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  
joehannes.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.

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). 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 \(s_0 \xrightarrow{t_1 \ldots t_n} s_n\) 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 \(s_0 \in S\), then we say that \(s_n\) is reachable. A transitions \(t\) is said to be in a path \(s_0 \xrightarrow{t_1 \ldots t_n} s_n\) 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

---

1In 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]. 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 $ce \subseteq T \times T$ is can-enabling iff $ce \supseteq \{(t, t') | \exists s, s' \in S : s \xrightarrow{t} s' \land t' \not\in enabled(s) \land t' \in enabled(s')\}$. 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 $dep \subseteq T \times T$ is a dependency relation, iff $\text{dep} \supseteq \{(t, t') | t \neq t' \land \exists s, s' \in S : t, t' \in enabled(s) \land s \xrightarrow{t} s' \land \text{either (a) } t' \not\in enabled(s') \lor \text{(b) } \exists s'' \in S : s \xrightarrow{t} s'' \land s \not\in enabled(s'')\}$. 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 (non-interfering) transitions in can-enabling and dependency relations.

**Definition 3:** A relation $net \subseteq T \times T$ is a necessary enabling transition (NET) relation, iff $\text{net} \subseteq \{(t, t') | \forall s_0 \in S_0, \forall s \in S, \forall t_1, \ldots, t_n \in T : \text{if } s_0 \xrightarrow{t_1t_2\ldots t_n} s \land t \in enabled(s), \text{then } t' = t_i \text{ for some } 1 \leq i \leq n\}$. 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-$\neg$X and CTL-$\neg$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

²Appendices are included in the technical report version of this paper available online [6].
These two functions to compute stubborn sets. The stubborn set algorithm (Algorithm 2) without the NET optimization where

We therefore start out by explaining the LPOR algorithm B. The Stubborn Set Algorithm

\[ T \leftarrow \{(t, \emptyset)\} \]

\[ T \leftarrow T' \]

\[ \text{for all } t \in T \text{ do} \]

\[ \text{if } (t, t') \notin \text{stubborn set} \text{ then return true;} \]

\[ \text{return false.} \]

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

\[
\begin{align*}
\text{Stub} & \leftarrow \{(t)\}; \\
\text{Trans} & \leftarrow \{t\}; \\
\text{while } \text{Trans} \neq \emptyset \text{ do} \\
& \quad \text{choose } t \in \text{Trans}; \\
& \quad \text{Trans} \leftarrow \text{Trans} \setminus \{t\}; \\
& \quad \text{forall } t \in \text{enabled}(s) \setminus \text{Stub} \text{ do} \\
& \quad \quad \text{if } (t, t) \notin \text{deep then} \\
& \quad \quad \quad \text{Stub} \leftarrow \text{Stub} \cup \{(t)\}; \\
& \quad \quad \text{else if } \text{FwdEnableSetIdx}(t, t) \text{ then} \\
& \quad \quad \quad \text{if } \exists (t_{\text{dep}}, e) \in \text{FwdEnableSet}(t) : (t_{\text{dep}}, t) \in \text{dep} \\
& \quad \quad \quad \quad \land (e = 0 \lor \forall t' \in e : (t' \notin \text{Stub} \land t' \in T)) \text{ then} \\
& \quad \quad \quad \quad \text{Stub} \leftarrow \text{Stub} \cup \{(t)\}; \\
& \quad \quad \quad \text{Trans} \leftarrow \text{Trans} \cup \{(t)\}; \\
& \quad \text{return } \text{Stub};
\end{align*}
\]

Algorithm 2: The LPOR \((t_1, s, \tau)\) stubborn set algorithm for a state \( s \in S \), an initial transition \( t_1 \in \text{enabled}(s) \), 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 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 \( ce \). 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 \( ce, \text{dep}, \text{and net} \), LPOR has three parameters: (1) a transition \( t_1 \in \text{enabled}(s) \), 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 \( \tau \) is a set containing \( t_1, \ldots, t_n \)). From D2, no stubborn set in \( s \) can be empty unless \( \text{enabled}(s) = \emptyset \). 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: \( \text{Stub} \), which represents the stubborn set (line 13) and \( \text{Trans} \), which contains a transition \( t \) in \( \text{Stub} \) such that new transitions might be added to \( \text{Stub} \) on behalf of \( t \) (line 24). Therefore, LPOR adds transitions to \( \text{Stub} \) until \( \text{Trans} \) is empty (lines 15-26) and \( \text{Stub} \) is returned (line 27). We now explain how transitions are added on behalf of a transition \( t \) in \( \text{Trans} \).

First, we add those enabled transitions \( t_1 \) that \( t \) is dependent on (lines 19-21). We add \( t_3 \) if either \( t_1 \) and \( t \) do not commute (disallowed by D1) or it can disable \( t \) (which can violate D2). Note that \( \text{dep} \) 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 \( \text{Trans} \) (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 \( t_{\text{dep}} \), we can verify, given the current state \( s \), that “\( t_2 \) has not been executed yet”. Assume that \( (t_{\text{dep}}, en) \) 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 \( \tau \) (line 24). Note that, for some transition \( t \), \( (t, en) \) can be in a forward enable set multiple
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, ce = \{(t3, t2), (t4, t3), (t5, t3)\}, dep = \{(t1, t2), (t2, t1)\}, net = \{(t4, t5), (t5, t3)\} are valid enabling, dependency, and NET relations, respectively. Note that we omit the possible \(n = n(t_3, t_2, t_4, t_5)\) transitions starting from \(s\) and that \(D1\) and \(D2\) hold, consider the paths starting from \(s\) with initial transition \(t\) executed in a state \(s\). Let \(s\) be a state such that \(s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_n\) is independent of all transitions \(i \leq i \leq n\). Therefore, the condition \(\exists s_0 \in S_0 : s_0 \rightarrow s, \text{LPOR}(t, s, \tau)\) 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 \(\text{LPOR}(t, s, \tau)\) is independent of all transitions \(t_1, t_2, ..., t_n\) that are in a path starting from \(s\) and that are outside \(\text{LPOR}(t, s, \tau)\). 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 \(t_1\) for some \(1 \leq i \leq n\) can disable \(t\), i.e., \(t \notin \text{enabled}(s)\). Therefore, \(t\) must be dependent on \(t_i\), a contradiction by the previous property.

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

6) Worst-case complexity: Algorithm 2 is guaranteed to terminate (proof in Appendix II) and has worst-case time complexity \(O(|T|^2|T|)\) 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) \(\text{Trans}\) containing at most \(|T|\) transitions (line 15), and (2) adding at most \(|T|\) transitions to the stubborn set on behalf of every transition in \(\text{Trans}\) (line 18). Note that every transition in \(\text{Trans}\) is also in \(\text{Stub}\) and no transition is ever removed from \(\text{Stub}\). Therefore, the condition in line 18 and that \(\text{enabled}(s)\) is fixed throughout an execution of Algorithm 2 guarantee that every transition is added at most once to \(\text{Trans}\). 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 \(\text{Trans}\) (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 \(\text{foralls}\) 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

![Fig. 2. A Petri net example.](image)

![Fig. 3. Illustration of the proof of Theorem 1.](image)
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 $t \in CompTS(t)$, 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 $= \{(t, t') | id(t) = id(t') \wedge \exists x \in W(t) \cap R(t') : x \notin Inc(t) \cap CompTS(t')\}$, 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 $= \{(t, t') | id(t) \neq id(t') \wedge id(t') \in t.O \wedge id(t) \in t'.I \wedge t.M_O \cap t'.M_I \neq \emptyset\}$.

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

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') | t ≠ t' ∧ id(t) = id(t') \} ∪ can-remote-enable.

3) **NET relation:** The following NET relation is based on the observation that a transition t with t.I ≠ 0 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| ≥ \binom{n}{\lceil \frac{n}{2} \rceil} + 1, 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, \( id(t'') = id(t' ) ∧ t'' \neq t' ∧ \{ (t', t), (t'', t) \} \subseteq \text{can-remote-enable} \)). 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 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') | t is input-deterministic ∧ id(t') \in t.I ∧ \forall (t'', t) \in \text{can-remote-enable}: t'' = t' ∨ 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 download3.

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.

```java
public interface LPORRelations<T> {
    public boolean dep(TransitionMP 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 \( id(t1) = id(t2) \).

```java
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.

```java
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.

```java
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.

```java
enabledTrans=lpорUtil.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.
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.
In our experiments, the MJI-based implementation was faster. Modeled layer and another one where it runs in the host JVM. Architectures, one where the LPOR algorithm runs in the modeled layer, execution in this layer is slower than in the host JVM (where JPF also runs), which is accessible from the modeled layer, which is a JPF-simulated JVM; second, in the host JVM, it can run Java code at two levels [27]: first, in the modeled layer, 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 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

[27] http://babelfish.arc.nasa.gov/trac/jpf
[29] http://www.isi.deterlab.net/
Algorithm 5: Generalized $FwdEnableSet(t)$ and $FwdEnableSetIdx(t, t')$ are pre-computed for every $t, t' \in T$ using the can-enabling relation $ce$, the dependency relation $dep$, and the NET relation $net$. Generalized LPOR uses a modified forward enable set function (Algorithm 3) which is different from Algorithm 1 in that it stores pairs of transitions in the en-fields (line 37). The reason why we store $(t_1, t_2)$ if $(t_1, t_2)$ is in a NET relation (and not only $t_2$ as in Algorithm 1) is that we can verify in the current state whether or not $t_2$ must be executed in future paths before $t_1$ can be enabled (as opposed to the simple check of Algorithm 1 if $t_2$ was ever executed in the current path). Assume that $(t_{dep}, en)$ is in the forward enable set of $t_1$ and the conditions in lines 49-50 are true. Using the notations of Algorithm 4, we only add $t_1$ to the stubborn set if either $t_3$ is not already in the stubborn set or $t_3$ needs not necessarily be executed for $t_2$, and thus, $t_{dep}$ to be enabled (line 51). The later condition is expressed by a relation called NET-transition-to-fire relation (state-conditional NET).

Definition 7: A relation $nttf \subseteq S \times T \times T$ is NET-transition-to-fire ($nttf$) if for all $t, t' \in T$, $s \in S$, if $(s, t, t') \in nttf$, then $t \notin enabled(s)$ and $t'$ is in $\sigma$ for all paths $\sigma$ from $s$ to some $s' \in S$ such that $t \in enabled(s')$.

The correctness of generalized LPOR (Algorithm 4) does not depend on the NET relation (state-conditional NET) used in Algorithm 3. The purpose of this relation is to “guess” (before model checking) those transitions that can potentially be subject to NET optimization, and $nttf$ is used to “verify” (during model checking) the soundness of NET. For example, a possible heuristic was shown in Section III, where NET contains those pairs $(t, t')$ transitions where $t$ can only be enabled if $t'$ is executed at least once before $t$. In this case, it is indeed correct that $(s, t, t') \in nttf$ iff $t'$ is not in the current search path $\tau$. It is possible in general, however, that $t'$ is in $\tau$ but it must be executed at least once more for $t$ to be enabled.

We now show a message-passing example that utilizes the Nttf-optimization of generalized LPOR. Based on MP-NET we can define a NET-transition-to-fire relation. If $t'$ is a NET for $t$, and in some state $s t'$ has not yet sent a message needed for $t$ to be enabled, then $t'$ must be in any path from $s$ to a state where $t$ is enabled. Note that it is possible that $t'$ is in $\tau$. This can happen if there is a message $m$ in the channel from the process executing $t'$ to the process executing $t$ such that $m \notin t.M_t$. Intuitively, $t'$ can enable $t$ because $t'.M_t \cap t.M_t \neq \emptyset$ but $t'$ can also enable other transitions and $m$ is not meant for $t$.

In Appendix IV we prove that the following relation is indeed an Nttf relation.

**Definition 8:** MP-NET-transition-to-fire $\subseteq S \times T \times T$ is a relation such that $(s, t, t') \in$ MP-NET-transition-to-fire iff $(t, t') \in$ MP-NET $\land (s(c_{en(t)}, id(t)) \cap t.M_t = \emptyset)$.

Theorem 1 is a special case of the following theorem.

**Theorem 3:** Let $(S, T, S_0)$ be an STS and $net \subseteq T \times T$ an arbitrary relation, and $ce$, $dep$, and $nttf$ a can-enabling, dependency, and NET-transition-to-fire relation, respectively. Then, for all $s \in S, t \in enabled(s)$, LPOR($t, s$) is a stubborn set.

**APPENDIX**

**I. GENERALIZED LPOR**

Algorithm 4, called generalized LPOR, is an LPOR-like algorithm that is optimized for better space-reductions using state-conditional NET relations.

Generalized LPOR uses a modified forward enable set function (Algorithm 3) which is different from Algorithm 1 in that it stores pairs of transitions in the $en$-fields (line 37). The reason why we store $(t_1, t_2)$ if $(t_1, t_2)$ is in a NET relation (and not only $t_2$ as in Algorithm 1) is that we can verify in the current state whether or not $t_2$ must be executed in future paths before $t_1$ can be enabled (as opposed to the simple check of Algorithm 1 if $t_2$ was ever executed in the current path). Assume that $(t_{dep}, en)$ is in the forward enable set of $t_1$ and the conditions in lines 49-50 are true. Using the notations of Algorithm 4, we only add $t_1$ to the stubborn set if either $t_3$ is not already in the stubborn set or $t_3$ needs not necessarily be executed for $t_2$, and thus, $t_{dep}$ to be enabled (line 51). The later condition is expressed by a relation called NET-transition-to-fire relation (state-conditional NET)

**II. GENERALIZED LPOR PROOF**

First we prove that Algorithm 4 terminates.

**Lemma 1:** Given the notation of Theorem 3, LPOR($t, s$) terminates for all $s \in S, t \in T$. In addition, the worst-case time complexity of LPOR($t, s$) is $O(|T|^2|T'|^2)$ (and $O(|T|^2)$) using (not using) the Nttf-optimization.

**Proof:** LPOR($t, s$) terminates if Trans is empty (line 42). Assume that LPOR($t, s$) runs forever, i.e., Trans is never empty at line 42. In every iteration of the while-loop (lines 42-53) exactly one transition is removed from Trans (line 44) and zero or more transitions are added to it (lines 48 and 53). Since the set of all transition ($T$) is finite, there must be a transition $t'$ that is added multiple times to Trans. However, since $t'$ is also added to Stub (line 47 and 52), no transitions are ever removed from Stub, and $t'$ can only be added to Trans if $t' \notin Stub$ (lines 46 and 49), $t'$ cannot be added to Trans multiple times, a contradiction.

Consequently, the body of the while-loop (lines 42-53) can be executed at most $|T|$ times. Within the body, transition
Given the notation of Theorem 3, for all $s \in S, t, t_1, \ldots, t_n \in T$ such that $t \in \text{LPOR}(t', s)$, $t_i \notin \text{LPOR}(t', s)$ for all $1 \leq i \leq n$, and $s \xrightarrow{t_1 \ldots t_n} s'$, it holds that $(i, t) \notin \text{dep}$.

Proof: Indirectly, assume that $(t_i, t) \in \text{dep}$ for some $1 \leq i \leq n$. If $t_i \in \text{enabled}(s)$, then $t_i$ is added to $\text{LPOR}(t', s)$ in lines 46-48. Otherwise, Lemma 2 implies that there is $1 \leq j < i$ such that $t_j \in \text{LPOR}(t', s)$, a contradiction.

Now we prove that generalized LPOR computes strong stubborn sets, which implies Theorem 3. In turn, Theorem 3 implies Theorem 1.

Theorem 4: Given the notation of Theorem 3, for all $s \in S, t' \in T$, $\text{LPOR}(t', s)$ is a strong stubborn set.

Proof: Given $s_1, s_2, \ldots, s_n \in S$, and $t_1, t_2, \ldots, t_n \in T$, let $s = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} s_n$ be a path such that $t_1, t_2, \ldots, t_n \notin \text{LPOR}(t', s)$. We first show that every $t \in \text{LPOR}(t', s)$ is a key transition (D2). Indirectly, assume that $t_i$ for some $1 \leq i \leq n$ can disable $t$, i.e., $t \notin \text{enabled}(s_i)$. Therefore, $(t_i, t) \in \text{dep}$ must hold, a contradiction by Corollary 1.

Next, we show that $\text{D1}$ holds. We know that every $t \in \text{LPOR}(t', s)$ is a key transition. Therefore, $t \in \text{enabled}(s_i)$ for every $1 \leq i \leq n$. Let $s'_n$ be a state such that $s_{n-1} \xrightarrow{t_{n-1}} s'_n$. From Corollary 1, $(t_n, t) \notin \text{dep}$, so there exists $s'_n$ such that $s_{n-1} \xrightarrow{t_{n-1}} s'_n \xrightarrow{t_n} s'_n$. As illustrated in Figure 3, repeating this rule $n$ times, we obtain a path $s \xrightarrow{t_1 \cdots t_n} s'_1 \xrightarrow{t_2 \cdots t_n} \cdots \xrightarrow{t_{n-1} \cdots t_n} s'_n$. Now we prove that stubborn sets generated by LPOR also satisfy D3 if the $\text{dep}$ relation used in LPOR is symmetric [23].

Theorem 5: Given the notation of Theorem 3, if $\text{dep}$ is symmetric, then for all $s \in S, t' \in T$, $\text{LPOR}(t', s)$ satisfies D3, i.e., for all $t, t_1, t_2, \ldots \in T$ such that $t \in \text{LPOR}(t', s)$ is a key transition, $t_1, t_2, \ldots \notin \text{LPOR}(t', s)$, and $s \xrightarrow{t_1 \ldots t_2} s$ is an infinite path, there is an infinite path $s \xrightarrow{t_1 t_2}$. 

Next, the conditions in lines 49-50 hold because either $(t_2, t_1) \in \text{dep}$ (if $t_1$ is added to $\text{Trans}$) or $(t_2, t'_1) \in \text{dep}$ (if $t'_1$ is added to $\text{Trans}$). The later is true because $(t_2, t_1) \in \text{dep}$ and $(t_1, t'_1) \in \text{dep}$ and $\text{Trans}$ is transitive. The set $en$ cannot be empty because $t'_k \notin \text{Stub}$ (line 51, first disjunct). Therefore, let $(t', t'') \in \text{en}$ such that $t'' \in \text{Stub}$ and $(s, t', t'') \in \text{nttf}$. Again, if no such $(t', t'')$ exists, then $t'_k$ is added to $\text{Stub}$, a contradiction. We now show that $t''$ is among $t'_1, t'_2, \ldots, t'_n$, a contradiction. First, $t'$ is among the transitions in $ES$. This is because $(t_2, en)$ is stored along $ES$ and every tuple in $\text{FwdEnableSet}(t'_k)$ is written only once when added to the forward enable set (line 37). Say that $t' = t'_i$ for some $1 \leq i \leq n$. From $(s, t', t'') \in \text{nttf}$ we know that $t''$ must be executed before $t'$ can be enabled. Since we know that $t'_i \in \text{enabled}(s_{i-1})$, $t''$ must be among $t'_1, \ldots, t'_{i-1}$, the final contradiction.

A simple consequence of Lemma 2 is the following Corollary:

Corollary 1: Given the notation of Theorem 3, for all $s \in S, t, t'_1, \ldots, t_n \in T$ such that $t \in \text{LPOR}(t', s)$, $t_i \notin \text{LPOR}(t', s)$ for all $1 \leq i \leq n$, and $s \xrightarrow{t_1 \ldots t_n} s'$, it holds that $(i, t) \notin \text{dep}$.
Proof: Indirectly, assume that there is $n > 0$ such that for all $s_n \in S$ where $s \xrightarrow{t_1 \ldots t_n} s_n$ it holds that $t_{n+1} \not\in enabled(s_n)$. As $t$ is a key transition (note that every transition in LPOR ($t', s$) is a key transition), $t$ is enabled in $s_n \in S$ where $s \xrightarrow{t_1 \ldots t_n} s_n$ (illustrated in Figure 4). Let $s_n' \in S$ be any state such that $s_n \xrightarrow{t} s_n'$. From Corollary 1, we know that $(t_{n+1}, t) \not\in dep$. In addition, as $dep$ is symmetric, it also holds that $(t, t_{n+1}) \not\in dep$. Therefore, $t$ cannot disable $t_{n+1} + 1$ and we have $t_{n+1} \in enabled(s_n')$. Finally, Theorem 4 (D1) implies that there is a path $s \xrightarrow{t_1 \ldots t_{n+1}} s_n' \xrightarrow{t_{n+1}+1}$, a contradiction.

III. PRESERVING TEMPORAL PROPERTIES WITH LPOR

D1 and D2 does not even suffice to preserve invariants, a simple but useful property in program analysis. For the preservation of invariants, stubborn sets must satisfy proviso and visibility, which can be implemented independently of the semantics of the transitions [11], [19], [23], [8], [16]. Proviso solves the ignorance problem and visibility guarantees that states missed in the reduced state graph do not interfere with the property.

A. Preserving LTL_{¬X} with LPOR

If transitions are deterministic and the stubborn sets satisfy proviso and visibility, then the reduced graph preserves any property expressible in LTL_{¬X} (Linear Temporal Logic without the next time operator). If transitions can be non-deterministic, then an additional condition called D3 must be satisfied [23]. Intuitively, D3 guarantees that infinite paths are preserved by POR. If transitions are deterministic, then D1 and D2 imply D3. However, D3 has to be separately established if transitions can be non-deterministic: let $s = s \xrightarrow{t_1 \ldots t_n} s_n$ be an infinite path in the unreduced graph and $t$ a key transition in $s$ in a stubborn set $stub(s)$. Assume that $t_1, t_2, \ldots$ are outside $stub(s)$. Since $t$ is a key transition, there is a run $s \xrightarrow{t_1 \ldots t_n} s_n$ for every $n > 0$. It is possible that although for every $n > 0$ there is a path $s_n = s \xrightarrow{t_1 \ldots t_n} s_n$ (from D1), every such $s_n$ proceeds through different states depending on $n$. Therefore, a cycle in $s$ may not correspond to a cycle in $s_n$ and, thus, there exists no infinite path $s \xrightarrow{t_1 \ldots t_n}$. If the dependency relation is symmetric, LPOR returns stubborn sets which satisfy D3 (the proof of this property can be found in Appendix II). The key to this result is the following property of LPOR (which is also used to prove D1 and D2) – we call this property commutativity: for all transitions in $\sigma_n$ it holds that $(t_i, t) \not\in dep$, i.e., $t$ is independent of $t_i$. If $dep$ is symmetric, then $t_i$ is also independent of $t$, which implies that $t$ cannot disable $t_i$. As a result, $t_{n+1} + 1$ is enabled in $s_n$ and there is $\sigma_{n+1} = s \xrightarrow{t_1 \ldots t_n} s_n \xrightarrow{t_{n+1}+1} s_{n+1}$. Now, $\sigma_n$ and $\sigma_{n+1}$ proceed through the same prefix of states resulting in the preservation of infinite paths.

B. Preserving CTL_{¬X} with LPOR

For the preservation of CTL_{¬X} (Computational Tree Logic without the next operator) a restrictive condition called NB is needed [23]. If all transitions are deterministic, NB requires that non-trivial stubborn sets contain exactly one “invisible” transition. In case of non-deterministic transitions, NB additionally requires that transitions in such stubborn sets are super-deterministic. Informally, a super-deterministic transition is deterministic, cannot be disabled by other transitions, and commutes with the execution of other transitions. The commutativity property of LPOR directly implies that stubborn sets computed by LPOR that contain a single deterministic and invisible transition satisfy NB. As a result, if not all transitions are non-deterministic, then LPOR can achieve reduction whilst preserving properties written in CTL_{¬X}.

IV. PROOFS OF LPOR RELATIONS FOR MESSAGE-PASSING

Lemma 3: Given any MP protocol, MP-can-enable is a can-enabling relation.

Proof: The proof is indirect. Assume that there are $s, s' \in S, t, t' \in T$ such that $t' \not\in enabled(s)$ and $s \xrightarrow{t} s'$ and $t' \in enabled(s')$ and $(t, t') \not\in MP-can-enable$. First, assume that $t$ and $t'$ are local. Let $i$ be the process executing $t$ (and $t'$). From $(t, t') \not\in MP-can-enable$ we have that either $C = \emptyset$ or $\forall x \in C : x \in Inc(t) \cap CompTS(t')$. From $t' \not\in enabled(s)$ we know that $\forall t \in Inc(t) \cap CompTS(t')$. Therefore, there is no $X \subseteq \cup_{i \in \mathbb{N}} ps(c_{j,i})$ such that $X \not\subseteq \cup_{i \in \mathbb{N}} ps(c_{j,i})$ because process $i$ sends no message to itself (it can consume messages through $\forall x \in C : x \in Inc(t) \cap CompTS(t')$. From this we know that $x_1, \ldots, x_k \in W(t)$, and, from $C = \emptyset$, $x_1, \ldots, x_k \not\in R(t')$. Now, let $s_1$ be a state which equals $s$ except that $s_1.x_{i_1} = s'.x_{i_1}$. From $x_1 \not\in R(t')$, $t'$ is disabled in $s_1$. Similarly, let $s_2$ be a state which equals $s_1$ except that $s_2.x_{i_1} = s', x_{i_2} \not\in R(t')$, $t'$ cannot be enabled in $s_2$. After repeating this rule $k$ times, we have that $s_k$ equals $s_k \xrightarrow{X_{i_k}} x_{i_k}$ and $t'$ is disabled in $s_k$, a contradiction. Therefore, we assume for all $X_{i_k}, \ldots, x_{i_k} \in C$ that $x_{i_k} \in Inc(t) \cap CompTS(t')$ ($1 \leq k \leq k$). Let $s_1$ be a state which equals $s$ except that $s_1.x_{i_1} = s'.x_{i_1}$. From $x_{i_1} \in Inc(t)$ we know that $s_1, x_{i_1} \not\in s.x_{i_1}$. This implies that $t'$ is disabled in $s_1$ because $x_{i_1} \in CompTS(t')$. We can continue like this to show that $t'$ is disabled in $s_1$. Note that $s_k$ may differ from $s_k.x_{i_k} \not= s.x_{i_k}$ for some $x \not\in C$. We can obtain $s'$ from $s_k$ and show that $t'$ is disabled in $s'$ similarly to the above case where $C = \emptyset$ was assumed.

Second, assume that $t$ and $t'$ are not local. Again, from $t' \not\in enabled(s)$ we know that $\forall t \in Inc(t) \cap CompTS(t')$. Since $t$ and $t'$ are not local, $\forall t \in Inc(t) \cap CompTS(t')$ because $t$ can only change the local state of process $id(t)$. Therefore, there must be $X' \subseteq \cup_{i \in \mathbb{N}} ps(c_{i, id(t')})$ such that $\forall t \in Inc(t) \cap CompTS(t')$. From $(t, t') \not\in MP-can-enable$, we have three cases. First, $id(t') \not\in O$. From $t' \not\in enabled(s)$ and that $t$ and $t'$ are not local we know that $X' \not\subseteq \cup_{i \in \mathbb{N}} ps(c_{i, id(t')})$, i.e., $X'$ is not accessible for $t'$ in $s$. Therefore, $s_{id(t), id(t')} \not\in s(c_{i, id(t')}) \not= 0$ because $s(c_{i, id(t')}) = s_{id(t), id(t')}$ for all
Fig. 5. Illustration of the proof of Lemma 4.

Let us now consider (b). The proof is illustrated in Figure 5.

Lemma 4: Given any MP protocol, MP-dependency is a dependency relation.

Proof: The proof is indirect. Assume that there is $s, s', s'' \in S, t, t' \in T$ such that $t, t' \in \text{id-enabled}(s)$ and $s \not\xrightarrow{t} s'$ and either (a) $t' \not\in \text{enabled}(s')$ and $(t, t') \not\in \text{MP-dependency}$ or (b) $s \xrightarrow{t} s''$ and there is no $s \xrightarrow{t'} s''$ and $(t, t') \not\in \text{MP-dependency}$. We first consider (a). $t'$ can be disabled in $s'$ either because $s''(id(t')) \neq s'(id(t'))$ or because there is $X \subseteq (\bigcup_{i \in I} P s_i(c_{i, id(t')}))$ which is accessible for $t'$ in $s$ but is not accessible for $t'$ in $s'$. Since $(t, t') \not\in \text{MP-dependency},$ and $t$ and $t'$ are not local. This means that $s(id(t')) = s'(id(t'))$ because $t$ can only change $s(id(t))$ and $id(t) \neq id(t')$. Also, since $t$ and $t'$ are not local we have $(\bigcup_{i \in I} P s_i(c_{i, id(t)})) \subseteq (\bigcup_{i \in I} P s_i(c_{i, id(t')})).$ This means that $X \subseteq (\bigcup_{i \in I} P s_i(c_{i, id(t)}))$, i.e., $X$ is accessible for $t'$ in $s'$.

Let us now consider (b). The proof is illustrated in Figure 5. Again, $t$ and $t'$ cannot be local. Let $id(t) = i$ and $id(t') = j$. Let $s_1$ be the state after the execution of $t$ in $s$. Let $s_1'$ be the state after the execution of $t'$ in $s$ with the accessible set $X_{t'}$ such that $t'$ is executed in $s_1$ with the same $X_{t'}$. Assume indirectly that $X_{t'}$ is not accessible for $t'$ in $s$. This means that $t$ sends a message $m$ to process $j$ such that $m \in X_{t'}$. Since $t$ and $t'$ are well-formed, we have that $j \in t.O, i \in t'.I, m \in t.M_O$, and $m \in t'.M_I$, which implies that $(t, t') \in \text{can-remote-enable}. This contradicts with (b) $t' \not\in \text{MP-dependency}$. Now, from $i \neq j$ we have $s(j) = s_1(j)$ and since $t'$ is executed in $s_1$ with the same $X_{t'}$ we also have $s_1'(j) = s''(j)$.

One reason that no $s \xrightarrow{t} s''$ exists can be that $t$ is disabled in $s_1$. However, this is impossible because process $j$ cannot change the local state of process $i$ and $t'$ can only add messages to the input buffers of $i$. Formally, assume that $s \xrightarrow{t} s_1$. We know that $s(i) = s'_1(i)$ and $X_t \subseteq \bigcup_{k \in K} P s'_1(c_{k, i})$ because $i \neq j$. Therefore, $g_t(X_t, s'_1(i))$ holds, i.e., $t$ is enabled in $s'_1$. Let $s''$ be the state such that $s'_1 \xrightarrow{t} s''$. We now show that $s'' = s''$, which leads to the final contradiction. We showed that $s'_1(j) = s''(j)$. This and $i \neq j$ imply that $s''(j) = s''(j)$. Similarly, we can show that $s''(i) = s''(i)$.

In addition, the content of all channels in $s''$ and $s''$ are identical too. For simplicity, assume that every message $m$ contains the identifier of the channel where $m$ resides. Therefore, it suffices to show that the unions $C''$ and $C''$ of all channels in $s''$ and $s''$ are the same. Let $C$ denote the union of all channels in $s$. Let $A_t (A_r)$ denote the messages added by transition $t$ (and $t'$), i.e., $A_t = \bigcup_{k \in K} P s'_1(c_{k, i}) \setminus s(c_{k, i}) = \bigcup_{k \in K} P s''(c_{k, i}) \setminus s(c_{k, i})$ and $A_r = \bigcup_{k \in K} P s'_1(c_{k, i}) \setminus s(c_{k, i}) = \bigcup_{k \in K} P s''(c_{k, i}) \setminus s(c_{k, i})$. We have that $C'' = (C \cup A_t \cup A_r) \setminus A_t$ and $C'' = (C \setminus A_t \cup A_r) \setminus A_t$. Since $X_t \cap A_r = \emptyset, C'' = C \setminus A_t \cup A_r$ and $A_t$. From $i \neq j$ we have $X_t \cap A_t = \emptyset$, which implies $C'' = C \setminus A_t \cup A_r$. Finally, $(t, t') \not\in \text{can-remote-enable}$ implies $X_{t'} \cap A_r = \emptyset$, which implies $C'' = C''$.

Lemma 5: Given any MP protocol, MP-NET-transition-to-fire is a NET-transition-to-fire relation.

Proof: The proof is indirect. Assume that there is $s \in S, t, t' \in T$ such that $(s, t, t') \in \text{MP-NET-transition-to-fire}$ and a path $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$ such that $t \in \text{enabled}(s_n)$ and $t_i \neq t'$ for every $1 \leq i \leq n$. Let $id(t) = i$ and $id(t') = j$. Since $t \in \text{enabled}(s_n)$ there must be an $X \subseteq M$ such that $s_n \xrightarrow{t(X)} s'$ for some $s' \in S$. From $(s, t, t') \in \text{MP-NET-transition-to-fire}$ we know that $(t, t') \in \text{MP-NET}$, which implies that $t \in \text{ID}$. Also, $t$ is well-formed. Therefore, $X \cap \text{sn}(c_{k, i}) \neq 0 \text{ if } k \in t.I$. Furthermore, from $(t, t') \in \text{MP-NET}$ we know that $j \in t.I$. Let $m$ be a message in $X \cap \text{sn}(c_{k, i})$. Let $t''$ be the transition that sends $m$ to process $i$. From the well-formedness of $t$ we have $m \in t.M_O$, and $s_1(c_{k, i}) t.M_I = \emptyset$ implies that $t''$ must be among $t_1, t_2, \ldots, t_n$. Now, $(t'', t') \in \text{can-remote-enable}$ because $t''$ and $t$ well-formed and by the definition of $m$. Since $id(t'') = j$ and $(t', t'') \in \text{MP-NET}$ the state must be that $t'' = t'$, contradiction.

V. Formal Model of Message-passing Protocols

The system consists of $n$ processes. Without loss of generality, $P = \{1, \ldots, n\}$ denotes the set of IDs of each process. Processes communicate via directed channels where each channel $c_{i, j}$ is a set of messages taken from $M$. For processes $i, j \in P$, $c_{i, j}$ represents a channel from process $i$ to $j$ and is called the outgoing channel of process $i$ and incoming channel of $j$. Each process $i$ is associated with a set $S_i$ of local states. A local state in $S_i \subseteq D_i \times \ldots \times D_i$ is an assignment $(x_{i_1} = v_{i_1}, \ldots, x_{i_l} = v_{i_l})$ of variables $x_{i_1}, \ldots, x_{i_l}$ to values from domains $D_i, \ldots, D_i$. Additionally, each process $i$ is also associated with a set $T_i \subseteq T$ of transitions, where $T$ denotes the set of all transitions. If $t \in T_i$, $id(t) = i$ is the ID of the process executing $t$. A transition $t$ is local to itself or
to another transition \( t' \) if \( \text{id}(t) = \text{id}(t') \), otherwise \( t \) is remote to \( t' \).

A (global) state \( s \in S \) of the system is a tuple containing all channels and local states, where \( S \) denotes the set of all system states. Initially, every process \( i \) assumes an initial state from \( S_i \) and all channels are empty. The content of channel \( c_{i,j} \) and process \( i \)'s local state in \( s \) can be written as \( s(c_{i,j}) \) and \( s(i) \), respectively. The value of a variable \( x \) in \( s \) is given by \( s.x \).

Every transition \( t \) is associated with a true/false condition (or guard) \( g_t \), which is a function of a set of messages and a local process state. A transition \( t \in T_t \) is enabled in \( s \in S \) if, for some subset \( X \) of the union of all incoming channels of \( i \), the condition \( g_t(X, s(i)) \) is true. In this case, the set \( X \) is called accessible for \( t \) in \( s \). If \( t \) is enabled in \( s \), it can be executed with accessible set \( X \) for \( t \) in \( s \), and the resulting state \( s' \) is identical to \( s \) except for the following: (1) the messages in \( X \) are removed from the input channels of \( i \), (2) depending on \( t, s(i) \) and \( X \), local state \( s'(i) \) can be any state from \( S_i \), and (3) zero or more messages are added to each outgoing channel of \( \mathcal{O} \) or irrelevant. By assumption, a process sends no message to \( (\mathcal{I}; \mathcal{O}) \) and \( g_t \) is in some subset \( \mathcal{M} \) for some message \( m \in \mathcal{M} \), then we say that \( t \) sends \( m \) to process \( j \). We use the notation \( s \xrightarrow{t(X)} s' \) or simply \( s \xrightarrow{t} s' \) if \( X \) is clear from the context or irrelevant. By assumption, a process sends no message to itself and, given \( X \) and \( s(i) \), transitions are deterministic. We say that \( t \) is a quorum transition if it can be executed with accessible set \( X \) such that \(|X| > 1\).

### A. Enriched syntax

We associate with every transition \( t \in T \) a tuple \((I, M_I, O, M_O)\) where \( I, O \subseteq P \) are sets of process IDs and \( M_I, M_O \subseteq M \) are sets of messages. The convention is that the field of the tuple is denoted by \( t.field \). The transitions are well-formed which means that for every \( X \subseteq M \), \( s, s' \in S \) and \( j \in P \) such that \( s \xrightarrow{t(X)} s' \) the following holds: (1) \( X \cap s(c_{i/dd(i)}) \neq \emptyset \) implies \( j \in t.I \) and (2) \( X \subseteq t.M_I \). Furthermore, \( s' \subseteq s(c_{id}(t)) \) and \( s \subseteq t.M_O \) and \( s' \subseteq s(c_{id}(t)) \). We say that \( t \) is input-deterministic if \( t \) is in some \( ID \subseteq T \) such that \( ID \subseteq \{t \in T \mid \forall s, s' \in S, i \in P, X \subseteq M : s \xrightarrow{t(X)} s' \} \). We write \( x \in W(t) \) and say the \( t \) is a write with respect to a variable \( x \) if it might change the value of \( x \). A set containing such variables is \( W(t) \supseteq \{x \mid \exists s, s' \in S : s \xrightarrow{t} s' \wedge s.x \neq s'.x \} \). Supposed the operators \(<, >\) and \( = \) are defined for \( x \), the set \( \text{Inc}(t) \subseteq W(t) \) contains variables whose value can only be increased: \( \text{Inc}(x) = \{x \mid \exists s, s' \in S : s \xrightarrow{t} s' \wedge s.x < s'.x \} \). In practice, \( x \) can be a timestamp which is incremented locally or upon receiving a message.

Further, a transition \( t \) is called a read transition with respect to a variable \( x \) if \( g_t \) depends on \( x \). A set containing such variables is \( R(t) \supseteq \{x \mid \exists s, s' \in S, X \subseteq M : (\forall y \neq x : s.y = s'.y) \wedge g_t(X, s(i)) \neq g_t(X, s'(i))\} \). As a special case, \( x \) in \( \text{CompTS}(t) \subseteq R(t) \) if \( t \) cannot be enabled with the same accessible set by increasing \( x \): \( \text{CompTS}(t) \subseteq \{x \mid \forall s, s' \in S, X \subseteq M : g_t(X, s(i)) = false \land g_t(X, s'(i)) = true \land (\forall y \neq x : s.y = s'.y) \land x \} \). In practice, \( x \) denotes a timestamp so that certain messages (with small timestamp) are discarded for higher values of \( x \).

Note that it is always sound to remove transitions from \( ID \) and, given a transition \( t \), to add (remove) variables to (from) \( W(t), R(t), \text{Inc}(t), \text{CompTS}(t) \).

### VI. Paxos, Echo Multicast, Regular Storage Models for LPOR and DPOR Experiments

The following list gives a short description of each target protocol:

- The Paxos protocol solves consensus, a fundamental primitive which can be used to implement state-machine replication [17]. Intuitively, consensus means that at most one value is “chosen”, i.e., all processes agree on this value. Paxos solves consensus if a majority of processes is correct and processes may fail by crashing. Paxos defines proposer, acceptor, and learner processes. A proposer can initiate a consensus instance by proposing a value to be chosen. Acceptors store values proposed by proposers. Learners query acceptors to learn about proposals and output a chosen value.

- Our second example is a fault-tolerant consistent multicast protocol, called Echo Multicast [20]. The agreement property of consistent multicast specifies that no two processes receive different messages. Echo Multicast implements agreement in a Byzantine environment where up to one third of the processes can fail arbitrarily. The protocol defines two types of processes, initiators and receivers. Initiators can launch a new multicast instance, whereas receivers are the recipients of the multicast messages.

- Our third example is a regular storage protocol in the style of [1]. As in every storage protocol, processes can be writers, base objects, and readers. The objective of distributed storage is to reliably store data despite failures of the base storing objects. A regular storage guarantees that a read operation returns a value not older than the one written by the latest preceding write operation. We assume a crash-tolerant setting where less than a majority of all base objects might crash.

**DPOR comparison basis.** Our protocol instances are not only finite-state, they also have acyclic state graphs. Again, this allows us to have a comparison with DPOR for all experiments. The absence of cycles follows from the facts that a transition always consumes at least one message and that the number of messages sent by a transition is finite in any path.

An insight which helps to better understand LPOR’s memory reduction relative to DPOR is the way our example protocols achieve fault-tolerance. They rely on the assumption that each correct process executes an instance of the same replicated service and that the number of correct processes stays above a given threshold. The threshold assumption implies that a set of messages from a large enough subset (or quorum) of processes contains some messages from correct processes. Therefore, a common technique in such systems...
is that the execution of a transition is triggered when a set of messages from a quorum (e.g., a majority) of processes is available. Such transitions are usually modeled via several quorum transitions, each corresponding to a different quorum of processes.

We observe that a high number of concurrent quorum transitions increases LPOR’s advantage over DPOR. Instances with only a single enabled quorum transition significantly reduce the sources of concurrency. Accordingly, DPOR then performs similarly to LPOR. Quorums that do not contain all processes are a source of concurrency because a quorum transition can be executed even if some processes have not replied. Concurrency results in incomparable vector clocks, the condition which triggers the addition of backtracking points in DPOR [10], [21].

Process faults. The above protocols tolerate two classes of faults, crash (Paxos and regular storage) and Byzantine (multicast). We do not explicitly model crash faults. This is because process transitions are scheduled in all possible ways and the effect of crash is modeled by scheduling other (non-crashed) processes first. Intuitively, crashed processes and extremely slow ones are pathologically equivalent. To model malicious faults, we specified Byzantine processes that do not obey the protocol. We consider different attack strategies to challenge the multicast protocol.

We distinguish Byzantine processes whether they are initiators or receivers. A Byzantine initiator attempts to violate the agreement property by sending a different message to each of two groups of honest receivers. A Byzantine receiver sends invalid confirmations to an honest initiator and cooperates with a Byzantine initiator by confirming (signing) both of its messages.

The Multicast (6) instance contains 4 honest receivers, 1 honest initiator, and 1 Byzantine receiver. The Echo Multicast protocol defines a quorum size 4 in this case. In this model, our realized attack strategy allows that all 5 quorum transitions (one receiver excluded in each quorum) are concurrently enabled, which explains DPOR’s low performance (Table I).

Fault injection. We inject faults into (a) correct processes and (b) the specification of the protocols. We specified “F-Paxos”, where one of the acceptors ignores the guard of the transition called “Phase 2(b)” [17]. In “F-Paxos2” we injected a subtle bug: an acceptor does not remember the highest-numbered proposal it has ever accepted but only the last proposal it has accepted. A bug in this protocol cannot be found with six processes (two proposals, three acceptors, one learner) because at least three proposals are needed for the bug to manifest and, in our models, every proposer can send one proposal. For Echo Multicast and regular storage we utilized deliberately incorrect specifications. In Echo Multicast we exceed the threshold of the number of maximum Byzantine processes. For storage we require that a read operation which completes after a write has to return the value written by the write even if the two operations are concurrent.

Initial transition heuristic [5]. We use a heuristic where transitions are preferred which either start a new instance of the protocol or, if there is no such transition, complete no ongoing instance. This heuristic shows good performance in our LPOR experiments. Intuitively, the execution of such a transition “delays” the decision of which instance is pursued at a given process. Surprisingly, this heuristic suggests the opposite of the transaction strategy proposed in [7]. We speculate that the difference lies in that our target protocols allow more concurrency than the cache coherence protocol analyzed in [7]. In fact, the processing of further client requests is blocked until the centralized cache controller (assumed to be fault-free) completes the ongoing instance of the protocol started by another client.