Supporting Domain-Specific State Space Reductions through Local Partial-Order Reduction

Péter Bokor
TU Darmstadt
Darmstadt, Germany
pbokor@cs.tu-darmstadt.de

Johannes Kinder
EPFL
Lausanne, Switzerland
johannes.kinder@epfl.ch

Marco Serafini
Yahoo! Research
Barcelona, Spain
serafini@yahoo-inc.com

Neeraj Suri
TU Darmstadt
Darmstadt, Germany
suri@cs.tu-darmstadt.de

***Abstract*—Model checkers offer to automatically prove safety and liveness properties of complex concurrent software systems, but they are limited by state space explosion. Partial-Order Reduction (POR) is an effective technique to mitigate this burden. However, applying existing notions of POR requires to verify conditions based on execution paths of unbounded length, a difficult task in general. To enable a more intuitive and still flexible application of POR, we propose *local* POR (LPOR). LPOR is based on the existing notion of statically computed stubborn sets, but its locality allows to verify conditions in single states rather than over long paths.**

As a case study, we apply LPOR to message-passing systems. We implement it within the Java Pathfinder model checker using our general Java-based LPOR library. Our experiments show significant reductions achieved by LPOR for model checking representative message-passing protocols and, maybe surprisingly, that LPOR can outperform dynamic POR.

I. Introduction

The use of formal verification methods can avoid failures in the design or implementation of a system and is thus of growing importance for the development processes of complex software. A successful and widely used method is model checking [8], which allows the fully automated verification of temporal properties. Model checking is limited by state explosion, however, a fundamental problem in verification, especially of concurrent systems.

The state space explosion problem can be greatly mitigated by Partial-Order Reduction (POR) [8], a general concept for reducing the model checking resources such as memory and time. Several notions of POR implement this concept [8], [23], [11], differing from each other in flexibility and efficiency. The commonality of these approaches is that the developer of a model checker is expected to verify complex conditions to guarantee soundness. This hurdle can prevent developers from implementing POR or even lead to erroneous implementations. In this paper, we propose an approach that *simplifies* the conditions to be verified, but gives up neither the *flexibility* nor the *efficiency* of POR. Next, we explain why previous notions of POR are difficult to use and how our approach improves on them.

The general concept of POR lies in the commutativity of non-interfering transitions. Conceptually, a transition is a mechanism to change the state of the system, e.g., a Java method, or the delivery of a message. POR is based on the simple observation that the execution of non-interfering transitions leads to the same state irrespective of which of these transitions is executed first. In Figure 1, t_1 and t_2 are non-interfering because both paths $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_{12}$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s_{12}$ lead to s_{12}. Therefore, it is sufficient to explore the execution of these transitions in a single representative order, reducing memory and time required for model checking.

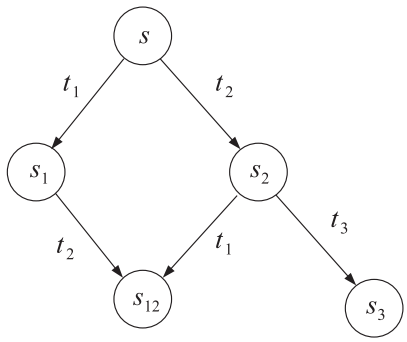

Fig. 1. Non-interfering transitions t_1 and t_2. States s_{12} and s_3 are deadlocks.

POR is *sound* if no state is missed that is relevant for verifying the target property. For example, although t_1 and t_2 are non-interfering, it is an unsound reduction to explore only the path $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_{12}$ if the property states the reachability of s_3. Existing notions of POR define necessary conditions of soundness that are hard to check in general because they require global knowledge about the state graph, which limits the applicability of POR. This problem is usually addressed by fixing the application of POR to a particular specification language and computational model, such that soundness is guaranteed by construction. As a result, existing specification languages with POR support are few and restrictive in different ways: they consider restricted computational models, for example FIFO-based message-passing [14], [13], Petri nets or process algebras [23], they only allow models with deterministic transitions [11], [8] or acyclic state graphs [10], [21], they preserve only invariants [12], [10], [15], or they only support bug finding [15].

We present a novel take on POR, to ease its application to rich specification languages. We call our approach *local* POR

Research supported in part by EC FP7 INDEXYS, LOEWE TUD CASED, and DFG TUD GKMM GRK 1362.

(LPOR) because locality is key to simplify the verification of POR conditions for designing new model checkers; in fact, the simplicity of LPOR allows an easy development of new PO reductions. LPOR consists of an input interface (accessible by the user of LPOR) and a POR algorithm (hidden from the user).[1] At the interface of LPOR, the user defines locally "interfering" transitions, whose soundness can be verified more easily than the global (path-based) soundness conditions in other POR approaches. This local information is sufficient for our LPOR algorithm to efficiently compute sound partial-order reductions. In the example of Figure 1, the user can define and verify the following local interferences: t_2 can enable t_3 (when executed in s), and t_1 is dependent on (is disabled by) t_3 (when executed in s_2). Based on this information, the LPOR algorithm knows that t_1 and t_2 are non-interfering and can establish that exploring only the paths $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s_{12}$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_3} s_3$ preserves all deadlock states, a fundamental preservation property used by LPOR to preserve more complex specifications.

In the following, we further detail our main contributions.

LPOR stubborn set algorithm. LPOR's interface (Section II) contains two intuitive relations between transitions, namely *can-enable* and *dependency*. Each of these relations is local, i.e., they are defined given paths of at most length two. Transitions that are not included in these relations are considered to be non-interfering and are used by LPOR to achieve reduction. The user has to prove the non-interferences correct, but it is sound to declare transitions as interfering even when they are not. An important feature of LPOR is that non-interfering transitions are completely configurable, while other approaches conservatively assume certain transitions to be interfering, e.g., transitions executed by the same process [8]. LPOR also supports *necessary enabling transitions*, which we generalize from [11]. Although the definition of such transitions does involve paths, they naturally appear in high-level languages.

The LPOR algorithm (Section III) computes stubborn sets statically [23] and supports general transition systems without assumptions about the state graph or transitions. Intuitively, a stubborn set is a large enough subset of the transitions enabled in the current state, e.g., $\{t_2\}$ in s in Figure 1, such that no deadlock state remains unvisited if only transitions in stubborn sets are executed. LPOR leverages stubborn sets to preserve properties in the temporal logic CTL^*_{-X}. LPOR is fast thanks to a novel *pre-computation* scheme, which allows to compute information needed by LPOR once, before model checking, and then to repeatedly use it in every new state.

Applying LPOR to message-passing. We instantiate the relations at LPOR's interface for general message-passing systems (Section IV). This example also shows that the use of LPOR is straightforward for domain experts.

We briefly discuss two additional LPOR application examples. First, we use a Petri net example in explaining the LPOR algorithm (Section III-B). Second, we show how the POR approach used in the SPIN model checker can be expressed in LPOR terms (Section VII).

Experiments and comparison with DPOR. We implement LPOR as an openly available Java library called Java-LPOR (Section V) that easily integrates with existing model checkers. As an example use case of Java-LPOR, we implement our message-passing instantiation of LPOR in the Java Pathfinder-based model checker MP-Basset [5].

We evaluate the efficiency of LPOR using message-passing examples. Our experiments with MP-Basset show that LPOR achieves significant (up to 94%) time and space reductions for model checking real-world fault-tolerant message-passing protocols (Section VI). Furthermore, countering current notions of dynamic POR being superior to static POR [10], we also show that LPOR (implementing static POR) competitively improves upon dynamic POR without entailing the constraints of dynamic POR.

II. The LPOR Interface

The typical application scenario of LPOR is adding POR to the analysis of systems written in some specification language. Assume that a model checker implementing the LPOR algorithm (Section III) is available for this language. We will show in Section V how we support the integration of LPOR into existing model checkers. Now, the user, an expert in the domain of the language, must provide two inputs at LPOR's interface. First, unless it is not already available, she must define the semantics of the language in terms of a state transition system (Section II-A). Second, based on her domain-specific knowledge, she defines and proves two intuitive relations containing pairs of interfering transitions (Section II-B). These relations are local considering paths of length at most two. LPOR leverages a third optional relation, which is not strictly local, but naturally appears in high-level languages.

A. Non-deterministic State Transition Systems

A state transition system (STS) is a triple (S, T, S_0) where S is the set of states, T is the set of transitions, and $S_0 \subseteq S$ is the set of initial states. Every *transition* $t \in T$ is a relation $t \subseteq S \times S$. A transition t is *enabled* in $s \in S$ iff there is an $s' \in S$ such that $(s, s') \in t$. Otherwise, t is *disabled* in s. The set of all enabled transitions in s is denoted by $enabled(s)$. A state $s \in S$ is called a *deadlock* if $enabled(s) = \emptyset$. We write $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s_n$ and say that there is a path from s_0 to s_n iff for every $0 \leq i < n$ we have that $(s_i, s_{i+1}) \in t_{i+1}$. In this case, we say that s_n is *reachable* from s_0. If $s_0 \in S_0$, then we say that s_n is reachable. A transitions t is said to be *in* a path $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s_n$ if t is among $t_1, t_2, \ldots, t_n$.

Our approach allows transitions to be *non-deterministic*, i.e., given $t \in T$ and $s \in S$, there might be multiple $s' \in S$ such that $(s, s') \in t$. Other approaches, e.g., [11], [8], are restricted to deterministic transitions. On the one hand, while a transition system always allows to refine a non-deterministic transition into several deterministic transitions, an implementation of

[1] In the remainder of this paper, by *user* we mean the user of LPOR and not necessarily the end-user of the model checker.

such a refinement is not necessarily straightforward for a particular system model. Furthermore, the performance of POR algorithms can be adversely affected by an increase in $|T|$, the number of all transitions. On the other hand, refining transitions can improve space reduction, since only some of the refined transitions might have to be contained in a stubborn set [11], [5]. Not requiring deterministic transitions leaves a larger design space for exploring trade-offs in transition refinement.

B. Interfering Transitions

A transition t *can enable* another transition t', if in at least one state where t' is disabled, executing t results in a state where t' is enabled. We say that a relation is *can-enabling* if it is a superset of all pairs (t, t') of transitions such that t can enable t'.

Definition 1: A relation $\mathbf{ce} \subseteq T \times T$ is *can-enabling* iff $\mathbf{ce} \supseteq \{(t,t') \,|\, \exists s, s' \in S : \; s \xrightarrow{t} s' \wedge t' \notin enabled(s) \wedge t' \in enabled(s')\}$.

We define that t' is *dependent* on t if both t and t' are enabled in some state (t and t' are co-enabled) and either (a) t can disable t' or (b) their subsequent execution in different orders results in different states (t and t' do not commute). By convention, t is not dependent on itself. We say that two transitions are dependent (independent) if one (none) of them is dependent on the other. Note that the following relation is not necessarily symmetric.

Definition 2: A relation $\mathbf{dep} \subseteq T \times T$ is a *dependency* relation, iff $\mathbf{dep} \supseteq \{(t,t') \,|\, t \neq t' \wedge \exists s, s' \in S : \; t, t' \in enabled(s) \wedge s \xrightarrow{t} s'$ and either (a) $t' \notin enabled(s')$ or (b) $\exists s'' \in S : \; s \xrightarrow{tt'} s''$ and not $s \xrightarrow{t't} s''\}$.

Next, we define a relation that contains a pair of transitions t and t' if t' is a *necessary enabling transition (NET)* for t, i.e., t' must be executed at least once for t to be enabled (adapted from necessary enabling sets [11]). Note that this relation is based on paths. It is purely optional though as it is sound to not include pairs of transitions in a NET relation or, in particular, to define an empty one. Similarly, it is always sound to include a pair of (even non-interfering) transitions in can-enabling and dependency relations.

Definition 3: A relation $\mathbf{net} \subseteq T \times T$ is a *necessary enabling transition (NET)* relation, iff $\mathbf{net} \subseteq \{(t,t') \,|\, \forall s_0 \in S_0, \forall s \in S, \forall t_1, \ldots, t_n \in T$: if $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s \wedge t \in enabled(s)$, then $t' = t_i$ for some $1 \leq i \leq n\}$.

Note that the transitive closure of every NET relation is also a NET relation. Every user-provided NET relation can thus be extended to its closure.

III. The LPOR Stubborn Set Algorithm

Now we present LPOR, our local partial-order reduction algorithm. Formally, LPOR computes stubborn sets [23], which are subsets of $enabled(s)$ in a state s such that it is sufficient to explore transitions in such a subset. LPOR can be configured to preserve properties from simple deadlock-freedom to arbitrary LTL$_{-X}$ and CTL$^*_{-X}$ specifications. LPOR can be adapted to similar POR semantics such as ample [8] or persistent sets [11]. We chose stubborn sets because they allow the most relaxed system model. For example, both persistent and ample sets assume deterministic transitions.

LPOR is a *static* POR algorithm, i.e., given a state s of the system, LPOR outputs a stubborn set in s without further exploration (as opposed to dynamic POR [10]). Therefore, LPOR can be implemented in stateful (even parallel [22]) explicit-state model checking. We present a simplified variant of the LPOR algorithm that assumes that the search path, i.e., a path from an initial state to s, is available. The search path can be obtained by depth-first search. However, a generalized form of LPOR makes no assumption about the search path and is compatible with both depth and breadth-first search. Therefore, it is amenable to symbolic (Binary Decision Diagram-based) implementations [3] as well. For space reasons, the generalized LPOR algorithm is presented Appendix I.[2]

We first review stubborn sets (Section III-A), then we present the core LPOR algorithm and sketch its correctness, i.e., LPOR indeed computes stubborn sets (Section III-B). Then, we discuss some optimizations of LPOR (Section III-C) and the preservation of general temporal properties (Section III-D).

A. Preliminaries: Stubborn Sets

Given a state s_0, a set $stub(s_0)$ of transitions is *(weakly) stubborn* if the two properties D1 and D2 are satisfied [23]. D1 verifies the commutativity of transitions in the stubborn set with transitions outside the stubborn set. D2 ensures that there is at least one transition that cannot be disabled by transitions outside the stubborn set.

- D1 $\forall t \in stub(s_0), \forall t_1, t_2, \ldots, t_n \in T \setminus stub(s_0), \forall s_n \in S$: if $s_0 \xrightarrow{t_1 t_2 \ldots t_n t} s_n$ then $s_0 \xrightarrow{t t_1 t_2 \ldots t_n} s_n$.
- D2 If $enabled(s_0) \neq \emptyset$ then $\exists t \in stub(s_0), \forall t_1, t_2, \ldots, t_n \in T \setminus stub(s_0)$: if $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s_n$ then $t \in enabled(s_n)$. Such a transition t is called *key transition*.

A stubborn set is called *strong* if every $t \in stub(s_0) \cap enabled(s_0)$ is a key transition. Note that a key transition is always enabled in s_0. The *unreduced state graph* is explored by starting from an initial state and executing every transition in $enabled(s)$ when a new state s is visited. The *reduced state graph* is obtained by executing only the enabled transitions from $stub(s)$. If $t \in stub(s)$ and t is non-deterministic, then every s' with $(s, s') \in t$ is visited. D1 and D2 guarantee that all deadlocks of the unreduced state graph are contained in the reduced one. In order to preserve properties other than deadlock-freedom, $stub(s_0)$ needs to satisfy additional constraints [23], [16]. Note that transitions in $stub(s)$ are not necessarily enabled in s. Although disabled transitions cannot be executed, they can ease the design of stubborn set algorithms [11] and even result in smaller stubborn sets when used to preserve certain temporal properties [23].

[2] Appendices are included in the technical report version of this paper available online [6].

```
   function FwdEnableSetIdx(t, t')
1    forall (t'', en) ∈ FwdEnableSet(t) do
2        if (t'', t') ∈ dep then return true;
3    return false;
   function FwdEnableSet(tr)
4    Tr' ← {(tr, ∅)};
5    do
6        Tr ← Tr';
7        forall t_1 ∈ T do
8            forall (t, en) ∈ Tr do
9                if (t, t_1) ∈ ce then
10                   en_1 ← en ∪ {t_2 | (t_1, t_2) ∈ net};
11                   Tr' ← Tr' ∪ {(t_1, en_1)};
     while Tr ≠ Tr';
12   return Tr;
```

Algorithm 1: $FwdEnableSet(t)$ and $FwdEnableSetIdx(t, t')$ are pre-computed for every $t, t' \in T$.

```
13 Stub ← {t_I};
14 Trans ← {t_I};
15 while Trans ≠ ∅ do
16     choose t ∈ Trans;
17     Trans ← Trans \ {t};
18     forall t_1 ∈ enabled(s) \ Stub do
19         if (t_1, t) ∈ dep then
20             Stub ← Stub ∪ {t_1};
21             if dep is non-transitive then Trans ← Trans ∪ {t_1};
22         else if FwdEnableSetIdx(t_1, t) then
23             if ∃(t_dep, en) ∈ FwdEnableSet(t_1) : (t_dep, t) ∈ dep
24             ∧(en = ∅ ∨ ∀t_2 ∈ en : (t_2 ∉ Stub ∨ t_2 ∈ τ)) then
25                 Stub ← Stub ∪ {t_1};
26                 Trans ← Trans ∪ {t_1};
27 return Stub;
```

Algorithm 2: The LPOR(t_I, s, τ) stubborn set algorithm for a state $s \in S$, an initial transition $t_I \in enabled(s)$, and a current search path $\tau \in T^*$.

B. The Stubborn Set Algorithm

As stated before, the use of NET in LPOR is optional. We therefore start out by explaining the LPOR algorithm (Algorithm 2) without the NET optimization where $\mathbf{net} = \emptyset$.

1) Forward enable sets: LPOR uses two helper functions $FwdEnableSetIdx(t, t')$ and $FwdEnableSet(t)$ (Algorithm 1), whose return values can be pre-computed (before model checking), because they are independent of the state. The first function returns true if t can be the first in a sequence of enabling transitions that enables another transition t'' on which t' is dependent (lines 1-3). $FwdEnableSetIdx$ is defined based on the *forward enable set* $FwdEnableSet(t)$ of t, which contains those transitions that can be enabled through a sequence of enabling transition starting with t (lines 4-12). More precisely, the set contains all transitions t' such that (t, t') is in the transitive closure of a can-enabling relation $\mathbf{ce}$. The set contains tuples of the form (t, en) where t is a transition and en is a set of transitions, which is used in the NET-optimized version of LPOR. If the NET relation is empty, en is also empty (line 10). We now explain how LPOR uses these two functions to compute stubborn sets.

2) Stubborn set computation: In addition to the relations $\mathbf{ce}$, $\mathbf{dep}$, and $\mathbf{net}$, LPOR has three parameters: (1) a transition $t_I \in enabled(s)$, called *initial transition*, which is in the stubborn set, (2) the current state s, and (3) the search path $\tau \in T^*$ (for Algorithm 2, it suffices that τ is a set containing $t_1, \ldots, t_n$). From D2, no stubborn set in s can be empty unless $enabled(s) = \emptyset$. Conceptually, LPOR proceeds, similarly to other static POR algorithms, by applying different rules of the form "if t is in the stubborn set, then transitions $t_1, t_2, \ldots$ must also be in the set". In this case, we say that $t_1, t_2, \ldots$ are added *on behalf of* t. LPOR maintains two sets of transitions: *Stub*, which represents the stubborn set (line 13) and *Trans*, which contains a transition t in *Stub* such that new transitions might be added to *Stub* on behalf of t (line 14). Therefore, LPOR adds transitions to *Stub* until *Trans* is empty (lines 15-26) and *Stub* is returned (line 27). We now explain how transitions are added on behalf of a transition t in *Trans*.

First, we add those enabled transitions t_1 that t is dependent on (lines 19-21). We add t_1 if either t_1 and t do not commute (disallowed by D1) or it can disable t (which can violate D2). Note that $\mathbf{dep}$ does not have to be symmetric as D1 allows that t and t_1 do not commute. We will show an example of this case in a message-passing instance of LPOR (Section IV).

There is another way to violate the stubborn set conditions: an enabled transition t_1 outside the stubborn set can start a sequence of enabling transitions that enables another transition on which t is dependent (D1). This can only happen if $FwdEnableSetIdx(t_1, t)$ is true (line 22). In this case, we add t_1 to the stubborn set (line 25). Note that the condition in lines 23-24 is trivially true if LPOR is run without NET optimization because the en-sets are empty.

In both previous cases, t_1 is added to *Trans* (line 21 and 26) so that LPOR can verify whether new transitions must be added on behalf of t_1. We discuss the optimization for transitive dependency relations (line 21) in Section III-C.

3) NET optimization: Stubborn set computation can benefit from the NET relation if more than one transition t_2 is necessary for some transition t_1 to be enabled. In this case, a stubborn set does not need to contain *all* such t_2 but only *one* that has not been executed yet. The NET optimization cannot be fully pre-computed as the check whether "a transition has not been executed yet" can only be carried out during the search. However, we can store these t_2 transitions in the en-field associated with t_1. It is key to our NET optimization that the content of en-fields is propagated along the can-enabling relation, i.e., if t can enable t_1 and (t, en) and (t_1, en_1) are in a forward enable set, then $en \subseteq en_1$ (line 10). This is because the transitions necessary to be executed for t to be enabled are, transitively, also necessary to be executed for t_1 to be enabled.

Then, using the notation of Algorithm 2, if some t_2 is in the en-field associated with a transition $t_{\mathbf{dep}}$, we can verify, given the current state s, that "t_2 has not been executed yet". Assume that $(t_{\mathbf{dep}}, en)$ is in the forward enable set of t_1 and the conditions in lines 22-23 are true. Then, we only add t_1 to the stubborn set if either t_2 is not in the stubborn set or t_2 has already been executed, i.e., is contained in the model checker's current search path τ (line 24). Note that, for some transition t, (t, en) can be in a forward enable set multiple

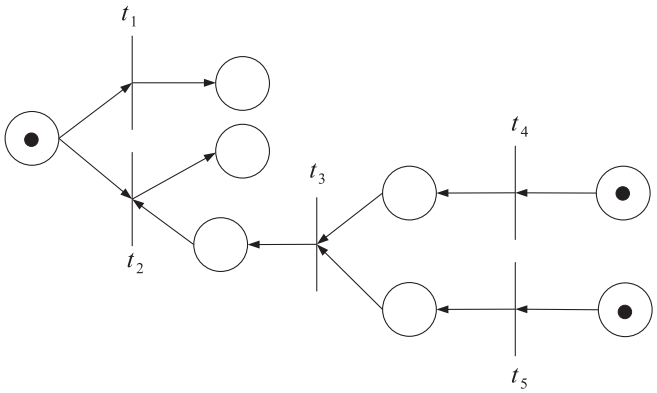

Fig. 2. A Petri net example.

times with different *en*. This is possible if t can be enabled by different sequences of transitions.

4) Example: We illustrate the LPOR algorithm on a simple Petri net example (Figure 2). For this net, $\mathbf{ce} = \{(t_3, t_2), (t_4, t_3), (t_5, t_3)\}$, $\mathbf{dep} = \{(t_1, t_2), (t_2, t_1)\}$, $\mathbf{net} = \{(t_4, t_3), (t_5, t_3)\}$ are valid enabling, dependency, and NET relations, respectively. Note that we omit the possible (t_3, t_2), (t_4, t_2), and (t_5, t_2) from $\mathbf{net}$ for this example. Figure 2 depicts the initial token marking s; the set of enabled transitions in s is $\{t_1, t_4, t_5\}$. Consider a run of LPOR in s with initial transition t_1, i.e., LPOR$(t_1, s, ())$. As t_2 is disabled in s, no transition is added to the stubborn set in lines 19-21. Supposed that transitions are processed by ascending index, t_4 is added to the stubborn set because *FwdEnableSet*$(t_4) = \{(t_4, \emptyset), (t_3, \{t_4, t_5\}), (t_2, \{t_4, t_5\})\}$, $(t_2, t_1) \in \mathbf{dep}$, and t_4 and t_5 are both not in the stubborn set. However, thanks to the NET optimization t_5 is not added because *FwdEnableSet*$(t_5) = \{(t_5, \emptyset), (t_3, \{t_4, t_5\}), (t_2, \{t_4, t_5\})\}$, t_4 already is the stubborn set, and τ is empty. As a result, LPOR$(t_1, s, ()) = \{t_1, t_4\} \subset$ *enabled*(s).

5) Correctness: The next theorem states that LPOR indeed generates stubborn sets. The proof of the theorem can be found in Appendix II. A sketch of the proof is given below.

Theorem 1: Let (S, T, S_0) be an STS and $\mathbf{ce}$, $\mathbf{dep}$, and $\mathbf{net}$ a can-enabling, dependency, and NET relation, respectively. Then, for all $s \in S$, $t_I \in enabled(s)$, and $\tau \in T^*$ with $\exists s_0 \in S_0 : s_0 \xrightarrow{\tau} s$, LPOR$(t_I, s, \tau)$ is a stubborn set.

Proof sketch.: A key property of LPOR is that, when executed in a state $s = s_0$, every transition t in LPOR(t_I, s, τ) is independent of all transitions $t_1, t_2, \ldots, t_n$ that are in a path starting from s and that are outside LPOR(t_I, s, τ). To show that D1 and D2 hold, consider the paths starting from s_0, as illustrated in Figure 3.

We first show that t is a key transition (D2). Indirectly, assume that t_i for some $1 \leq i \leq n$ can disable t, i.e., $t \notin enabled(s_i)$. Therefore, t must be dependent on t_i, a contradiction by the previous property.

As t is a key transition, $t \in enabled(s_i)$ for every $1 \leq i \leq n$. Let s'_n be a state such that $s_{n-1} \xrightarrow{t_n} s_n \xrightarrow{t} s'_n$. From the above property, t is independent of t_n, so there exists s'_{n-1} such that $s_{n-1} \xrightarrow{t} s'_{n-1} \xrightarrow{t_n} s'_n$. Repeating this rule n times, we obtain a path $s \xrightarrow{t} s' \xrightarrow{t_1} s'_1 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} s'_{n-1} \xrightarrow{t_n} s'_n$, which proves D1. ■

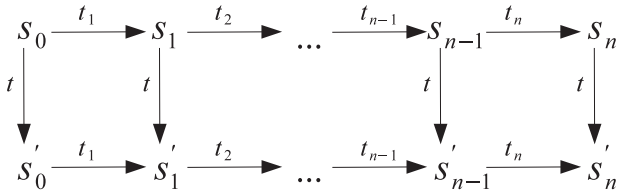

Fig. 3. Illustration of the proof of Theorem 1.

6) Worst-case complexity: Algorithm 2 is guaranteed to terminate (proof in Appendix II) and has worst-case time complexity $O(|T|^3 2^{|T|})$ with and $O(|T|^2)$ without NET optimization. Despite the worst-case exponential overhead of the NET optimization, our experiments show that LPOR with NET can achieve significant reductions of model checking time (Section VI).

We now sketch the idea behind the above complexity results. Assume that checks for set inclusion and adding/removing elements to/from sets take constant time. The basic quadratic time complexity in $|T|$ is due to (1) *Trans* containing at most $|T|$ transitions (line 15), and (2) adding at most $|T|$ transitions to the stubborn set on behalf of every transition in *Trans* (line 18). Note that every transition in *Trans* is also in *Stub* and no transition is ever removed from *Stub*. Therefore, the condition in line 18 and that $enabled(s)$ is fixed throughout an execution of Algorithm 2 guarantee that every transition is added at most once to *Trans*. Without NET optimization, the condition in lines 23-24 is always true. Therefore, no computation overhead is added in this case. With NET optimization, the condition requires to range through possibly each element in a forward enable set and check if this element is in the stubborn set. As elements of the forward enable set are tuples of a transition and a subset of transitions, the maximum size of such a set is $|T|2^{|T|}$.

C. Further Optimizations and Possible Extensions

First, if the dependency relation is transitive, then the enabled transition t_1 does not have to be added to *Trans* (line 21). This is sound because all transitions that would be added to the stubborn set on behalf of t_1 are also added on behalf of t.

LPOR is a non-deterministic algorithm with three main sources of non-determinism, each of them possibly affecting the size of the stubborn set: (1) the selection of the initial transition, (2) the selection of t in line 16, and (3) the order in which **forall** iterates through the transitions in line 18. The tuning of these parameters in such a way that they result in small stubborn sets depends on the analyzed system.

We improve the NET-optimization by making it state-conditional, i.e., t' is a NET for t in a state s if t is not enabled in s and t' must be in any path starting from s before

t can be enabled. The details of this optimization can be found in Appendix I. While state-conditionality can increase the achieved state-reduction, it also increases the time-overhead by limiting the possibilities for pre-computation.

The NET optimization can be generalized to necessary enabling *sets*, i.e., for each transition t a set T' of transitions such that at least one transition in T' must be executed for t to be enabled. This gives more flexibility compared to LPOR where T' contains at most one transition.

LPOR computes *strong* stubborn sets, which implies that all transitions that can disable a transition t in the stubborn set are also included in the set. In general, it is possible that a transition is removed from a strong stubborn set such that the resulting set is stubborn in the weak but not in the strong sense. However, an algorithm computing weak stubborn sets can incur a higher time overhead; in LPOR, this would require to refine dependency in terms of "can disable" and "might not commute" relations.

D. Preserving Temporal Logics with LPOR

The reduced search using stubborn sets preserves all deadlocks of the unreduced state graph. In order to preserve other properties such as invariants or liveness, stubborn sets must satisfy constraints in addition to D1 and D2. LPOR can be configured to preserve a general class of properties written in CTL^*_{-X} (Computational Tree Logic without the next operator) [8], [16]. Subclasses of this logic include simple invariants or LTL_{-X} (Linear Temporal Logic without the next operator). For details of how LPOR can be used to preserve CTL^*_{-X} properties, we refer the reader to Appendix III.

IV. A Case Study: LPOR for Message-Passing

We now briefly introduce a general language for message-passing systems (with detailed formalization in Appendix V) and define suitable LPOR relations (from Section II). The simplicity of these definitions shows that the use of LPOR is indeed straightforward for domain experts.

A. Specifying Message-Passing Systems

A message-passing (MP) system (or protocol) consists of *processes* that communicate via *messages*. Every process maintains a *local state* that is updated by executing local, guarded *transitions* from a set T. A transition t is executed by process $id(t)$ if the *guard* of t evaluates to true; the guard depends on the incoming messages and the local state of the process. The execution of a transition is an atomic event which consumes zero or more messages received by the executing process, changes the local state, and sends multiple messages on behalf of the process. A transition is called a *quorum* transition if it can consume multiple messages. Transitions can be non-deterministic. For example, if a transition can be executed for two different incoming messages, then the first message to be consumed by the transition is non-deterministically selected. The *global state* of the system consists of the local process states and all undelivered messages. An STS corresponding to an MP protocol can be naturally defined such that S, T and S_0 are the sets of states, transitions, and initial states of the MP system, respectively.

So far, the language resembles the usual formalization of message-passing systems [2], [4]. Now, we extend the syntax with some special transitions. Every transition t can be associated with $t.M_I$ (and $t.M_O$), the set of messages possibly received (sent) by t, and $t.I$ (and $t.O$), the set of processes that t can receive (and send) messages from (to). We assume the local state of a process to be an assignment of values to *local variables*. Given a variable x, t is a *write* transition with respect to x and we write $x \in W(t)$ if t can change the value of x in some state. Similarly, t is called a *read* transition ($x \in R(t)$) if the guard of t depends on the value of x. As a special case, a write transition t is an *increment* transition ($x \in Inc(t)$) if t always increases the value of x. Increment transitions are relevant in the context of *timestamp-compare* read transitions t ($x \in CompTS(t)$), a class of transitions common in concurrent systems, e.g., [17]. Such a transition t uses x to store a "timestamp" and compare it with the timestamps of incoming messages. The guard of t can be true only if the timestamp of the message is greater or equal than the current value of x. The sets $R(t)$, $W(t)$, $Inc(t)$, and $CompTS(t)$ can be conservatively determined by lightweight static analysis.

B. LPOR Relations for Message-Passing Systems

1) Can-enable relation: We say that a transition t can *locally* enable another transition t' of the same process if t is a write and t' is a read transition with respect to some common variable x. An exception to this rule is if t is an increment and t' is a timestamp-compare transition with respect to x. In this case t cannot enable t' because a process sends no new message to itself and the timestamp x is increased by t. Formally, can-local-enable = $\{(t,t') \mid id(t) = id(t') \wedge \exists x \in W(t) \cap R(t') : x \notin Inc(t) \cap CompTS(t')\}$, where $id(t)$ denotes the process executing transition t.

A transition t can *remotely* enable a transition t' if it may send messages that can be received by t'. A necessary condition for this to happen is that t and t' are executed by different processes ($id(t) \neq id(t')$), that transition t can send a message to the process executing t' ($id(t') \in t.O$), that transition t' can receive a message from the process executing t ($id(t) \in t'.I$), and that t can send a message that can be received by t' ($t.M_O \cap t'.M_I \neq \emptyset$). Therefore, we define that can-remote-enable = $\{(t,t') \mid id(t) \neq id(t') \wedge id(t') \in t.O \wedge id(t) \in t'.I \wedge t.M_O \cap t'.M_I \neq \emptyset\}$.

Definition 4: Given an MP system, *MP-can-enable* = can-remote-enable $\cup$ can-local-enable.

2) Dependency relation: A transition t' is dependent on t if both are executed by the same process or if t can remotely enable t'. The intuition is that local transitions may change the state of the same process and, if t can remotely enable t', then t can send a message that is processed by t'. Our dependency relation can be refined by excluding pairs of transitions that are executed by the same process and access a disjunct set of variables. This is a refinement that we do not consider in

this paper. Note that the following relation can be asymmetric, which enables LPOR to compute smaller stubborn sets.

Definition 5: Given an MP system, *MP-dependency* $= \{(t,t') \mid t \neq t' \wedge id(t) = id(t')\} \cup$ can-remote-enable.

3) NET relation: The following NET relation is based on the observation that a transition t with $t.I \neq \emptyset$ cannot be enabled unless a process sends a message to process $id(t)$. For example, imagine that t represents a function that requires input from a majority of processes. This implies that $|t.I| = \lceil \frac{n}{2} \rceil$, i.e., a majority of the number of all processes n. Then, t can be enabled only after each of these processes has sent a message to process $id(t)$.

Note that we have to check two additional conditions to make sure that a transition is indeed a NET for t. Firstly, t is required to be *input-deterministic*, i.e., t always consumes a message from every process in $t.I$. Otherwise, t can possibly be enabled even if a process in $t.I$ sends no message to process $id(t)$. Secondly, it is possible that $i \in t.I$ and process i has multiple transitions, say t' and t'', that can enable t (formally, $id(t'') = id(t') \wedge t'' \neq t' \wedge \{(t',t),(t'',t)\} \subseteq$ can-remote-enable). In this case, neither t' nor t'' is necessarily a NET for t.

The NET relation is defined below. In Appendix I, an example is shown of how the content of the channels can be used to make this relation state-conditional.

Definition 6: Given an MP system, *MP-NET* $= \{(t,t') \mid t$ is input-deterministic $\wedge\, id(t') \in t.I \wedge \forall (t'',t) \in$ can-remote-enable: $t'' = t' \vee id(t'') \neq id(t')\}$.

The next theorem states that the above relations are indeed LPOR relations as of Section II-B, a task that must be carried out by the user. The proof of this theorem can be found in Appendix IV.

Theorem 2: Given an MP system, *MP-can-enable*, *MP-dependency* and *MP-NET* are can-enabling, dependency, and NET relations, respectively.

V. Java-LPOR: An LPOR Implementation

We implement LPOR in a Java library, called Java-LPOR. Java-LPOR can be integrated into any explicit state model checker. The LPOR algorithm currently implemented by Java-LPOR computes stubborn sets satisfying D1, D2, and an additional constraint regarding *visible* transitions [8], i.e., transitions that might interfere with the target property. This constraint of visible transitions allows LPOR to preserve *invariants*, i.e., state-local assertions that must hold in every reachable state. The source code of Java-LPOR is available for download[3].

The main steps of integrating Java-LPOR are as follows. As a running example, we show how we used Java-LPOR to implement message-passing LPOR from Section IV.

1) Specifying the transitions: Before the search can start, the transitions of the system must be provided as Java classes. For example, the input language of MP-Basset [5], our model checker for message-passing protocols, is an extension of Java and implements the language from Section IV-A. Within MP-Basset, transitions are represented by the class `TransitionMP`.

2) Implementing the LPOR relations: Java-LPOR exports LPOR's relations via the following interface. This generic interface is parametric in the class `T` of transitions.

```
public interface LPORRelations<T> {
    public boolean dep(T t1,T t2);
    public boolean canEnable(T t1,T t2);
    public boolean net(T t1,T t2);
}
```

For example, the following snippet shows the implementation of our dependency relation for message-passing systems (compare with Definition 5). The method `t1.isLocal(t2)` returns true iff $id(t_1) = id(t_2)$.

```
public boolean dep(TransitionMP t1,TransitionMP t2){
    return !t1.equals(t2) &&
         t1.isLocal(t2) || canRemoteEnable(t1, t2);
}
```

3) Setting up LPOR: For the preservation of invariants, Java-LPOR requires to identify visible transitions. In our current implementation, the user is required to annotate visible transitions using the following interface.

```
public interface VisibilityChecker<T> {
    public boolean isVisible(T t);
}
```

Given the list of all transitions `trans`, the LPOR relations `rel`, and a class `vis` for checking visible transitions, an LPOR utility instance can be created. Its constructor is responsible for pre-computing the forward enable sets. The instance of `LPORUtil` can then be used to compute stubborn sets for a particular state by invoking the `LPOR` method. As arguments, the method requires an initial transition and the list of enabled transitions. Transitions are identified by their index in `trans`.

```
public class LPORUtil<T>{
    public LPORUtil(List<T> trans,
                    LPORRelations<T> rel,
                    VisibilityChecker<T> vis){
           this.trans=trans;
           this.rel=rel;
           this.vis=vis;
           precompute();
    }
    public int[] LPOR(int t_I, int[] enabledTrans){
        ...
    }
    ...
}
```

4) Computing stubborn sets: Finally, the following snippet shows how the set of transitions that must be executed in a state is pruned by a call to the `LPOR` method of an `LPORUtil` instance. This is also how we integrated Java-LPOR into MP-Basset.

```
enabledTrans=lporUtil.LPOR(initTrans, enabledTrans);
```

VI. LPOR Experiments

In this Section, we present our results of using LPOR to model check various fault-tolerant message-passing protocols.

[3]http://www.deeds.informatik.tu-darmstadt.de/peter/Java-LPOR.jar

TABLE I
PERFORMANCE RESULTS OF LPOR IMPLEMENTED WITHIN MP-BASSET USING JAVA-LPOR.

Protocol (# processes)	Res.	Unreduced		DPOR		LPOR								
						Stateless			LPOR only			LPOR + NET		
		States	Time	States	Time	States		Time (on-line)	States		Time (on-line)	States		Time (on-line)
Paxos (6)	OK	>38mil	>192h	3,305,752	22h53m	1,118,341	MJI	6h14m (6h19m)	1,130,234	MJI	6h59m (7h1m)	**548,061**	MJI	**3h18m** (3h21m)
							Mod.	8h51m (28h32m)		Mod.	8h51m (24h10m)		Mod.	4h45m (18h52m)
F-Paxos (6)	CE	238,790	1h34m	**2,028**	**50s**	3489	MJI	1m16s	3489	MJI	1m43s	3415	MJI	1m40s
F-Paxos2 (7)	CE	>16mil	>192h	**21,177**	**12m31s**	175,725	MJI	1h24m	173,414	MJI	1h25m	173,414	MJI	1h28m
Register (5)	OK	287,638	47m	27,763	6m50s	27,763	MJI	5m57s (5m59s)	**18,451**	MJI	**4m32s** (4m32s)	18,451	MJI	4m36s (4m36s)
							Mod.	6m17s (9m23s)		Mod.	5m3s (7m1)		Mod.	4m55 (7m52s)
Register (5)	CE	7,619	1m52	**2,344**	**40s**	4,654	MJI	1m4s	3,497	MJI	55s	3,497	MJI	58s
Register (6)	CE	24,939,222	181h	11,235	3m56s	11,235	MJI	3m37s	**6,987**	MJI	**2m32s**	6,987	MJI	2m34s
Multicast (5)	OK	7,279	1m34s	7,945	1m46s	2,674	MJI	38s (38s)	6,607	MJI	1m29s (1m30s)	**2178**	MJI	**37s** (37s)
							Mod.	1m2s (1m47s)		Mod.	2m7s (2m7s)		Mod.	59s (1m46s)
Multicast (6)	OK	102,058	28m13s	183,265	44m45s	24,382	MJI	6m12s	94,186	MJI	26m26s	**12,494**	MJI	**3m34s**
							Mod.	7m8s (11m2s)		Mod.	29m32s (31m15s)		Mod.	5m4s (9m32)
Multicast (6)	CE	7,543	3m32s	4,890	2m8s	4,890	MJI	1m57s	**2,139**	MJI	**1m4s**	2,139	MJI	1m47s

A. Target Protocols and Properties

We selected the following representative protocols: Paxos [17], a widely-used [24], [26] crash-tolerant consensus protocol, the Byzantine-tolerant Echo Multicast protocol [20], and a crash-tolerant regular storage protocol in the style of [1]. We assume meaningful finite protocol instances where at least one process fault is tolerated.

We consider the main safety properties of these protocols, namely Paxos must not return different values (consensus), Echo Multicast sends the same value to each recipient (agreement), and a read operation returns a value not older than the one written by the latest preceding write operation (regularity). Each of these properties can be expressed by invariants, a class of properties preserved by LPOR. For evaluating the bug-finding capabilities of LPOR, we inject faults into both the protocols and the properties.

A detailed description of these specifications can be found in Appendix VI.

B. Comparison with Dynamic POR

We compare LPOR with dynamic POR (DPOR) [10]. We explain how DPOR differs from static POR (SPOR) in Section VII. In general, the benefit of DPOR is that it needs to be less conservative about the selection of paths that are explored in the reduced search. However, our experiments show the efficiency of LPOR over DPOR, improving on the reductions of a message-passing DPOR implementation.

Like any SPOR algorithm, LPOR can be soundly combined with DPOR for further reduction [10]. This must respect the restrictions imposed by DPOR, however. For example, DPOR assumes the absence of cycles in the state space. We only consider protocol examples with acyclic state spaces for a fair comparison.

We compare LPOR with the original DPOR algorithm by Flanagan and Godefroid [10] because this preserves (with the visibility constraint) the properties of our example protocols. For example, the DPOR variant in [21] only guarantees that every transition executed in the unreduced search is also executed in the reduced one.

In order to preserve invariants, Java-LPOR prevents non-trivial stubborn sets from including visible transitions [23], [8]. This constraint can also be implemented in DPOR such that if a visible transition is executed in a state during the search, then all enabled transitions in this state will be executed. For comparing LPOR with DPOR, we use the Basset model checker [18], which implements an adaptation of Flanagan and Godefroid's DPOR algorithm for actor programs. The actor semantics used in Basset is similar to our model of message-passing except that quorum transitions are not supported. Therefore, we extended Basset's DPOR implementation with quorum transitions: when a process executes a quorum transition, the vector clock of the process will be updated to be the maximum of (1) its current value and (2) the values of the vector clocks of the senders of the messages, where the values correspond to the time of sending the message. In Basset this computation involves one sender as every transition consumes a single message.

C. Experimental Setup

We run our experiments in a DETERlab testbed [29] on 2GHz Xeon machines. We compare LPOR with the unreduced models and DPOR, our extension of Basset's DPOR implementation as explained above. We integrated both this DPOR algorithm and LPOR (as described in Section V) within the MP-Basset model checker [5]. The source of this version of MP-Basset is available online [28]. For fair comparison, both of our POR implementations use the same heuristic for initial transitions. We refer the reader to Appendix VI for details of this heuristic. DPOR is run as stateless search because DPOR can be unsound if state comparison is used [10].

We use three versions of the LPOR algorithm. First, we run the full-fledged algorithm but switch off state comparison (stateless). Second, we run a stateful search but switch off the NET optimization (LPOR only). Third, we run stateful search and LPOR with (state-conditional) NET support (LPOR + NET). We also count the number of visited states in the stateless searches, for both LPOR and DPOR.

D. Our Reduction Results

The results of our experiments are shown in Table I. We write OK if the model checker finds no bug, otherwise (in case of faulty protocols or wrong specifications) a counterexample

(CE) is returned. F-Paxos and F-Paxos2 are two faulty versions of Paxos. We used wrong specifications for the other protocols. The best result for each protocol instance is written in bold. In buggy instances the search is stopped after finding the first bug, i.e., the search is non-exhaustive. Therefore, the number of visited states depends on the order in which transitions are executed in a state. This schedule can be different in DPOR and LPOR.

We observe that:

- The POR-based search finds bugs faster than unreduced search and there is no clear winner between DPOR and LPOR.
- LPOR is highly efficient as shown by the exhaustive search results (OK) reducing the number of states by up to to 94% and search time by up to 90% – see register example.
- Although the additional online checks in the NET optimization slow down LPOR (as discussed in Section III-B), e.g., 74 states/sec versus 59 states/sec for exhaustive Multicast (5), the additional state reduction can add up to reducing the total model checking time. Indeed, the NET optimization can be very efficient by achieving additional space and time reductions of up to 87% – see Multicast (6) exhaustive search result.
- LPOR outperforms DPOR in *all* exhaustive search experiments, even in stateless search where the benefit of LPOR is not biased by the stateful optimization. In addition, LPOR proves to be more time efficient than DPOR, i.e., the time overhead of LPOR is smaller. For example, the stateless exhaustive runs of Register (5) visit the same number of states but LPOR is faster.

E. Execution Time Issues

In this Section, we discuss the trade-offs affecting the time overhead of LPOR as implemented within MP-Basset.

MP-Basset is an extension of Basset [18], a model checker for actor programs. Basset, in turn, builds on Java Pathfinder (JPF) [27], a stateful model checker for Java. Similarly to Basset, MP-Basset is a Java application run by JPF. As such, it can run Java code at two levels [27]: first, in the modeled layer, which is a JPF-simulated JVM; second, in the host JVM (where JPF also runs), which is accessible from the modeled layer via an interface called Model Java Interface (MJI). Roughly speaking, JPF explores the state space of the application run in the modeled layer. Due to the indirection of the modeled layer, execution in this layer is slower than in the host JVM. The modeled application can always execute code in the host JVM using MJI. However, as there is a speed penalty of using MJI, time efficient JPF applications should use MJI with care. One source of this time overhead is that MJI converts parameters of MJI method calls between the modeled and the host JVM's object model.

To explore this trade-off, we created and compared two architectures, one where the LPOR algorithm runs in the modeled layer and another one where it runs in the host JVM. In our experiments, the MJI-based implementation was faster. This meets our expectations for (state-unconditional) "LPOR only" because no state information is passed (and thus converted) to Java-LPOR, whereas in (state-conditional) "LPOR + NET", the NET relation is a function of a small fraction of the current state (see Section IV-B). For our message-passing instantiation of LPOR, the MJI overhead turns out to be more time efficient than executing the LPOR algorithm in the modeled layer even in the "LPOR + NET" case. This does not necessarily generalize. In other LPOR applications, particularly where the entire state has to be converted for MJI, the execution time penalties may trade off differently.

Table I shows the model checking time of both implementations (MJI and Mod. stands for the implementation in the modeled and the host JVM layer, respectively). For space reasons, we omit the modeled layer times for the CE results as they show similar trends as for OK.

We also measure the benefit of using pre-computation. The times where forward enable sets are computed on-line (no pre-computation) are written in parentheses. Otherwise, the times shown include the time of pre-computation. The benefit of pre-computation is significant in the modeled layer implementation. We observe a higher relative gain of using pre-computation in NET optimized LPOR. The reason is that forward enable sets containing non-empty *en*-fields (in the NET optimized case) tend to be larger, thus, their computation takes longer. The reason why the MJI implementation does not greatly benefit from pre-computation for our particular protocol examples is two-fold: first, lines 22-26 in LPOR (Algorithm 2) are executed in a relative small number of states; second, the body of the do-while loop in the forward enable set computation (Algorithm 1) is executed only a few (1-2) times during an average invocation of *FwdEnableSet*. We leave the investigation of other protocols, which could very well show a completely different profile, for future work.

VII. Related Work

The basic structure of the LPOR algorithm is similar to Godefroid's stubborn (and persistent) set algorithms [11], which start with a transition and keep adding new transitions using the dependency and can enabling relations until the current set of transitions is not stubborn. An application of these algorithms to new languages is only possible after a translation into a specific language used in [11] that specifies processes communicating via shared objects. Transitions in this language are assumed to be deterministic. Furthermore, the algorithms in [11] do not support pre-computation. The ample set algorithms in [8], [14], [13] also restrict to process-based systems and deterministic transitions. Moreover, they conservatively assume that a non-trivial ample set consists of all enabled transitions of a particular process.

Promela is a general language with explicit support for multi-process systems and message-passing. SPIN is a widely-used model checker for specifications written in Promela [13]. SPIN supports a specific form of POR, which is based on the observation that transitions t_1 and t_2 are independent if they are from different processes and t_1 is the only transition

writing to (or reading from) a FIFO channel (exclusive write or read, respectively) [14], [13]. Such interferences can be easily expressed in LPOR by excluding (t_1, t_2) and (t_2, t_1) from the dependency relation. We note that in the description of [14], t_1 and t_2 are considered "independent" only in states where the channel is non-empty (non-full). This is because their definition of dependency includes that a transition can enable another transition. In fact, t_1 can enable read (send) transitions but t_1 and t_2 are always (state-unconditionally) independent in the sense of Definition 2.

It is possible to give a graph theoretic implementation of LPOR as proposed in [23]. In this approach, the vertices of the graph are transitions and t is connected to t_1 if t_1 needs to be added to the stubborn set on behalf of t. Then, certain vertices of this graph, e.g., included in properly selected strongly connected components, correspond to stubborn sets.

Dynamic POR (DPOR) [10] is a POR implementation which computes a persistent set in some state s gradually while the successors of s are explored. In this way the persistent set algorithm can learn about interfering transitions and needs not to guess them as in static POR. In other words, DPOR explores future paths instead of guessing them. However, DPOR also makes static assumptions about co-enabled dependent transitions. Furthermore, DPOR is inherently a depth-first search, it needs to know the sequence of transitions in the current path (which is not straightforward in parallel model checking [22]) and can be unsound with stateful model checking [25].

In recent work [5], we propose a heuristic to translate from one transition system to another to maximize the reduction of POR and apply it to message-passing systems. This translation is orthogonal to LPOR, which requires a transition system at its input.

The input relations of LPOR can be partly or entirely derived automatically using a SAT solver, an approach similar to [7]. Moreover, SAT-based bounded model checking can be used to compute more accurate enabling sequences than our forward enable sets. For example, given transitions t_1, t_2, t_3, it is possible that t_1 can enable t_2, and t_2 can enable t_3, but t_2 cannot enable t_3 if t_2 was enabled by t_1.

VIII. Conclusions

We have proposed LPOR, a framework for easy-to-use, flexible, and efficient POR implementations. While existing POR implementations trade flexibility for ease-of-use and efficiency, e.g., SPIN's POR limits to exclusive write/read FIFOs or DPOR prohibits cycles, the strength of LPOR is that it provides these features at the same time. In ongoing work, we study if state-conditional can-enabling and dependency relations can improve on LPOR's reductions. For example, a state-conditional can-enabling relation can be used to rule out transitions t_1 in line 22 of Algorithm 2 that cannot enable any transition in the current state. Another possible extension is to add symmetry reduction to LPOR. Although PO and symmetry reductions are compatible in theory [9], no implementation of their combination is available nor its efficiency was tested on real examples.

Acknowledgement. We thank Gerard Holzmann for his insights of the POR theory as implemented by SPIN to enable an objective comparison across LPOR and SPIN.

References

[1] H. Attiya, A. Bar-Noy, D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995.
[2] H. Attiya, J. Welch. *Distributed Computing*. John Wiley and Sons, 2004.
[3] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, S. Rajamani. Partial-Order Reduction in Symbolic State-Space Exploration. *FMSD*, 18(2): 97–116, 2001.
[4] P. Bokor, M. Serafini, N. Suri. On Efficient Models for Model Checking Message-Passing Distributed Protocols. *FORTE*, pp. 216–223, 2010.
[5] P. Bokor, J. Kinder, M. Serafini, N. Suri. Efficient Model Checking of Fault-Tolerant Distributed Protocols. *DSN-DCCS*, pp. 73–84, 2011.
[6] P. Bokor, J. Kinder, M. Serafini, N. Suri. Supporting Domain-Specific State Space Reductions through Local Partial-Order Reduction. Technical Report, Technische Universität Darmstadt, TR-TUD-DEEDS-07-01-2011, 2011. (http://www.deeds.informatik.tu-darmstadt.de/peter/papers/LPOR.pdf)
[7] R. Bhattacharya, S. German, G. Gopalakrishnan. Exploiting Symmetry and Transactions for Partial Order Reduction of Rule Based Specifications. *SPIN*, pp. 252-270, 2006.
[8] E. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 2000.
[9] E. Emerson, S. Jha, D. Peled. Combining Partial Order and Symmetry Reductions. *TACAS*, pp. 19–34, 1997.
[10] C. Flanagan, P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. *POPL*, pp. 110–121, 2005.
[11] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.
[12] G. Gueta, C. Flanagan, E. Yahav, M. Sagiv. Cartesian Partial-Order Reduction. *SPIN*, pp. 95–112, 2007.
[13] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
[14] G. J. Holzmann, D. Peled. An Improvement in Formal Verification. *FORTE*, pp. 197–211, 1994.
[15] V. Kahlon, C. Wang, A. Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. *CAV*, pp. 398–413, 2009.
[16] L. M. Kristensen, K. Schmidt, A. Valmari. Question-guided Stubborn Sets for State Properties. *FMSD*, 29(3):215-251, 2006.
[17] L. Lamport. The Part-time Parliament. *ACM Trans. Comp. Sys.*, 16(2):133–169, 1998.
[18] S. Lauterburg, M. Dotta, D. Marinov, G. Agha. A Framework for State-Space Exploration of Java-Based Actor Programs. *ASE*, pp. 468–479, 2009.
[19] R. Nalumasu, G. Gopalakrishnan. A New Partial Order Reduction Algorithm for Concurrent System Verification. *CHDL*, pp. 305-314, 1997.
[20] M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. *CCS*, pp. 68–80, 1994.
[21] K. Sen, G. Agha. Automated Systematic Testing of Open Distributed Programs. *FASE*, pp. 339–356, 2006.
[22] U. Stern, D.L. Dill. Parallelizing the Murφ Verifier. *FMSD*, 18(2): 117–129, 2001.
[23] A. Valmari. The State Explosion Problem. *Petri Nets I: Basic Models*, pp. 429-528, 1998.
[24] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, L. Zhou. MODIST: Transparent MC of Unmodified Distributed Systems. *NSDI*, pp. 213–228, 2009.
[25] Y. Yang, X. Chen, G. Gopalakrishnan, R.M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. *SPIN*, pp. 288–305, 2008.
[26] http://hadoop.apache.org/zookeeper/
[27] http://babelfish.arc.nasa.gov/trac/jpf
[28] http://www.deeds.informatik.tu-darmstadt.de/peter/mp-basset/
[29] http://www.isi.deterlab.net/