the first occurrence of t removed; if $t \notin w$, then $w \setminus t$ is w.

For a set of sequences of transitions W, we write $s \vdash W$ to denote that $s \vdash w$ for each $w \in W$. We write $\mathcal{E}(s)$ for the set of execution sequences w with $s \vdash w$.

2.2 Partial Order Representation

The basic idea in partial order reduction is to consider executions as partial orders on transitions. Executions that are represented by the same partial order are considered equivalent. We therefore formalize how to view executions as partially ordered sets of events.

Let E be an execution sequence. An *event* of E is a particular occurrence of a transition in E. More precisely, an event is a pair $\langle t, i \rangle$, representing the ith occurrence of transition t in the execution sequence. We use $e, e', \ldots$ to range over events. We let $[e]$ denote the transition of e, and $\widehat{e}$ denote the process of $[e]$. We let $dom(E)$ denote the set of events $\langle t, i \rangle$ which are in E (i.e., $\langle t, i \rangle \in dom(E)$ iff E contains at least i occurrences of t). We use $<_E$ to denote the total order between events in E, i.e., $e <_E e'$ denotes that e occurs before e' in E. We use $next_{[E]}(t)$ to denote the event that transition t represents when executed immediately after E.

The partial order view of an execution sequence is obtained by defining a *happens-before relation* on its events. Intuitively, the happens-before relation captures the causal ordering between events in an execution sequence. More precisely, it captures the orderings that are important for the result of the execution sequence. For each execution sequence E, the happens-before relation $\rightarrow_E$ is defined on $dom(E)$ as the transitive closure of the relation $\rightsquigarrow_E$ defined by letting $e \rightsquigarrow_E e'$ if $e <_E e'$ (i.e., e occurs before e'in E) and either

(i) e and e' are performed by the same process, or
(ii) some shared variable is accessed by both e and e', and at least one of e or e' performs a write access.

Note that two events that are different occurrences of the same transition can access different sets of variables, e.g., if the accesses are conditional on some test. It follows that $\rightarrow_E$ is a partial order on $dom(E)$. Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence that has exactly the same events as E (i.e., $dom(E') = dom(E)$) and the same happens-before relation $\rightarrow_{E'}$ as $\rightarrow_E$. This means that the relation $\rightarrow_E$ induces a set of equivalent execution sequences, all with the same happens-before relation. We will sometimes refer to such equivalence classes as *Mazurkiewicz traces* [15]. 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. If $E \simeq E'$, then all variables are modified by the same sequence of statements, implying that $s_{[E]} = s_{[E']}$.

Example 1. In Fig. 1, the three processes p, q, and r perform dependent accesses to a shared variable `x`. In this example, let us consider two accesses as dependent if they concern the same variable and one of them is a write. Since there are no

writes to `y` and `z` here, accesses to `y` and `z` are not dependent with anything else. For this program, there are four Mazurkiewicz traces (i.e., equivalence classes of executions), each characterized by its sequence of accesses to `x` (three accesses can be ordered in six ways, but two pairs of orderings are equivalent since they differ only in the ordering of adjacent reads, which are not dependent). An execution sequence and its corresponding happens-before relation is shown in Fig. 2.

Partial order reduction reduces the effort for analyzing a concurrent system by analyzing only a representative subset of all execution sequences. The idea is that for each execution sequence E of TS, it is sufficient to analyze a sequence E' which is equivalent to E. In fact, it is sufficient to analyze a sequence E' which is equivalent to some extension of E, i.e., such that $E' \simeq E.v$ for some v. Any local assertion violation or global deadlock in E is then also visible in E'. This relation between E' and E is important, so we will introduce some additional notation that will make the subsequent exposition more convenient.

Definition 1. *Let E and E' be execution sequences.*

- *Let $E \sqsubseteq E'$ denote that there is a sequence w such that $E.w$ is an execution sequence with $E.w \simeq E'$.*
- *Let $E \sim E'$ denote that there are sequences w and w' such that $E.w$ and $E'.w'$ are execution sequences with $E.w \simeq E'.w'$.*

Intuitively, $E \sqsubseteq E'$ denotes that the sequence E is a possible way to start an execution that is equivalent to E'; thus E can be thought of a "partial order prefix" of E', in the sense that the events in E are a downward-closed subset of the events in E', with the same happens-before relation as in E'. Analogously, $E \sim E'$ denotes that the sequence E is a possible way to start an execution that is equivalent to an execution sequence of the form $E'.w'$; thus E and E' are "consistent" in the sense that they can be extended to produce two equivalent sequences.

We also introduce relativized versions of these definitions.

Definition 2. *Let E be an execution sequence. Then*

- *$v \sqsubseteq_{[E]} w$ denotes that $E.v \sqsubseteq E.w$,*
- *$v \sim_{[E]} w$ denotes that $E.v \sim E.w$.*
- *$v \simeq_{[E]} w$ denotes that $E.v \simeq E.w$.*

We will sometimes use $\sqsubseteq_{[s]}$ for $\sqsubseteq_{[E]}$, where E is such that s is $s_{[E]}$, and analogously for $\sim_{[s]}$ and $\simeq_{[s]}$ (note that $\sqsubseteq_{[E]}$ and $\sim_{[E]}$ and $\simeq_{[E]}$ are uniquely determined by $s_{[E]}$). An important property is that

p:	*q*:	*r*:
`write x;`	`read y;`	`read z;`
	`read x;`	`read x;`

Fig. 1. Writer-readers code excerpt.

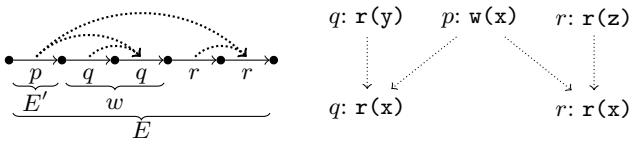

Fig. 2. A sample execution sequence of the program in Fig. 1 is shown to the left. This execution sequence is annotated by a happens-before relation (the dotted arrows). To the right, the happens-before relation is shown as a partial order.

– $v \sqsubseteq_{[E]} w$ and $w \sim_{[E]} w'$ implies $v \sim_{[E]} w'$.

Note that $v \sqsubseteq_{[E]} w$ and $w \sim_{[E]} w'$ does not in general imply $v \sqsubseteq_{[E]} w'$. As a simple counterexample, let v and w be p and let w' be the empty sequence $\langle\rangle$, and observe that $v' \sim_{[E]} \langle\rangle$ for any sequence v'. From this observation, it also follows that $\sim_{[E]}$ is not transitive.

Finally, we introduce special notation and terminology for the case that w consists of a single transition:

Definition 3 (Initials and Weak Initials). *For an execution sequence w from s, the set $I_{[s]}(w)$ of* initials *and the set $WI_{[s]}(w)$ of* weak initials *are sets of transitions defined as follows:*

– $t \in I_{[s]}(w)$ *iff* $t \sqsubseteq_{[s]} w$, *and*
– $t \in WI_{[s]}(w)$ *iff* $t \sim_{[s]} w$.

We let $s \vdash t \diamondsuit w$ denote that $s \vdash t.w$ and $next_{[E]}(t) \not\rightarrow_{E.t.w} e$ for each $e \in dom_{[E.t]}(w)$ (i.e., the event of t does not "happen before" any event in w), where E is such that s is $s_{[E]}$. Intuitively, $s \vdash t \diamondsuit w$ means that t and w are independent after the state s. If $s \vdash t \diamondsuit w$ and s is $s_{[E]}$, then $E.t.w \simeq E.w.t$, which implies $s_{[E.t.w]} = s_{[E.w.t]}$, i.e., t and w can be swapped without changing the resulting state. In the special case when w contains only one transition t', then $s \vdash t \diamondsuit t'$ denotes that the transitions t and t' are independent after s. We use $s \nvdash t \diamondsuit w$ to denote that $s \vdash t$ and $s \vdash w$, but that $s \vdash t \diamondsuit w$ does not hold.

As examples, in Fig. 2, we have $q.r \sqsubseteq_{[E']} q.q.r.r$ but $q.q \not\sqsubseteq_{[E']} r.r$. We also have $q.q \sim_{[E']} r.r$ since $E'.q.q.r.r \simeq E'.r.r.q.q$. In Fig. 2, we further have $E' \vdash q \diamondsuit r$ since q and r are not happens-before related in $E'.r.q$. We also observe that $I_{[E']}(w) = \{q\}$, as q is the only process occurring in w and its first occurrence has no predecessor in the dotted relation in w. Furthermore, $WI_{[E']}(w) = \{q, r\}$, since r is not happens-before related to any event in w.

3 Principles of Partial Order Reduction

The purpose of partial order reduction is to reduce the state-space explosion that arises from the many possible interleavings of transitions of concurrent processes. Instead of generating and analyzing all executions of a concurrent system, POR generates and analyzes only a representative subset.

Following other works [9, 7], we can consider partial order reduction as constructing a *reduced state graph*. A transition system $\langle \mathcal{P}, \mathcal{X}, S, s_0, T \rangle$ induces a labeled state-transition graph A_G, whose nodes are the states in S, with labeled edges of the form $s \xrightarrow{t} s'$ whenever $t(s) = s'$. Each execution sequence of the concurrent system corresponds to a path from the initial node of A_G. One can explain POR as constructing a *reduced* state-transition graph A_R, whose nodes are a subset S_R of S, and whose edges are a subset of the edges of A_G. The reduced graph A_R must still cover all behaviors of A_G in the sense that for any execution sequence E of A_G, there is an execution sequence E' in A_R with $E \sqsubseteq E'$. Note that the above principle of constructing a reduced state-transition graph represents the situation both in stateful model checking, with an arbitrary A_G, as well as in stateless model checking, where A_G is tree-shaped.

In enumerative state-space exploration, A_G is constructed by a recursive exploration of all enabled transitions from global states already found to be reachable, starting from the initial global state. When applying partial-order reduction, only a subset of the enabled transitions are explored from any global state, thereby constructing a reduced state-transition graph A_R. Let us consider what are the restrictions on this subset. Consider a global state s. The above requirement that A_R must cover all behaviors of A_G can be satisfied by requiring that for any execution sequence w from s in A_G, there is an execution sequence w' from s in A_R with $w \sqsubseteq_{[s]} w'$. Let now w' be of the form $t.w''$, i.e., let t be the first transition in w'. From $w \sqsubseteq_{[s]} t.w''$ it follows that $w \sim_{[s]} t$, i.e., $t \in WI_{[s]}(w)$. Since w is an arbitrary execution sequence from s, we conclude that the set of transitions explored from s must include some transition in $WI_{[s]}(w)$ for each execution sequence from w. Thus, in the reduced state graph, it is is *necessary* to explore at least one transition in $WI_{[s]}(w)$ for each execution w from s. We therefore give a name to such sets.

Definition 4 (Source Sets). *Let s be a state, and let W be a set of execution sequences from s. A set T of transitions is* a source set for W after s *if for each $w \in W$ we have $WI_{[s]}(w) \cap T \neq \emptyset$.*

We say that T is a source set after s to denote that T is a source set for $\mathcal{E}(s)$. For an execution sequence E, we say that T is a source set (for W) after E if T is a source set (for W) after $s_{[E]}$.

The key property is that if T is a source set after s, then for each execution sequence $w \in W$, there is a transition $t \in T$ and an execution sequence w' such that $E.t.w' \simeq E.w.v$ for some sequence v. In particular, if $E.w$ is maximal, then $E.t.w' \simeq E.v$. Therefore, when an exploration algorithm intends to cover all of $\mathcal{E}(s)$, the set of transitions that are chosen for exploration from s *must* be a source set after s. We formulate this observation as a theorem.

Theorem 1 (Key Property of Source Sets). *Let s be a state, and let W' be a subset of $\mathcal{E}(s)$ such that for each $w \in \mathcal{E}(s)$ there is a $w' \in W'$ with $w \sqsubseteq_{[s]} w'$. Then the set of first transitions of sequences in W' is a source set after E.*

This theorem implies that a necessary condition for the correctness of POR is that the set of explored transitions is a source set. As we will show in the next

section, this condition is actually sufficient if A_G is acyclic. If A_G may contain cycles, one more condition will be needed.

4 Partial Order Reduction in State-Space Exploration

In this section, we show how source sets can be used for partial order reduction in enumerative model checking. This section has borrowed inspiration from the presentation by Clarke, Grumberg, Minea, and Peled [7, Sect. 4], that shows how to use ample sets (which for the purposes of this paper are essentially the same as persistent sets) in POR. We show that source sets can be used instead of ample sets or persistent sets to obtain reduction. Since, as we show in Section 6, source sets are at least as powerful as persistent sets of ample sets, the obtained reduction will be at least as good and sometimes better.

Let us consider the conditions for producing a reduced state transition graph A_R by restricting the set of enabled transitions that are explored from each state. We have the following main theorem.

Theorem 2. *Assume a concurrent system with global state graph A_G. Let A_R be a reduced state transition graph obtained from A_G by restricting the set of transitions that are explored from each state. If the following two conditions are satisfied:*

1. *for each state s in A_R, the set of explored transitions is a source set after s, and*
2. *for each cycle in A_R, if a transition t is enabled in all states of the cycle, then t must be explored from some state of the cycle,*

then for each execution E in A_G, there is an execution E' in A_R with $E \sqsubseteq E'$.

Proof. We will prove the following stronger property:

> *for each state $s \in A_R$ and execution sequence w from s in A_G, there is an execution w' in A_R with $w \sqsubseteq_{[s]} w'$*

by induction on the length of w. The base case is trivial. For the inductive step, by condition (1), A_R explores some transition t in $WI_{[s]}(w)$ from s. We have two cases.

- $t \in w$. From $t \in WI_{[s]}(w)$ we infer $t.(w \setminus t) \simeq w$. By the induction hypothesis applied to the state $t(s)$ and execution sequence $w \setminus t$ from $t(s)$, the reduced state graph A_R contains a sequence w'' with $(w \setminus t) \sqsubseteq_{[t(s)]} w''$, implying that A_R contains the sequence $t.w''$ from s. We furthermore have that $t.(w \setminus t) \sqsubseteq_{[s]} t.w''$, i.e., since $t.(w \setminus t) \simeq_{[s]} w$ we have $w \sqsubseteq_{[s]} t.w''$, meaning that we can take $t.w''$ as w'.
- $t \notin w$. Let us use t_1 to denote t. Then $t_1 \in WI_{[s]}(w)$ and $t_1 \notin w$ imply $s \vdash t_1 \diamondsuit w$, which implies that w is an execution sequence also from $t_1(s)$. Therefore, again by condition (1), A_R explores some transition t_2 in $WI_{[t(s)]}(w)$ from $t(s)$. Continuing in this way, we have two cases:

(i) there is a sequence $t_1 t_2 \dots t_k$ such that for $i = 1, \dots, k$, the sequence w is an execution from $(t_1 \dots t_{i-1})(s)$ and $(t_1 \dots t_{i-1})(s) \vdash t_i \diamondsuit w$ but $t_k \in I_{[(t_1 \dots t_{k-1})(s)]}(w)$. By extending the reasoning from the first case, we have that A_R contains a sequence $t_1 \dots t_k . w''$ from s such that $t_k . (w \setminus t_k) \sqsubseteq_{[s]} t_1 \dots t_{k-1} w''$. Since $t_k . (w \setminus t_k) \simeq_{[s]} w$, we have $w \sqsubseteq_{[s]} t_1 t_2 \dots t_{k-1} w''$, meaning that we can take $t_1 t_2 \dots t_{k-1} w''$ as w'.

(ii) there is an unbounded sequence $t_1 t_2 \dots$ such that for $i = 1, 2, \dots$, the sequence w is an execution from $t_1 \dots t_{i-1}(s)$ and $t_1 t_2 \dots t_{i-1}(s) \vdash t_i \diamondsuit w$. It follows that the sequence of states $t_1 \dots t_{i-1}(s)$ must form a loop somewhere, and furthermore that the first transition of w is enabled in all states of that loop. By condition (2), the first transition must then be executed from some state of the loop, and we are back to the previous case. □

5 Computing Source Sets

In the previous section, we showed how source sets can be used for partial order reduction in model checking. In this section, we consider the problem of how to actually compute source sets, given some transition system model. The definition of source sets can not be used directly for this purpose, since it is formulated in terms of all possible sequences that should be analyzed, and we do not want to explore all of them in the first place. For the case of persistent sets, a variety of techniques for computing them have been proposed. One broad classification of techniques is into *static* techniques, which compute source sets by analyzing the program source, and *dynamic* techniques, which compute persistent sets incrementally by analyzing already performed exploration. We will present algorithms for computing source sets in both these settings. We first present a technique for static computation of source sets, and thereafter a dynamic one.

5.1 Static Computation of Source Sets

Early approaches to partial order reduction [27, 18, 9] computed persistent sets statically, by analyzing the program text. Various static algorithms were proposed for computing persistent sets, ample sets, or stubborn sets; these algorithms are not so different from each other, and are related to an earlier algorithm by Overman [17]. Below, we present an analogous technique for static computation of source sets.

We say that a transition t is *may-enabled* in a state s if it is enabled in s or can become enabled after a sequence of transitions from processes other than $\widehat{t}$. An *access* is defined to be a read or a write to some shared variable. Two accesses *conflict* if they access the same shared variable and at least one of them is a write.

For the computation of source sets, we assume that for each state we can compute an over-approximation of the set of *may-enabled* transitions in that state. Define a *may-set* in a state s to be a set of transitions which contains exactly

one may-enabled transition of each process that has a may-enabled transition in s; for this definition, we use the computed over-approximation of the set of may-enabled transitions. We further assume that we can compute the following, possibly overapproximating, sets. This can be done, e.g., by static analysis of the control flow of the code of each process.

- $\mathit{init\_accesses}_{[s]}(t)$, for an enabled transition t, contains the set of accesses t can perform when t is executed as the first transition of process $\widehat{t}$ from s.
- $\mathit{future\_accesses}_{[s]}(t)$, for a may-enabled transition t, contains the set of accesses that can be performed by $\widehat{t}$ in any execution sequence from s, in which t is the first transition of $\widehat{t}$.

We expect that $\mathit{init\_accesses}_{[s]}(t) \subseteq \mathit{future\_accesses}_{[s]}(t)$.

The following theorem provides a sufficient characterization of source sets.

Theorem 3. *Let s be a global state, A set $T \subseteq \mathit{enabled}(s)$ of transitions is a source set after s if each may-set T' in s contains a non-empty subset T'' such that*

- *for each $t'' \in T''$ and each $t' \in (T' \setminus T'')$, no access in $\mathit{init\_accesses}_{[s]}(t'')$ conflicts with any access in $\mathit{future\_accesses}_{[s]}(t')$.*

Proof. We use the definition of source sets (Definition 4). Consider a sequence $w \in \mathcal{E}(s)$. Define a may-set T' in s which for each process p that has a transition in w contains the transition of p that occurs first; if p has no transition in w then T' can contain any may-enabled transition of p. By the condition in the theorem, the set T' has a subset $T'' \subseteq T$ such that for each $t'' \in T''$ and each $t' \in (T' \setminus T'')$, no access in $\mathit{init\_accesses}_{[s]}(t'')$ conflicts with any access in $\mathit{future\_accesses}_{[s]}(t')$. We split the proof into two cases.

1. w contains no transition in T''. We then claim that $s \vdash t'' \diamondsuit w$ for each $t'' \in T''$. This follows by establishing that no access by t'', when performed in s, conflicts with any access by any transition in w. To see this, consider an arbitrary transition t''' in w. Let t' be the first transition of process $\widehat{t'''}$ in w. The condition in the theorem then implies that no access in $\mathit{init\_accesses}_{[s]}(t'')$ conflicts with any access in $\mathit{future\_accesses}_{[s]}(t')$, in particular not with any access by t'''.
2. w contains a transition in T''. Then let t'' be the first of these. Then w has a prefix of the form $v.t''$, and by using the condition in the theorem in the same way as in the previous case, we infer that $s \vdash t'' \diamondsuit v$, which implies that $t'' \in I_{[s]}(w)$, i.e., $t'' \in WI_{[s]}(w)$. □

Let us illustrate a use of Theorem 3, and also contrast it with persistent sets. Consider the simple program with two processes in Fig. 3. Here, the set $\{t_2, t_3\}$ is a source set, which satisfies the conditions in Theorem 3. If we choose T' as $\{t_1, t_3\}$ in the theorem, then we can choose T'' as $\{t_3\}$. Note that $\{t_2, t_3\}$ is *not* a persistent set: any persistent set must include all transitions.

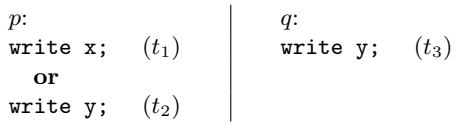

Fig. 3. Non-deterministic program.

5.2 Dynamic Computation of Source Sets

Let us next consider how source sets can be computed dynamically, i.e., by actually exploring execution sequences from a state. The motivation for the dynamic approach is that static analysis often over-approximates possible conflicts between transitions, thereby limiting the achievable reduction. Dynamic approaches improve the precision by recording actually occurring conflicts during the exploration and using this information to construct source sets on-the-fly, "by need". The dynamic approach has been particularly successful in stateless model checking, under the name *Dynamic Partial Order Reduction* (DPOR), first presented by Flanagan and Godefroid [8]. DPOR requires that the state space is acyclic and finite. This means that executions must terminate by themselves within a bounded number of steps. Then the state space is a finite tree, built from execution sequences E. Each execution sequence E leads to a unique $s_{[E]}$. From the point of view of the exploration, two different execution sequences E and E' are considered to be different states, even if $s_{[E]} = s_{[E']}$.

Let us consider how to compute source sets dynamically. A naïve approach may consume a significant effort in exploring sequences from $s_{[E]}$ only for the purpose of computing source sets, exploration which is otherwise wasted. A key idea of DPOR is to compute source sets, not by separate exploration, but by analyzing the execution sequences that are anyway explored for analyzing the transition system. More precisely, if E is an explored execution sequence, then the set of transitions which will be explored from $s_{[E]}$ is constructed incrementally by the DPOR algorithm. Following [8], we use $\mathit{backtrack}(E)$ to denote this set of transitions. When initiating the exploration from $s_{[E]}$, the set $\mathit{backtrack}(E)$ is initialized with the enabled transitions of an arbitrary process. During the exploration from $s_{[E]}$, the algorithm explores sequences of the form $t.w$ from $s_{[E]}$, for each $t \in \mathit{backtrack}(E)$. These sequences are analyzed and, if needed, transitions are added to $\mathit{backtrack}(E)$, which will then induce exploration of additional sequences, etc. The key idea is that whenever a sequence of form $t.w'.t'$ is explored from $s_{[E]}$, in which the events corresponding to t and t' are in a race, then the DPOR algorithm must ensure that $\mathit{backtrack}(E)$ contains a transition which begins some execution sequence in which this race is reversed. At the end of the exploration, the set $\mathit{backtrack}(E)$ should be a source set after $s_{[E]}$.

Let us now make this idea more precise in our context. We first introduce some notation. For an execution sequence E and an event $e \in \mathit{dom}(E)$, let:

- $\mathit{pre}(E, e)$ denote the prefix of E up to, but not including, the event e.

– $notdep(e, E)$ denote the sub-sequence of E consisting of the events that occur after e but do not "happen after" e (i.e., the events e' that occur after e such that $e \not\rightarrow_E e'$).

Let E be an execution sequence, and let e and e' be two events in $dom(E)$, with $e <_E e'$. We say that e and e' are in a *reversible race*, denoted $e \precsim_E e'$, if

– $\widehat{e} \neq \widehat{e'}$ and $e \rightarrow_E e'$ and there is no event $e'' \in dom(E)$, different from e' and e, such that $e \rightarrow_E e'' \rightarrow_E e'$, i.e., e and e' are in a race, and
– $pre(E, e) \vdash notdep(e, E).[e']$, i.e., e' can be executed even if e is not, i.e., it is not the case that e' is enabled by e.

Intuitively, $e \precsim_E e'$ denotes that there is an equivalent execution sequence $E' \simeq E$ in which e and e' are adjacent, and that e does not enable e'. Therefore, the order of e and e' can be reversed.

Example 2. In Fig. 2, there are two pairs of events e and e' that are in a race, namely $\langle p, 1\rangle, \langle q, 2\rangle$ and $\langle p, 1\rangle, \langle r, 2\rangle$. It also holds for both these pairs that $e \precsim_E e'$ since both q and r are enabled before $\langle p, 1\rangle$. In other words, both the races in the program are reversible.

We will now derive sufficient conditions for adding transitions to $backtrack(E)$. For this derivation, we assume, as inductive hypothesis, that whenever $t \in backtrack(E)$, our algorithm explores all sequences in $\mathcal{E}(E.t)$ in the sense that for each $w \in \mathcal{E}(E.t)$ it explores a sequence w' with $w \sqsubseteq_{[E.t]} w'$. This inductive hypothesis can also be used to build a proof of correctness the resulting DPOR algorithm (see [2, 8]).

The requirement that $backtrack(E)$ be a source set after $s_{[E]}$ upon finishing the exploration implies that upon finishing the exploration from $s_{[E]}$, there should be no execution sequence w from $s_{[E]}$ with $WI_{[E]}(w) \cap backtrack(E) = \emptyset$. Assume, to derive a contradiction, that such a sequence w anyway exists. W.l.o.g. we can assume that w does not have any proper prefix with this property. It then follows from Definition 4 that w is of form $w'.t'$ such that $E \vdash t \diamondsuit w'$ for some $t \in backtrack(E)$. To see this, note that by the assumption that w is shortest there is a transition t in $WI_{[E]}(w') \cap backtrack(E)$, and since we cannot have $t \in I_{[E]}(w')$ (this would imply $t \in I_{[E]}(w)$), we must have $E \vdash t \diamondsuit w'$. Since $t \in backtrack(E)$, it could be hoped that the exploration will explore a sequence of the form $t.w'.t'$ from $s_{[E]}$. However, in spite of the fact that $E \vdash (w'.t')$, it is not certain that $E \vdash (t.w'.t')$, since t' may be disabled by t (note that $E.t.w' \simeq E.w'.t$ since $E \vdash t \diamondsuit w'$). We must therefore consider two cases.

1. $E \vdash (t.w'.t')$, i.e., t' is not disabled by t after $E.w'$. Let the events in $E.t.w'.t'$ corresponding to t and t' be e and e', respectively. Then $e \precsim_{E.t.w'.t'} e'$, i.e., e and e' are in a reversible race, since e depends with e' but not with any event in w'. In this case, since t is already in $backtrack(E)$, by the above inductive hypothesis, some sequence of the form w'' with $w'.t' \sqsubseteq_{[E.t]} w''$ is explored from $s_{[E.t]}$. Since the sequence $w'.t'$ is a partial-order prefix of w'', the race between e and e' will occur also in $E.t.w''$, i.e., w'' has a prefix

$w'''.t'$ where t' corresponds to e', such that $e \precsim_{E.t.w'''.t'} e'$. Now let u be $notdep(e, E.t.w''')$, i.e., u consists of the events of w''' that do not happen-after e (note that e corresponds to t in $E.t.w'''$). It follows that $u \sqsubseteq_{[E.t]} w'''$, hence that $u \sqsubseteq_{[E.t]} w''$, which together with $w'.t' \sqsubseteq_{[E.t]} w''$ implies $w' \sim_{[E.t]} u$. Noting that no events in w' nor in u happen-after e, we derive $w' \sim_{[E]} u$. Moreover, since w' and u contain the same events that happen-before e' after E, we infer $w'.t' \sim_{[E]} u.t'$. Let us now impose the requirement that $backtrack(E)$ must contain some transition t'' in $I_{[E]}(u.t')$: if $backtrack(E)$ does not already contain such a transition, then one is added. Then $w'.t' \sim_{[E]} u.t'$ and t'' in $I_{[E]}(u.t')$ implies $t'' \in WI_{[E]}(w'.t')$, i.e., $t'' \in WI_{[E]}(w)$. This violates the assumption $WI_{[E]}(w) \cap backtrack(E) = \emptyset$, and we have derived our contradiction.

The conclusion in this case is that the dynamic computation of source works if it has the property that whenever a race of the form $e \precsim_{E.t.w'''.t'} e'$ is encountered during exploration, then $backtrack(E)$ must contain some transition in $I_{[E]}(u.t')$, where $u.t'$ is as above.

2. $E \not\vdash (t.w'.t')$, i.e., t' is disabled by t after w'. In this case, we can only rely on the fact that $E \vdash (t.w')$. As in the previous case, it is guaranteed that a sequence of the form w'' such that $w' \sqsubseteq_{[E.t]} w''$ will be explored from $s_{[E.t]}$ We would now like to find some prefix of w'' which plays the same role as w''' in the preceding case, i.e., such that thereafter t' could be executed but has been disabled by t. If so, we could detect a "race" between the (blocked) execution of t' and e (corresponding to t), and proceed as in case 1. The problem is that, in general, there may not be such a prefix. Namely, it may be the case that w'' contains other transitions of $\widehat{t'}$ that are independent from the events in w', and change the control location of $\widehat{t'}$ so that t' cannot be performed even if it had not been disabled by t. This can be solved by requiring that the race detection in the construction of source sets consider the possible subsequences u of w'' that may be equivalent with w', for which $w' \simeq_{[E]} u$ could be possible. Since the race detection does not know the sequence w''', it must consider any subsequence u with $u \sqsubseteq_{[E]} w''$. For any such subsequence, we let u play the same role as in the previous case, and continue as in the previous case.

In conclusion, we have derived requirements on how to add transitions to the set $backtrack(E)$, which guarantee that $backtrack(E)$ is a source set when the exploration from $s_{[E]}$ has completed. The above inductive hypothesis can then be used to prove that the resulting DPOR algorithm explores all equivalence classes of executions, in the same way as in Theorem 2.

Let us now collect this reasoning into an algorithm. Algorithm 1 shows an adaptation of the Source-DPOR algorithm [2, Algorithm 5] to our setting, without the use of sleep sets.

Source-DPOR uses the recursive procedure $Explore(E)$ to perform a depth-first search, where E can be interpreted as the stack of the search, i.e. the past execution sequence explored so far. The algorithm maintains, for each $E' \leq E$, a set $backtrack(E')$ of transitions that will eventually be explored from

ALGORITHM 1: Source-DPOR algorithm.

Initial call : $Explore(\langle\rangle)$

```
1  Explore(E)
2      foreach e ∈ dom(E) do
3          let E' = pre(E, e);
4          let w = notdep(e, E);
5          foreach subsequence u of w such that u ⊑_[E'] w do
6              foreach transition t that is blocked in s_[E] do
7                  if [e] disables t after E.u then
8                      let v = u.t;
9                      if I_[E'](v) ∩ backtrack(E') = ∅ then
10                         add some t' ∈ I_[E'](v) to backtrack(E');
11     if there is some enabled process p at s_[E] then
12         backtrack(E) := all enabled transitions of p;
13         while ∃t ∈ backtrack(E) do
14             foreach e ∈ dom(E) such that(e ≾_E.t next_[E](t)) do
15                 let E' = pre(E, e);
16                 let v = notdep(e, E).t;
17                 if I_[E'](v) ∩ backtrack(E') = ∅ then
18                     add some t' ∈ I_[E'](v) to backtrack(E');
19             Explore(E.t);
```

E'. Each set $backtrack(E)$ is a set of transitions that are enabled from $s_{[E]}$. It will be gradually expanded during the exploration; at the end it will be a source set after E. $Explore(E)$ initializes $backtrack(E)$ to consist of all enabled transitions of some arbitrary process (line 12). Thereafter, for each transition t in $backtrack(E)$ the algorithm performs two phases: race detection (lines 14–18) and state exploration (line 19). In addition, for each explored execution E, it performs a race detection for disabled transitions (lines 2–10).

In the race detection phase, which corresponds to case 1 above and starts at line 14, the algorithm finds the events $e \in dom(E)$ that are in a reversible race with the next step of t. This next step of t corresponds to event e' in the above case 1. For each such event $e \in dom(E)$, the algorithm must explore execution sequences in which the race is reversed. This is done as explained in case 1 above, where E' in the algorithm corresponds to E in case 1, where E in the algorithm corresponds to $E.t.w'''$ in case 1, where t in the algorithm corresponds to t' in case 1, and where v in the algorithm corresponds to $u.t'$ in case 1.

The race detection for disabled transitions, corresponding to case 2 above, is at lines 2–10. It is done for each execution sequence E, as explained in the text in case 2, where E' in the algorithm corresponds to E in case 2, where E in the algorithm corresponds to $E.t.w''$ in case 2, where u in the algorithm corresponds to u in case 2, and where t in the algorithm corresponds to t' in case 2.

6 Relating Source Sets and Persistent Sets

In this section, we provide some comparisons between source sets and persistent sets. We first define persistent sets. Thereafter, we show some properties of source sets and persistent sets that distinguish them from each other, followed by theorems that provide conditions under which source sets can be strictly smaller than persistent sets.

6.1 Persistent Sets

Let us first define persistent sets in our framework. Adapted to our context, a set T of transitions is a persistent set for W after s if for each sequence in W, the first step that is dependent with the first step of some transition in T must be taken by some transition in T. A formalization could go as follows.

Definition 5 (Persistent Sets). *Let s be a state, and let $W \subseteq \mathcal{E}(s)$ be a set of execution sequences from s. A set T of transitions is* a persistent set for W after s *if for each prefix w of some sequence in W, which contains no occurrence of a transition in T, we have $E \vdash t \diamondsuit w$ for each $t \in T$.*

We say that T is a *persistent set after s* to mean that T is a *persistent set for $\mathcal{E}(s)$ after s*. We note that the definition of persistent sets is slightly more complex than the definition of source sets. In particular, since its definition involves universal quantification over all elements of the persistent set, the elements of a persistent set are not independent of each other. This is noted in the following theorem.

Theorem 4. *If T is a source set after s and $T \subseteq T'$, then T' is a source set after s. This property does not hold for persistent sets, i.e., it is possible that T is a persistent set after s, but T' is not.*

In other words, persistent sets have the unpleasant property that adding a process may disturb the persistent set so that even more process may have to be added. This property is relevant in the context of DPOR, where the first member of the persistent set is often chosen rather arbitrarily (it is the next process in the first exploration after E), and where the persistent set is expanded by need.

For source sets, Theorem 4 follows directly from Definition 4. For persistent sets, a counter-example is provided by the program in Fig. 1, where $\{q\}$ is a persistent set, but $\{p, q\}$ is not.

Continuing the comparison between source sets and persistent sets, we first note some rather direct properties, including the following.

- *Any persistent set is a source set.*
- *Any one-process source set is a persistent set.*

An interesting question is then whether there are situations where any persistent set contains a strictly smaller source set. We note that the program in Fig. 1 does not illustrate such a situation, since the smallest persistent sets and the smallest source sets coincide: they are either $\{q\}$ or $\{r\}$. Nevertheless, the answer to this question is yes, and we formulate this as a theorem.

Theorem 5. *There are programs for which any persistent set from the initial state contains a strictly smaller source set.*

Proof. We give an example. In Fig. 4, the three processes p, q, and r perform dependent accesses to the shared variables `x`, `y`, and `z`. Two accesses are dependent if they access the same variable and at least one is a write. For this program, there are 75 possible execution sequences, partitioned over 7 Mazurkiewicz traces (there are 8 ways to direct the three races in the program, but it is not possible to let the read precede the write in all of them).

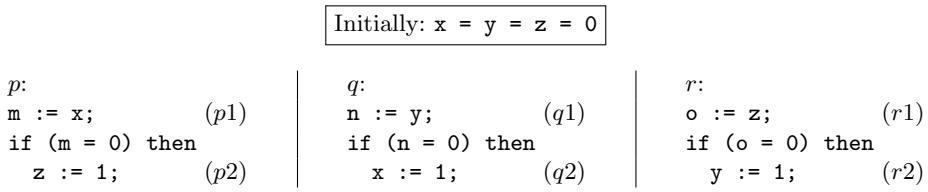

Fig. 4. Program with non-minimal persistent sets.

In Fig. 4, it is obvious that a single transition cannot be a source set. For instance, the set $\{p1\}$ does not contain the initials of execution $q1.q2.p1.r1.r2$, since $q2$ and $p1$ perform conflicting accesses. On the other hand, any subset containing two enabled transitions is a source set. To see this, let us choose $\{p1, q1\}$ as the source set. Obviously, $\{p1, q1\}$ contains an initial of any execution that starts with either $p1$ or $q1$. Any execution sequence which starts with $r1$ is equivalent to an execution obtained by moving the first step of either $p1$ or $q1$ to the beginning:

- If $q1$ occurs before $r2$, then $q1$ is an initial, since it does not conflict with any other transition.
- If $q1$ occurs after $r2$, then $p1$ is independent of all steps, so $p1$ is an initial.

We claim that $\{p1, q1\}$ cannot be a persistent set. The reason is that the execution sequence $r1.r2$ does not contain any transition in the persistent set, but its second step is dependent with $q1$. By symmetry, it follows that no other two-transition set can be a persistent set. □

6.2 Fine-Grained Source Sets

It follows rather directly from the definition that for persistent sets we have the following property:

> *If some enabled transition of process p is in the persistent set at s, then all enabled transitions of process p are in the persistent set at s.*

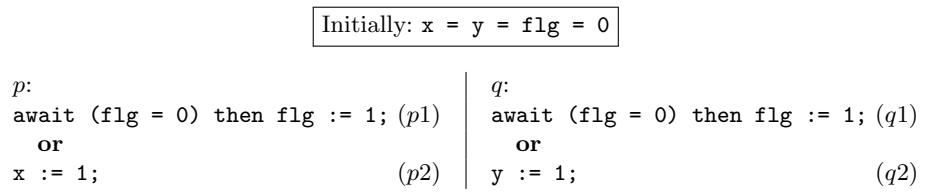

Fig. 5. Program with fine-grained source set.

The property follows by observing that if transition t of process p is in the persistent set, and t' is some other enabled transition of p, then the execution sequence t' is dependent with t, and therefore t' must be in the persistent set.

For source sets, however, this property need not hold. Consider the program in Fig. 5. Here the transitions $(p1)$ and $(q1)$ should be interpreted as guarded commands that are enabled only when `flg = 0`. (It is possible to make example programs also without disabling transitions, but merely changing their effect depending on some test.) We claim that the set $\{(p2), (q2)\}$ is a source set from the initial state. For instance, if an execution sequence E begins with $(p1)$, then $(q1)$ is disabled, so $(q2)$ is a weak initial of E.

6.3 Excluding Conditionals

We note that the example in Fig. 4 uses conditionally executed statements. This is not a coincidence. In fact, it turns out that if the code of the processes does not contain any branches or conditionals, i.e., all executions of a process access the same variables in the same order, then Theorem 5 does not hold as long as processes are deterministic. Still recall however, that non-minimal source sets need not be persistent sets.

If we allow non-determinism, then it was shown in Fig. 3 that there are simple programs whose minimal source sets are strictly included in any persistent sets. In the absence of both conditionals and non-determinism, minimal source sets and persistent sets are the same, as shown in the following theorem.

Theorem 6. *For programs, in which each process performs a single sequence of unconditional reads and writes, minimal source sets coincide with minimal persistent sets.*

Proof. To see this, consider a program consisting of a set of processes, in which each process executes a deterministic straight-line program without conditionals. We can uniquely identify a transition by the process that executes it. Let us consider the program in its initial state. For processes p, q, let $p \rhd q$ denote that the first step of p is dependent with *some* step of q. It can easily be seen that persistent sets satisfy the following property:

P *is a persistent set iff* $p \in P$ *and* $p \rhd q$ *implies* $q \in P$.

This follows rather naturally from the definition of persistent sets: if $p \in P$ and $p \rhd q$ and $q \notin P$, then consider the sequence of steps of q up to and including the step that is dependent with the first step of p. This sequence is outside P, but is dependent with the first step of some process in P, hence P is not a persistent set. The above property implies that each minimal persistent set is a terminal strongly connected component (SCC) in the direct graph whose nodes are processes, and whose directed edges are defined by the relation $\rhd$. Conversely, a terminal SCC P is a persistent set, since any execution sequence that does not contain steps of processes in P is by definition independent of the first steps ot processes in P.

For source sets, it also holds that a source set must include a terminal SCC in the direct graph defined by the relation $\rhd$. Namely, suppose some set T does not include a terminal SCC. This means that for each $p_1 \in T$ there is a sequence of processes $p_1\ p_2 \ldots p_n$ with $p_i \in T$ and $p_i \rhd p_{i+1}$ for $i = 1, \ldots, n-1$, but $p_n \notin T$. Consider a longest such sequence. We first assume that it includes all processes in T. Let the execution sequence E consist of all the steps of process p_n, thereafter the steps of process p_{n-1} and so on, until finally all steps of p_1. It is obvious that no process in T can be a weak initial of E. This argument can also be extended to the case where there is no sequence that includes all processes in T. Conversely, any set which includes a terminal SCC is a source set, since for any execution sequence E we can take as initial the first occurring process of that SCC. □

7 Concluding Remarks

In this paper, we have shown that source sets are suitable as a general foundation for partial order reduction. We have shown that source sets can be used both in enumerative model checking, as performed by SPIN, and in stateless model checking for possibly non-deterministic terminating programs. We have also highlighted some differences between source sets and persistent sets, thereby providing some insights into conditions under which source sets and persistent sets do or do not differ.

Since source sets are more succinct, and often strictly more succinct than corresponding persistent sets, it means that source sets can replace persistent sets as a foundation for partial order reduction.

Acknowledgments

We would like to thank the anonymous reviewers for comments and suggestions that have improved the presentation.

References

1. Abdulla, P., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) Tools

and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015. Proceedings. LNCS, vol. 9035, pp. 353–367. Springer, Berlin Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-46681-0_28
2. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Proceeedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 373–384. POPL '14, ACM, New York, NY, USA (2014)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
4. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR'98 Concurrency Theory, 9th International Conference, Proceedings. LNCS, vol. 1466, pp. 485–500. Springer, Berlin, Heidelberg (1998)
5. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) Hybrid Systems III: Verification and Control. LNCS, vol. 1066, pp. 232–243. Springer, Berlin, Heidelberg (1996), http://dx.doi.org/10.1007/BFb0020949
6. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logics specification: A practical approach. In: Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages. pp. 117–126. ACM Press (1983), http://doi.acm.org/10.1145/567067.567080
7. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. International Journal on Software Tools for Technology Transfer 2(3), 279–287 (Nov 1999), http://dx.doi.org/10.1007/s100090050035
8. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 110–121. POPL '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1040305.1040315
9. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Ph.D. thesis, University of Liège (1996), also, volume 1032 of LNCS, Springer.
10. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. In: Larsen, K.G., Skou, A. (eds.) Computer Aided Verification. LNCS, vol. 575, pp. 332–342. Springer-Verlag, London, UK (1991), http://dx.doi.org/10.1007/3-540-55179-4_32
11. Holzmann, G.: The model checker SPIN. IEEE Trans. on Software Engineering SE-23(5), 279–295 (May 1997)
12. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 165–174. PLDI 2015, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2737924.2737975
13. Kähkönen, K., Saarikivi, O., Heljanko, K.: Unfolding based automated testing of multithreaded programs. Autom. Softw. Eng. 22(4), 475–515 (Dec 2015), http://dx.doi.org/10.1007/s10515-014-0150-6
14. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: Rosenblum, D.S., Taentzer, G. (eds.) Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010. LNCS, vol. 6013, pp. 308–322. Springer, Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-12029-9_22

15. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Applications and Relationships to Other Models of Concurrency. LNCS, vol. 255, pp. 279–324. Springer, Berlin Heidelberg (1987), http://dx.doi.org/10.1007/3-540-17906-2_30
16. McMillan, K.L., Probst, D.K.: A technique of a state space search based on unfolding. Formal Methods in System Design 6(1), 45–65 (Jan 1995), http://dx.doi.org/10.1007/BF01384314
17. Overman, W.: Verification of Concurrent Systems: Function and Timing. Ph.D. thesis, UCLA (Aug 1981)
18. Peled, D.: All from one, one for all, on model-checking using representatives. In: Courcoubetis, C. (ed.) Computer Aided Verification. LNCS, vol. 697, pp. 409–423. Springer-Verlag, London, UK (1993)
19. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer Verlag, Berlin Heidelberg (1982), http://dx.doi.org/10.1007/3-540-11494-7_22
20. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: 26th International Conference on Concurrency Theory, CONCUR 2015. LIPIcs, vol. 42, pp. 456–469. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015), http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.456
21. Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: Application of Concurrency to System Design (ACSD), 12th International Conference on. pp. 132–141. IEEE, Los Alamitos, CA, USA (Jun 2012)
22. Salah, R.B., Bozga, M., Maler, O.: On interleaving in timed automata. In: Baier, C., Hermanns, H. (eds.) Concurrency Theory, 17th International Conference, CONCUR 2006, Proceedings. LNCS, vol. 4137, pp. 465–476. Springer (2006)
23. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Baresi, L., Heckel, R. (eds.) Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer, Berlin Heidelberg (2006)
24. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Bin, E., Ziv, A., Ur, S. (eds.) Haifa Verification Conference. LNCS, vol. 4383, pp. 166–182. Springer, Berlin Heidelberg (2007)
25. Serbanuta, T., Chen, F., Rosu, G.: Maximal causal models for sequentially consistent systems. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Third International Conference, RV 2012, Revised Selected Papers. LNCS, vol. 7687, pp. 136–150. Springer, Berlin Heidelberg (Sep 2012), http://dx.doi.org/10.1007/978-3-642-35632-2_16
26. 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.) Formal Techniques for Distributed Systems. LNCS, vol. 7273, pp. 219–234. Springer, Berlin Heidelberg (2012)
27. 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-Verlag, London, UK (1991)