TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs

Samira Tasharofi[1], Rajesh K. Karmani[1], Steven Lauterburg[2], Axel Legay[3], Darko Marinov[1], and Gul Agha[1]

[1] Department of Computer Science, University of Illinois, Urbana, IL 61801, USA
{tasharo1, rkumar8, marinov, agha}@illinois.edu
[2] Salisbury University, Salisbury, MD 21801, USA
stlauterburg@salisbury.edu
[3] INRIA, Campus de Beaulieu, France
alegay@irisa.fr

Abstract. To detect hard-to-find concurrency bugs, testing tools try to systematically explore all possible interleavings of the transitions in a concurrent program. Unfortunately, because of the nondeterminism in concurrent programs, exhaustively exploring all interleavings is time-consuming and often computationally intractable. Speeding up such tools requires pruning the state space explored. Partial-order reduction (POR) techniques can substantially prune the number of explored interleavings. These techniques require defining a *dependency relation* on transitions in the program, and exploit independency among certain transitions to prune the state space.

We observe that actor systems, a prevalent class of programs where computation entities communicate by exchanging messages, exhibit a dependency relation among co-enabled transitions with an interesting property: *transitivity*. This paper introduces a novel dynamic POR technique, TransDPOR, that exploits the transitivity of the dependency relation in actor systems. Empirical results show that leveraging transitivity speeds up exploration by up to two orders of magnitude compared to existing POR techniques.

1 Introduction

Concurrent programs are becoming increasingly important as multicore and networked computing systems become the norm. A model of concurrent programming that has been gaining popularity is the *actor model* [1]. The actor model is used in many systems such as ActorFoundry, Asynchronous Agents, Charm++, E, Erlang, and Scala.[4] Actor programs consist of computing entities called actors (each with its own local state and thread of control) that communicate by exchanging messages asynchronously. An actor *configuration* consists of the local state of the actors and a set of *pending messages*. In response to receiving

[4] For a more extensive list of actor systems, refer to [14].

a message, an actor can update its local state, send messages, or create new actors. At each step in the computation of an actor system [2], an actor from the system is scheduled to process one of its pending messages. Assuming that this processing terminates, the actor system transitions to a new configuration.

Concurrent systems, such as actor systems, present a significant challenge for the testing and verification community. Such systems can exhibit exponentially many different interleavings of concurrent transitions. In the case of actors, the execution of an actor program can have different results from an exponentially large number of potential interleavings of messages. The nondeterminism in actor systems stems from the fact that multiple messages sent to the same actor may be processed in different orders, thus resulting in different configurations, and only some specific interleavings/configurations may reveal bugs.

A naïve exploration that would explore all the interleavings to reach all possible system configurations does not scale. Partial-order reduction (POR) techniques [5, 6, 8, 9, 11, 20, 22, 24–27, 29] can be applied to help mitigate the resulting state-space explosion by exploring a representative subset of all possible interleavings. POR techniques have been widely used for testing and verification of concurrent protocols and software, including in tools such as SPIN [13], VeriSoft [8], and Java PathFinder [28].

To prune state-space exploration, POR techniques explore a *subset* of the set of enabled transitions in each configuration. This subset should be selected such that by exploring only the transitions in the subset, all the properties of interest are guaranteed to be preserved. For example, in one of the popular POR techniques, this subset is defined as a *persistent set* [9]. POR techniques require the definition of a *dependency relation* between transitions in the system and then exploit the independency between certain transitions to compute this subset. A valid dependency relation is a reflexive and symmetric (but *not necessarily transitive*) binary relation on the transitions.

Traditionally, dependencies among transitions, such as in *persistent sets* [9] proposed by Godefroid, were computed via static analysis. More recently, Flanagan and Godefroid introduced a POR algorithm, called dynamic POR (DPOR), that relies on dynamic analysis for computing dependencies [6]. More precisely, this algorithm maintains for each configuration a *backtrack* set and updates the backtrack sets during the execution of a test program. Flanagan and Godefroid proved that the computed backtrack sets are persistent sets [6]. They show that DPOR can significantly improve on POR techniques based on static analysis by computing smaller persistent sets. Note that DPOR is *stateless*, i.e., it does not store states/configurations across different executions.

In this paper, we leverage the fact that actors do not share their states, and we define a dependency relation between the transitions that is *transitive* on the transitions enabled in the same configuration (called co-enabled transitions). We present a new stateless dynamic POR algorithm, called *TransDPOR* which extends DPOR to take advantage of the transitive dependency relations in actor systems. We show that TransDPOR in some cases explores fewer configurations/transitions than DPOR, but it never explores more. TransDPOR is

complete like DPOR, i.e., when the state space is acyclic, TransDPOR can reach every deadlock or local safety violation in the system (space limitations do not allow us to provide a proof of these properties in this version of the paper).

We implemented TransDPOR in Basset [17], a tool for the systematic testing of actor programs written in the Scala programming language [12] or the ActorFoundry library for Java [21]. TransDPOR code is publicly available with Basset at `http://mir.cs.illinois.edu/basset`. We compare TransDPOR and DPOR (we previously adapted the original DPOR algorithm to work for actor systems [18]) on eight programs without bugs and three programs with bugs. The experimental results show that TransDPOR reduces the number of transitions executed during state space exploration by $2.39x$ on *average* and *up to* $163.80x$ over DPOR. When we combine TransDPOR and DPOR with sleep sets (a traditional POR technique) [7], we find that TransDPOR can find bugs *up to* $2.56x$ faster than DPOR.

2 Illustrative Example

To illustrate how TransDPOR works, we use the simple actor program shown in Figure 1. It has four actors: one *master* (which is the entry point of the program), one *registry server*, and two *workers*. The registry keeps track of the actors registered in the system. The master first registers itself by sending its ID to the registry. It then creates two workers and sends them each a message with the registry's ID. After receiving the message, each worker sends its ID to the registry. In the comments for *send* statements, we labeled each of the five messages: $worker1$ and $worker2$ receive messages w_1 and w_2 respectively, and the registry eventually receives three messages—r_0, r_1, and r_2. In this example, the nondeterminism is the order in which the registry receives these three messages and thus assigns the values for its three local variables. For example, the program could have a bug if it assumes that r_0 is received before r_1 and r_2.

We observe that without any assumption one would have to explore up to 5! permutations of the messages exchanged between actors. We will see that this number reduces considerably by using DPOR algorithms that consider a basic property in the actor model. In the actor model, actors have no shared states but only communicate by exchanging messages. Since processing a message in one actor cannot change the states of other actors, only the transitions that process the messages sent to the same actor are *dependent*. Hence, when exploring actor systems, to reach all local safety violations and deadlocks, it suffices to explore different interleavings of processing messages in each actor, i.e., it is not necessary to explore interleavings of processing messages across different actors. For example, if m_a and m_b respectively stand for processing message m_a in actor a and message m_b in actor b, it suffices to explore only one of interleavings $m_a.m_b$ or $m_b.m_a$.

Figure 1 shows the state spaces that DPOR and TransDPOR explore for our example program. Each node represents a configuration, and each edge shows a transition labeled with the message being processed.

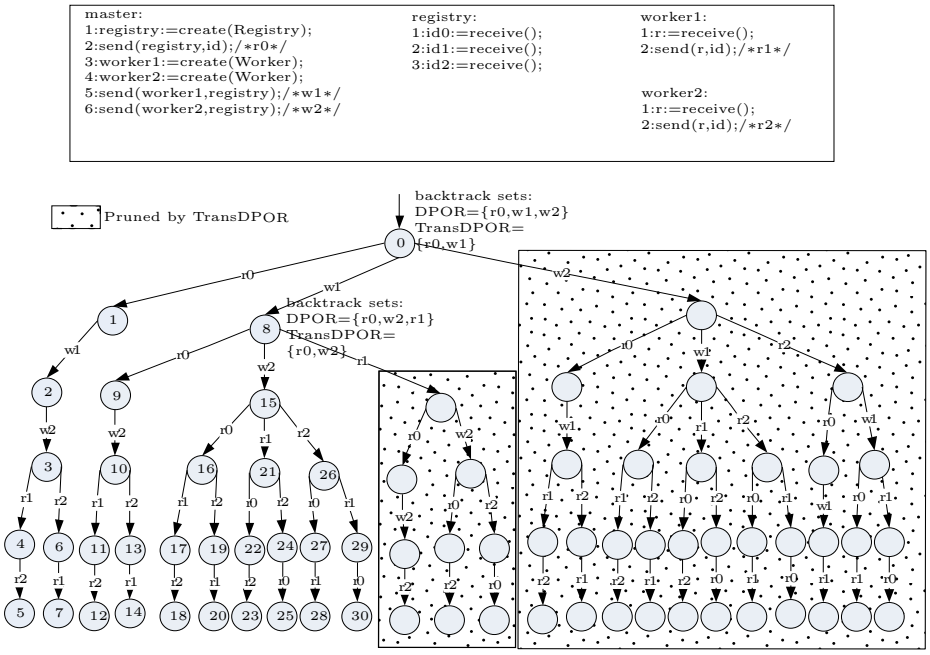

Fig. 1. The registry example and the state space explored by DPOR and TransDPOR.

Let us first focus on DPOR. Specifically, this algorithm first executes an actor program to obtain an execution path and for each configuration keeps a *backtrack* set of all messages to be explored from that configuration. These sets start empty but grow as DPOR discovers dependencies among transitions. In our example, DPOR first executes the path $r_0.w_1.w_2.r_1.r_2$. Then, the algorithm observes that r_1 and r_2 are sent to the same *registry* actor, which makes them dependent. DPOR thus adds r_2 to the backtrack set of configuration 3. Moreover, because r_1 and r_2 are dependent with r_0 but not enabled in the initial configuration, DPOR adds the messages that can produce r_1 and r_2—namely, w_1 and w_2—to the backtrack set of the configuration 0 so that different interleavings of those messages with r_0 can be explored. After the first path, DPOR backtracks in a depth-first manner to configuration 3 and executes $r_0.w_1.w_2.r_2.r_1$. It then backtracks to configuration 0 and explores w_1 and w_2 from that configuration. Observe that some redundant paths such as $r_0.w_2.w_1.r_2.r_1$ have been removed from the exploration. In the end, DPOR explores 24 paths. Recall that DPOR is stateless, i.e., it does not store the history of previous explored paths and every time it backtracks to a configuration it chooses a message from backtrack set and runs the program nondeterministically.

While the above pruning is already an improvement over full exploration, it does not fully exploit the semantics of the actor model. More precisely, one can observe that adding w_1 to the backtrack set of the initial configuration would be

enough to explore all possible permutations of the messages processed in each actor, i.e., all paths of the subtree that starts with w_2 are redundant. Intuitively this is because only *registry* can receive more than one message and different permutations of the three messages sent to the registry have been explored in the previous paths. The same holds for the backtrack set of configuration 8 where the subtree that starts with r_1 is redundant.

Our new POR algorithm TransDPOR detects these redundant paths and as a result explores only 10 paths in this example (those *not* included in dotted boxes in Figure 1). The main idea in TransDPOR is to add (at most) one new message to the backtrack set for a configuration. After the newly added message is explored, *only if it is necessary* TransDPOR adds more messages to the backtrack set. TransDPOR implements this idea by attaching a boolean flag, *freeze* flag, to each configuration. It only adds a message to the backtrack set of a particular configuration if the *freeze* flag of that configuration is not set. While initially this flag is not set in any configuration, TransDPOR sets this flag when it adds a message to the backtrack set of a configuration. It resets the flag when it backtracks to a configuration and explores a new message from that configuration.

In the example, when TransDPOR adds w_1 to the backtrack set of configuration 0, it sets the *freeze* flag and that prevents the addition of w_2 to the backtrack set of configuration 0. The same situation happens for the backtrack set of configuration 8. That leads to smaller backtrack sets than DPOR for the two configurations 0 and 8. This reduction is allowed due to the *transitivity* of the dependency relation between the transitions that may be co-enabled in a configuration, and we show that this reduction does not miss any bug that DPOR can find. An example of adding all messages can be seen in configuration 15 (all three messages r_0, r_1, and r_2 are added to the backtrack set of the configuration). In this configuration, after exploring r_0, both r_1 and r_2 are dependent with r_0, but TransDPOR only adds r_1 to the backtrack set of configuration 15. The *freeze* flag prevents the addition of r_2 to the backtrack set at the same time. Due to the transitivity of the dependency relation, r_1 and r_2 are also dependent. Thus, after exploring r_1 from configuration 15, the *freeze* flag is reset and r_2 is added to the backtrack set of configuration 15. The algorithm will end up adding all three messages r_0, r_1, and r_2 to the backtrack set of configuration 15 and will not miss any permutation of these three messages.

3 Actor Semantics

While the above example relied on an intuitive understanding of actors, we now define the semantics of actor programs precisely. Formally, we view actor programs as state-transition systems. A (global) *state* of an actor program, termed a *configuration*, in notation $\kappa = \langle \alpha, \mu \rangle$, consists of a map $\alpha : \mathbb{A} \to \mathbb{L}$, where $\mathbb{A}$ are actor identifiers and $\mathbb{L}$ are possible local states, and a set of *pending* messages $\mu \subseteq \mathbb{M}$, where $\mathbb{M}$ is the set of all possible messages in the system. We use $\mathbb{K}$ to denote the set of all configurations in a system and $pending(\kappa)$ to denote the

set of pending messages for $\kappa \in \mathbb{K}$. Each message is a tuple of *receiver actor*, *content*, and *unique message identifier*. Conceptually, the messages in μ can be partitioned according to their receiver actor, i.e., μ is a union of disjoint message sets, one for every actor in the system.

At each step in execution, an actor processes a message from its message set: the actor removes the message from its set and potentially updates its local state, sends messages to other actors or itself, and creates new actors. The processing can be viewed as a single, atomic macro-step [2] because actors do not share state [14]. The actor model allows *constraints* that enable or disable processing of some message by an actor depending on its local state. Formally, for an actor a, its constraint $c_a \subseteq \mathbb{L} \times \mathbb{M}$ is a predicate on the local state of the actor and the set of messages.

Definition 1 *The* transition t_m *for a message* m *is a partial function* $t_m : \mathbb{K} \rightharpoonup \mathbb{K}$. *For a given* $\langle \alpha, \mu \rangle \in \mathbb{K}$, *let the receiver of* m *be actor* a *with the local state* s *and constraint* c_a; *the transition* t_m *is enabled if* $t_m(\langle \alpha, \mu \rangle)$ *is defined (i.e.,* $\alpha(a) = s$ *and* $m \in \mu$*) and* $\langle s, m \rangle \in c_a$. *If* t_m *is enabled, it can be executed and produces a new configuration, updating the local state of the actor from* s *to* s', *sending messages* $out_s(t_m)$, *and creating new actors with their initial local state* $new_s(t_m)$*:*

$$\langle \alpha, \mu \rangle \xrightarrow{t_m} \langle \alpha[a \mapsto s'] \cup new_s(t_m), \mu \backslash \{m\} \cup out_s(t_m) \rangle$$

We denote $msg(t_m) = m$ and $actor(t_m) = a$. We denote with $out(t_m)$ and $new(t_m)$ the sets of all new messages and actors, respectively, that the transition t_m can create for *any* local state s. Observe that as is usual in actor semantics, we assume that the behavior of an actor in response to a message is *deterministic*. Moreover, we assume that all transitions terminate–this is a standard assumption in testing programs. Thus, the execution of a transition t in a configuration κ leads to a unique successor κ' (up to the choice of fresh identifiers for new actors and messages).

4 Definitions for Partial-Order Reduction

We introduce several terms and definitions required for presenting our TransD-POR algorithm, following the DPOR presentation style of Flanagan and Godefroid [6]. Then we present an important property of the actor model that will be used to improve DPOR. Let τ be the set of all transitions in the system and τ^* be the set of all transition sequences. We write $\kappa \stackrel{\omega}{\Rightarrow} \kappa'$ to denote that the execution of finite sequence $\omega \in \tau^*$ leads from κ to κ'. A configuration in which no transition is enabled is called a *deadlock* or *terminating* configuration. A *transition sequence* S of an actor system is a (finite) sequence of transitions $t_1.t_2 \ldots t_n$ where there exist configurations $\kappa_0, \ldots, \kappa_n$ such that κ_0 is the initial configuration and $\kappa_0 \xrightarrow{t_1} \kappa_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} \kappa_n$. A transition sequence that ends in a deadlock or terminating configuration is called an *execution path* of the system.

We define an actor system as a transition system $A_G = \langle \mathbb{K}, \Delta, \kappa_0 \rangle$, where $\Delta = \{\langle \kappa, \kappa' \rangle \mid \exists t \in \tau : \kappa \xrightarrow{t} \kappa'\}$ and κ_0 is the initial configuration. We first recap the general definition for a valid dependency relation between transitions [6], then we show how to adapt it to the actor model.

Definition 2 *Let t_1 and t_2 be two transitions of an actor system. We say that t_1 and t_2 are* independent *if for all configurations κ in the state space A_G of the system:*

- *if t_1 is enabled in κ and $\kappa \xrightarrow{t_1} \kappa'$, then t_2 is enabled in κ iff t_2 is enabled in κ' (i.e., independent transitions cannot disable or enable each other); and*
- *if t_1 and t_2 are enabled in κ, there is a unique configuration κ' such that $\kappa \overset{t_1.t_2}{\Rightarrow} \kappa'$ and $\kappa \overset{t_2.t_1}{\Rightarrow} \kappa'$ (i.e., enabled independent transitions must commute).*

The reflexive and symmetric binary relation D is a valid dependency relation on τ iff $D = \{(t_1, t_2) | t_1, t_2 \text{are not independent transitions}\}$. The pair of transitions (t_1, t_2) are said to be dependent iff they belong to a valid dependency relation.

We observe that for actor programs, a transition t_m cannot be enabled until the receiver actor for m is created and the message m is sent (m becomes pending). Second, once a message m is sent to an actor a, only transitions of the actor a can enable or disable the transition t_m that processes the message m. In other words, the constraint c_a does not depend on the global state but only on the local state of a and the message m. Therefore, we can easily show that two transitions t_1 and t_2 are independent if $actor(t_1) \neq actor(t_2)$, $msg(t_1) \notin out(t_2)$, and $actor(t_1) \notin new(t_2)$. Based on these observations, we can cast Definition 2 in the actor programs setting to obtain the following proposition.

Proposition 1. *Two transitions $t_1, t_2 \in \tau$ are* dependent *iff one of the following conditions holds:*

- $actor(t_1) = actor(t_2)$*; or*
- $msg(t_1) \in out(t_2)$ *or* $msg(t_2) \in out(t_1)$*; or*
- $actor(t_1) \in new(t_2)$ *or* $actor(t_2) \in new(t_1)$.

Based on Proposition 1, one can extract an important property of our model, which will be used to improve over DPOR. We say that two transitions t_1 and t_2 *may be co-enabled* if there may exist some configuration in which both t_1 and t_2 are enabled. For a shorthand, we introduce a binary relation on transitions called *race relation*; we say that a pair of transitions $\langle t_1, t_2 \rangle$ are *in race* if they are dependent and may be co-enabled. A key observation is that if $\langle t_1, t_2 \rangle$ are in race, then $actor(t_1) = actor(t_2)$. Indeed, while our definition of dependency allows two other cases ($msg(t_1) \in out(t_2)$ or $actor(t_1) \in new(t_2)$), the transitions that satisfy those two other cases can never be co-enabled (because those cases require that the message or actor for t_1 be created after the execution of t_2). As a result, the following proposition holds.

Proposition 2. *The race relation is reflexive, symmetric, and transitive.*

Given a transition sequence, two adjacent transitions that are independent can be permuted without changing the behavior of the transition sequence. To formalize the set of equivalent transition sequences, we recap the happens-before relation presented in [6].

Definition 3 *The* happens-before *relation* $\rightarrow_S$ *for a transition sequence* $S = t_1 \dots t_n$ *is the smallest relation on* $\{1, \dots, n\}$ *such that (1) if* $i \leq j$ *and* t_i *is dependent with* t_j*, then* $i \rightarrow_S j$*; and (2)* $\rightarrow_S$ *is transitively closed.*

Since happens-before relation is a partial order [6], we introduce the following equivalence relation:

Definition 4 *Two transition sequences* S_1 *and* S_2 *are equivalent iff they have the same set of transitions, and they are linearizations of the same happens-before relation.*

We use $[S]$ to denote the set of transition sequences that are equivalent to S.

5 TransDPOR: A New DPOR Algorithm

Figure 2 presents our TransDPOR algorithm, which explores the state space of an actor system dynamically in a depth-first manner. The underlined parts are the differences between TransDPOR and the original DPOR [6] adapted for actors. The input to the algorithm is a transition sequence S (Line 1). Notation-wise, for a sequence $S = t_1 \dots t_n$: $dom(S)$ is the set $\{1, \dots, n\}$; S_i for $i \in dom(S)$ is transition t_i; $pre(S, i)$ for $i \in dom(S)$ is the configuration in which t_i is executed; and $last(S)$ is the configuration reached after executing S. We denote with $next(\kappa, m)$ the transition that processes message m in the configuration κ. Following [6], we also use a variant of the happens-before relation to determine if some messages are sent as the result of executing other transitions:

Definition 5 *In a transition sequence* S*, the relation* $i \rightarrow_S m$ *holds for* $i \in dom(S)$ *and message* m *iff either (1)* $m \in out(S_i)$ *or (2)* $\exists j \in dom(S)$ *such that* $i \rightarrow_S j$ *and* $m \in out(S_j)$*.*

Like DPOR, TransDPOR maintains a backtrack set $backtrack(\kappa)$, which keeps the messages to be explored from each configuration κ in the input sequence S. The main difference is that, in addition, TransDPOR also uses a boolean flag $freeze(\kappa)$. As explained in Section 2, this flag can prevent adding some messages to $backtrack(\kappa)$, and hence it reduces the size of $backtrack(\kappa)$. As we shall see, because of the transitivity of the race relation, TransDPOR can use this flag to improve over DPOR.

TransDPOR starts by finding the current configuration κ for the input sequence S (Line 2). For every message m in $pending(\kappa)$ (Line 3), it considers the transition $next(\kappa, m)$ for processing that message. It finds the *last* transition i in the sequence S which is in the race with $next(\kappa, m)$, i.e., $actor(S_i) = actor(next(\kappa, m))$ and $i \not\rightarrow_S m$. If the *freeze* flag is set in $pre(S, i)$, the algorithm

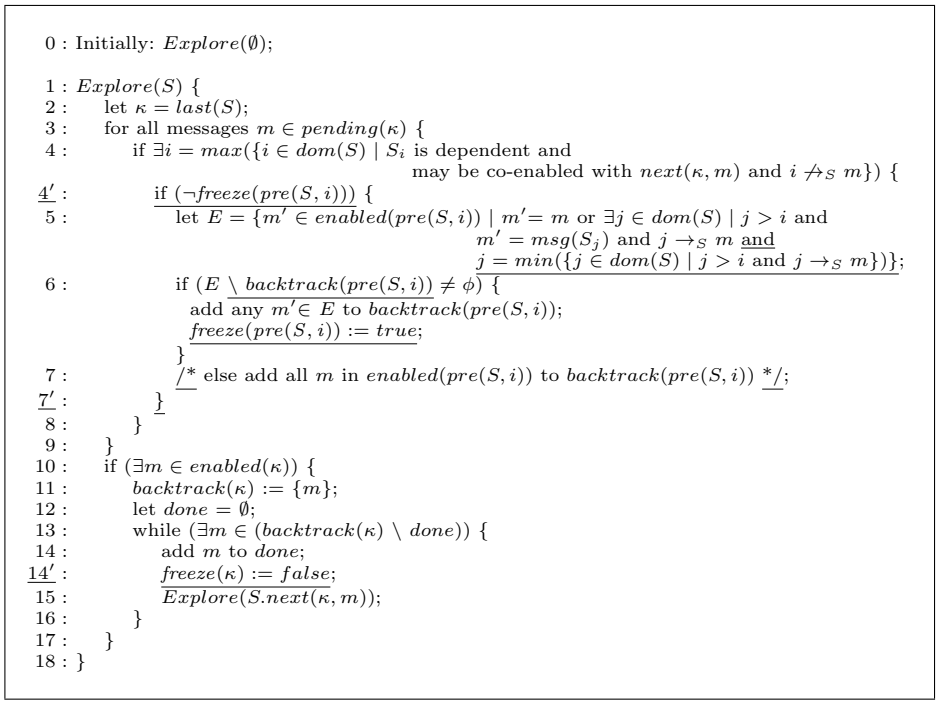

Fig. 2. The TransDPOR algorithm (The differences with DPOR are underlined).

does not update the backtrack set for $pre(S, i)$ (Line 4′); this line does not exist in DPOR and is a major difference between TransDPOR and DPOR. Effectively, this step prevents additional messages in $backtrack(pre(S, i))$ until the previously added message is explored by the algorithm. Due to the transitivity of our race relation, we prove that the messages not added right away are added later if necessary to explore them from κ. However, as TransDPOR does not add them right away, it may terminate faster than DPOR. If $freeze(pre(S, i))$ is not set, the algorithm next finds the message that should be added to $backtrack(pre(S, i))$ by computing the set E (Line 5) from the messages whose transitions are enabled in $pre(S, i)$. If m is enabled in $pre(S, i)$ it is added to E ($m' = m$); otherwise a message m' is added to E if its transition is the *first* transition after S_i that happens before m (in this case, m is produced as a result of executing other transitions after S_i). Note that our approach for computing E differs from DPOR in that it finds the *minimum* index $j > i$ such that j happens before m, while DPOR finds *all* $j > i$ such that j happens before m. As a result of this change, E in our case has at most one element. After computing E, if it contains a message that is not already in $backtrack(pre(S, i))$, the message is added to $backtrack(pre(S, i))$, and the *freeze* flag is set (Line 6). In DPOR, if E is not empty, it can have more than one message, and the algorithm nondeterministically chooses one message to add to $backtrack(pre(S, i))$ (hence "add any").

If E is empty, then m is in $pending(pre(S,i))$ but t_m is not enabled in $pre(S,i)$. Intuitively, because of the transitivity of race relation, every enabled message in $pre(S,i)$ that can enable t_m would be in race with S_i (all of them belong to the same actor) and would be added to $backtrack(pre(S,i))$ either in the next iteration of the for loop or in the recursive calls to $Explore$. Therefore, TransDPOR does not add anything to $backtrack(pre(S,i))$ at this point (Line 7 is effectively deleted). In contrast, in DPOR, if E is empty, the algorithm adds *all* messages from E to $backtrack(pre(S,i))$.

After Lines 3-9 update the backtrack set of configurations seen previously in the sequence S, Lines 10-17 process the messages from the current configuration κ. The algorithm nondeterministically chooses an enabled message from κ (Line 10) to initialize $backtrack(\kappa)$ (Line 11). It then processes all messages from the backtrack set that have not been explored before (Line 13). Every time the algorithm backtracks to κ and explores a new message, it adds that message to the *done* set and resets the *freeze* flag (Line 14′). The algorithm finally recursively calls itself with the transition sequence extended with the $next(\kappa, m)$.

In our example in Section 2, once TransDPOR adds w_1 to $backtrack(\kappa_0)$, it sets $freeze(\kappa_0)$, which prevents from adding w_2 to $backtrack(\kappa_0)$. After the algorithm explores w_1 from κ_0, it resets $freeze(\kappa_0)$, but because none of the messages in paths that start with w_1 from κ_0 is in the race with w_1, no message is added to the $backtrack(\kappa_0)$ (w_2 is not added). Similarly, in κ_8, adding w_2 to $backtrack(\kappa_8)$ prevents from adding r_1. After exploring w_2 from κ_8, because none of the messages r_0, r_1, and r_2 are in the race with w_2, no message is added to $backtrack(\kappa_8)$ (r_1 is not added). On the other hand, consider κ_{15}. After exploring r_0, both r_1 and r_2 are in the race with r_0. Because of the *freeze* flag, the algorithm only adds r_1 to $backtrack(\kappa_{15})$. After exploring r_1 from κ_{15}, because of *transitivity* of race relation, r_2 is also in the race with r_1 and it is eventually added to $backtrack(\kappa_{15})$.

When the loop terminates, the exploration from κ is finished, and the algorithm backtracks to the previous configuration. Note that the algorithm is stateless, i.e., it does not store states/configurations across different execution paths. However, it may store configurations for the current path on a stack, depending on the implementation strategy.

It is trivial to show that TransDPOR never explores more execution paths than DPOR. As a result of the changes that we have made in Lines 4′ and 7 of the algorithm, in each call to $Explore(S)$, we add either fewer or the same number of messages to the backtrack set of each configuration κ.

Theorem 1 states that by starting from a fix initial configuration if A_G is acyclic, TransDPOR will explore at least one execution path from each set of equivalent execution paths in A_G, i.e., it can detect any *deadlock* and *local safety violation* in the program [8].

Theorem 1. *In a program $\mathcal{P}$, by starting from an initial configuration, let A_G be the* acyclic *state-space graph and A_R be the reduced state space explored by TransDPOR. If Ω_G and Ω_R denote the set of execution paths of $\mathcal{P}$ in A_G and A_R respectively, then $\forall \omega \in \Omega_G, \exists \omega' \in \Omega_R$ such that $\omega' \in [\omega]$.*

6 Implementation and Evaluation

To compare TransDPOR and DPOR, we implemented TransDPOR in the Basset tool [17]. Basset provides an extensible environment for testing Java-based actor programs written in the Scala Actors library [12] or ActorFoundry [21]. We use vector clocks [16] to track the happens-before relation at runtime as shown in [24].

We use eight different subject programs in our evaluation. Each actor program was either originally implemented in ActorFoundry or ported to it for this evaluation. `fibonacci` computes the n^{th} element in the Fibonacci sequence. In this case, we show the result for $n = 5$. `quicksort` is a distributed sorting implementation using a standard divide-and-conquer strategy to carry out the computation. `pi` is a porting of a publicly available [23] MPI example, which computes an approximation of π by distributing the task among a set of worker actors. The results shown here are for a configuration with five worker actors. `pipesort` is a modified version of the sorting algorithm used in the dCUTE study [24]. `chameneos` is an implementation of the `chameneos-redux` benchmark from the Great Language Shootout (`http://shootout.alioth.debian.org`). `leader` is an implementation of a leader election algorithm previously used in the dCUTE study [24]. `shortpath` is an implementation of the Chandy-Misra shortest path algorithm [4]. This subject appears twice in the results: once for a graph with 4 nodes (`shortpath4`) and once for a graph with 5 nodes (`shortpath5`), where the two graphs are dissimilar. `regsim` is a server registration simulation. The numbers with the name of subjects, if available, represent the values of program parameters. We performed all experiments using Sun's JVM 1.6.0_20-b02 on a 2.93GHz Intel Core(TM)i7 running Ubuntu release 10.04.

Our recent work [18] shows that the effectiveness of DPOR techniques is highly sensitive to the *order* in which messages are explored. However, one cannot easily determine before the exploration which order will work the best. For that reason, we present results for three ordering heuristics, *ECA*, *LCA*, and *FIFO*. FIFO sorts the messages based on the time they are sent in the ascending order. ECA sorts messages according to the creation time of the receiving actor in ascending order; messages for the *earliest created actor* are considered first. LCA is similar to ECA but sorts the actors in descending order of their creation time.

To illustrate the speedup that can be achieved using TransDPOR, we performed a set of nine experiments which compare explorations performed using DPOR with explorations performed using TransDPOR. Table 1 shows the results for these experiments. For each subject and DPOR technique, we show the number of paths executed in their entirety while exploring the specified subjects, the total number of transitions executed (across all execution paths), the total exploration time in seconds, and memory usage in MB. Since the length of paths might be different in a program, and the time is dependent on the platform and noise in the system, we focus on the number of explored transitions as the primary metric for comparison.

The experiments suggest that TransDPOR can explore up to *over two orders of magnitude* fewer transitions than DPOR. In all the experiments TransDPOR

Table 1. Comparison of TransDPOR and DPOR

		DPOR				TransDPOR				Speedup			
Heur.	Subject	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]
FIFO		40	203	5	176	40	203	4	176	1.00x	1.00x	1.25x	1.00x
ECA	fib5	327	1650	28	455	203	1051	18	377	1.61x	1.57x	1.56x	1.21x
LCA		16	91	3	173	16	91	3	159	1.00x	1.00x	1.00x	1.09x
FIFO		368	1586	26	343	368	1586	26	463	1.00x	1.00x	1.00x	0.74x
ECA	quicksort6	3822	16766	264	381	1519	6992	115	751	2.52x	2.40x	2.30x	0.51x
LCA		32	156	4	197	32	156	4	179	1.00x	1.00x	1.00x	1.10x
FIFO		120	931	16	265	120	931	16	374	1.00x	1.00x	1.00x	0.71x
ECA	pi5	120	931	16	263	120	931	16	374	1.00x	1.00x	1.00x	0.70x
LCA		19845	156070	2509	451	312	2452	40	376	63.61x	63.65x	62.73x	1.20x
FIFO		1791	8562	101	375	755	3541	45	375	2.37x	2.42x	2.24x	1.00x
ECA	pipesort4	288	1293	17	374	288	1293	18	450	1.00x	1.00x	0.94x	0.83x
LCA		5970	32385	361	375	2221	11999	136	451	2.69x	2.70x	2.65x	0.83x
FIFO		3240	19459	233	376	600	3673	44	376	5.40x	5.30x	5.30x	1.00x
ECA	chameneos2	19683	118197	1360	550	1728	10554	123	374	11.39x	11.20x	11.06x	1.47x
LCA		216	1231	16	375	216	1231	16	453	1.00x	1.00x	1.00x	0.83x
FIFO		18098	107780	26872	336	14984	86889	37516	341	1.21x	1.24x	0.72x	0.99x
ECA	leader4	11957	68373	1207	240	11909	68125	1312	266	1.00x	1.00x	0.92x	0.90x
LCA		39238	236330	4301	634	27287	163030	3120	525	1.44x	1.45x	1.38x	1.21x
FIFO		238	910	12	261	238	910	12	261	1.00x	1.00x	1.00x	1.00x
ECA	shortpath4	392	1464	20	262	392	1464	19	260x	1.00x	1.00x	1.05x	1.01x
LCA		640	2158	27	260	370	1337	17	264	1.73x	1.61x	1.59x	0.98x
FIFO		528	2443	32	454	528	2443	33	372	1.00x	1.00x	0.97x	1.22x
ECA	shortpath5	2658	8476	104	368	1170	3737	49	261	2.27x	2.27x	2.12x	1.41x
LCA		1865	7076	93	375	1272	4704	61	375	1.47x	1.50x	1.52x	1.00x
FIFO		211750	590835	14440	989	1320	3607	64	76	160.42x	163.80x	225.63x	13.01x
ECA	regsim	208034	591454	14782	989	1950	5434	93	79	106.68x	108.84x	158.95x	12.52x
LCA		720	1962	34	64	720	1962	36	66	1.00x	1.00x	0.94x	0.97x
									Max	160.42x	**163.80x**	225.63x	13.01x
									Average	2.39x	**2.39x**	2.38x	1.18x

has a speedup for at least one heuristic, and it is *never* the case that the use of TransDPOR results in more executed transitions than DPOR. Although the speedup in transitions executed can at times be small (e.g., only $1.24x$ or less for `leader`), it can be also quite significant. For `chameneos`, the speedup is over $11x$, and for `regsim`, it is over $163x$. The `regsim` experiment using DPOR did not complete in 4 hours for either FIFO or ECA.

***Combining with Sleep Sets*:** Sleep sets is a POR technique based on the history of exploration [7]. Specifically, sleep sets record the transitions that have already been explored from a particular configuration, and avoid exploring them in successor configurations until some condition is met. Sleep sets can further prune the number of transitions and configurations that are explored [8]. In the case where the state space is acyclic (which is the assumption in this paper), sleep sets can be combined with dynamic POR in exactly the same way as static POR [6]. We implemented a variant of TransDPOR that is combined with sleep sets and compared it with the combination of DPOR with sleep sets.

In addition to the eight programs used in our initial experiment, we added three more programs. These programs have such a large state space that the exploration times out without sleep sets. `diningphil` is an implementation of the dining philosopher protocol in ActorFoundry. `minesweeper` is a simulation of the minesweeper game written using the Scala Actors library. `le-erlang` is an implementation of a fault-tolerant leader election algorithm for Erlang that had been running on Ericsson switches. Some bugs were found in the program by Arts et al. [3] in in the presence of node failures. We re-implemented the buggy program in ActorFoundry in order to test it using our tool.

Table 2. Comparison of TransDPOR+Sleep sets and DPOR+Sleep sets

		DPOR+ S				TransDPOR+ S				Speedup			
Heur.	Subject	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]
FIFO		16	101	3	173	16	101	3	173	1.00x	1.00x	1.00x	1.00x
ECA	fib5	16	139	4	159	16	139	4	173	1.00x	1.00x	1.00x	0.92x
LCA		16	91	3	174	16	91	3	174	1.00x	1.00x	1.00x	1.00x
FIFO		32	179	5	181	32	179	5	181	1.00x	1.00x	1.00x	1.00x
ECA	quicksort6	32	272	7	269	32	272	7	270.00x	1.00x	1.00x	1.00x	1.00x
LCA		32	156	5	194	32	156	5	193	1.00x	1.00x	1.00x	1.01x
FIFO		120	931	17	264	120	931	17	376	1.00x	1.00x	1.00x	0.70x
ECA	pi5	120	931	17	266	120	931	17.00	263.00	1.00x	1.00x	1.00x	1.01x
LCA		120	1236	22	377	120	990	18	456	1.00x	1.25x	1.22x	0.83x
FIFO		288	1448	20	378	288	1422	20	377	1.00x	1.02x	1.00x	1.00x
ECA	pipesort4	288	1293	19	376	288	1293	18	376	1.00x	1.00x	1.06x	1.00x
LCA		288	1944	27	376	288	1935	27	376	1.00x	1.00x	1.00x	1.00x
FIFO		216	1681	23	378	216	1453	20	377	1.00x	1.16x	1.15x	1.00x
ECA	chameneos2	216	1826	24	374	216	1530	21	376	1.00x	1.19x	1.14x	0.99x
LCA		216	1231	17	376	216	1231	17	375	1.00x	1.00x	1.00x	1.00x
FIFO		492	3125	43	454	492	3097	43	372	1.00x	1.01x	1.00x	1.22x
ECA	leader4	492	3267	45	376	492	3218	42	377	1.00x	1.02x	1.07x	1.00x
LCA		492	3311	46	680	492	3311	46	377	1.00x	1.00x	1.00x	1.80x
FIFO		126	473	8	261	126	473	8	262	1.00x	1.00x	1.00x	1.00x
ECA	shortpath4	126	489	8	260	126	489	8	260	1.00x	1.00x	1.00x	1.00x
LCA		126	522	9	262	126	502	8	262	1.00x	1.04x	1.13x	1.00x
FIFO		296	1408	22	375	296	1408	22	375	1.00x	1.00x	1.00x	1.00x
ECA	shortpath5	296	1031	16	264	296	997	17	265	1.00x	1.03x	0.94x	1.00x
LCA		296	1228	20	376	296	1218	19	451	1.00x	1.01x	1.05x	0.83x
FIFO		720	3453	37	376	720	2019	22	377	1.00x	1.71x	1.68x	1.00x
ECA	regsim	720	4054	47	375	720	2152	26	452	1.00x	1.88x	1.81x	0.83x
LCA		720	1962	22	264	720	1962	22	375	1.00x	1.00x	1.00x	0.70x
FIFO		1296	8636	141	427	1296	5537	92	417	1.00x	1.56x	1.53x	1.02x
ECA	regsim-2-level	1296	14990	267	558	1296	7486	129	560	1.00x	2.00x	2.07x	1.00x
LCA		1296	6481	115	381	1296	6295	111	381	1.00x	1.03x	1.04x	1.00x
FIFO		31	1375	38	524	31	1375	37	524	1.00x	1.00x	1.03x	1.00x
ECA	diningphil	31	2082	55	348	31	1662	44	366	1.00x	1.25x	1.25x	0.95x
LCA		31	1333	38	374	29	1147	33	407	1.07x	1.16x	1.15x	0.92x
									Max	1.07x	**2.00x**	2.07x	1.80x
									Average	1.00x	**1.13x**	1.13x	0.98x
Buggy programs (Exploration stops at first bug instance.)													
FIFO		1	15	2	112	1	15	2	112	1.00x	1.00x	1.00x	1.00x
ECA	diningphil	16	915	29	340	16	767	22	342	1.00x	1.19x	1.32x	0.99x
LCA	(deadlock)	1	15	2	112	1	15	2	112	1.00x	1.00x	1.00x	1.00x
FIFO		1	29	3	163	1	29	2	163	1.00x	1.00x	1.50x	1.00x
ECA	minesweeper	2710	15577	484	717	2710	15381	499	744	1.00x	1.01x	0.97x	0.96x
LCA	(deadlock)	6993	69350	1950	1041	6993	55430	1651	893	1.00x	1.25x	1.18x	1.17x
FIFO		457	1976	41	462	452	1944	27	427	1.01x	1.02x	1.52x	1.08x
ECA	le-erlang3-failure	30	174	4	241	30	174	4	236	1.00x	1.00x	1.00x	1.02x
LCA	(safety)	233738	759109	14401	2020	93640	296229	3557	1557	2.50x	2.56x	4.05x	1.30x
FIFO		2209	11006	169	605	2191	10146	155	586	1.01x	1.08x	1.09x	1.03x
ECA	le-erlang4	1	27	2	112	1	27	2	112	1.00x	1.00x	1.00x	1.00x
LCA	(no leader, new)	198713	698759	14405	2011	88505	277440	3924	1383	2.25x	2.52x	3.67x	1.45x
									Max	2.50x	**2.56x**	4.05x	1.45x
									Average	1.16x	**1.22x**	1.40x	1.08x

The results are presented in Table 2. For `le-erlang`, our tool was able to find all the previously known bugs in the algorithm (in the presence of node failures). We also tested the algorithm for four processes and *without a failure-recovery scenario*. To our surprise, our tool detected a *new bug*, which allows the program to reach a state in which no leader is elected. We contacted the developers and they confirmed the new bug.

The combination with sleep sets reduces the improvement as sleep sets already prune many redundant transitions; however TransDPOR is always equal to or better than DPOR in terms of paths and transitions. For seven experiments, TransDPOR provides the speedup of over $1.20x$ for at least one heuristic. Note that it is not obvious from the program what may be a good heuristic, and the results table suggests the same. For example, ECA is the best heuristic for `le-erlang`, and FIFO is the best for `minesweeper`. Moreover, different compo-

nents of a single application, such as the `regsim-2-level`, may have different good heuristics for exploration.

Overall, the results suggest that our algorithm combined with sleep sets outperforms the combination of DPOR and sleep sets. We achieved speedup as high as $2x$ for the `regsim` benchmark as shown in Table 2. TransDPOR is also very efficient when exploring programs with bugs. We consistently find the bug faster than DPOR. Even in the presence of sleep sets, we were able to find the bugs up to $2.56x$ faster than DPOR.

7 Related Work

One of the earliest POR approaches is based on computing persistent (or stubborn) sets [8, 9, 27] and the related technique of ample sets [22]. Persistent sets can be computed statically or dynamically. Using static analysis for computing persistent sets [8] suffers from conservative approximation, resulting in coarse persistent sets. Therefore, *dynamic POR* (DPOR) techniques [6], which compute the persistent sets on the fly, have been proposed to yield more accurate persistent sets. Another variation of persistent (or stubborn) is *weak persistent sets* [9, 27], which can generate smaller sets and lead to better reduction. This reduction needs additional knowledge about the transitions that enable and disable each other, which may not be easy to compute during the exploration.

Sen and Agha proposed a DPOR technique for testing multi-threaded programs [25], as well as for testing distributed message-passing programs [24]. Both papers present an operational definition of the set of transitions to be explored from a state, and the presented algorithms are conceptually similar to that in [6]. Kastenberg and Rensink proposed a new DPOR which is based on *probe sets* for handling dynamic creation and destroying of processes and objects [15]. Probe sets relies on abstract enabling and disabling relations among actions, rather than associated sets of concurrent processes. The authors show that their technique leads to a better reduction in comparison to persistent sets.

Recently a new partial-order reduction technique called cartesian POR was proposed, which is based on *cartesian semantics* [11] and stateful exploration. The authors provide an operational definition, and present a dynamic algorithm that overcomes the acyclic state space restriction in stateless approaches. The technique is shown to improve over optimal persistent sets for some examples. The cartesian approach trades space for time since the approach requires storing program states precisely.

Lei and Carver [19] propose a technique that explores only one interleaving from each partial order. However their technique assumes FIFO channels and requires a non-trivial amount of memory for storing interleavings that are yet to be explored. Message Passing Interface (MPI) [10] is a popular environment for writing message-passing programs. MPI programs are more constrained than actor programs. Specifically, MPI processes assume FIFO channels and usually have matched sends and receives. Vakkalanka et al. [26] proposed a stateless DPOR technique for MPI programs, called POE, which exploits these constraints. POE

can produce only one interleaving for large MPI programs that do not have an MPI wild-card receive.

8 Conclusions and Future Work

We have proposed a new stateless DPOR technique called TransDPOR for message-passing (actor) systems. We exploit the transitivity of a dependency relation between co-enabled transitions in actor systems to achieve faster exploration than the existing DPOR based on persistent sets. Experimental results suggest that TransDPOR can substantially reduce the number of transitions executed during state space exploration by up to $163.80x$ compared to DPOR, and it can detect bugs up to $2.56x$ faster than DPOR. TransDPOR code is available with Basset at `http://mir.cs.illinois.edu/basset`.

Although we applied our algorithm for message-passing systems, we believe that the technique discussed in this paper may be applicable to shared-memory multi-threaded programs if a dependency relation between the co-enabled transition is defined so that it is transitive. However, the detailed discussion in this regard is outside the scope of this paper. We also plan to explore the possibility of combining our algorithm with stateful exploration.

9 Acknowledgments

We would like to thank Yaniv Eytani, Bobak Hadidi, Vilas Jagannath, Timo Latvala, and P. Madhusudan for discussions and other assistance during the course of this project, and Nicholas Chen, Stas Negara, Milos Gligoric, Mohsen Vakilian, and the participants of the software engineering seminar at UIUC for comments on a draft of this paper. This material is based upon work partially supported by US Department of Energy under Grant No. DOE DE-FG02-06ER25752 and the National Science Foundation under Grant Nos. CCF-0916893, CNS-0851957, CCF-0746856, and CNS-0509321, and by the AFRL and the AFOSR under agreement number FA8750-11-2-0084.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.
2. G. Agha, I. A. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.
3. T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. *LNCS*, 3395:140–154, 2005.
4. K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *ACM*, 1982.
5. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2-3):151–195, 1994.

6. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
7. P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, pages 176–185, 1991.
8. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems-An Approach to the State-Explosion Problem*. LNCS. 1996.
9. P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *CAV*, pages 438–449, 1993.
10. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. MPI: The Complete Reference: Vol 2, The MPI-2 Extensions, 1998.
11. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. *Model Checking Software*, pages 95–112, 2009.
12. P. Haller and M. Odersky. Actors that unify threads and events. *Lecture Notes in Computer Science*, 4467:171, 2007.
13. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
14. R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In *PPPJava*, pages 11–20, 2009.
15. H. Kastenberg and A. Rensink. Dynamic partial order reduction using probe sets. In *CONCUR*, pages 233–247, 2008.
16. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.
17. S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of Java-based actor programs. In *ASE*, pages 468–479, 2009.
18. S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *FASE*, pages 308–322, 2010.
19. Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32:382–403, 2006.
20. K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV*, pages 164–177, 1992.
21. Open Systems Laboratory, University of Illinois at Urbana-Champaign. *The Actor Foundry: A Java-based Actor Programming Environment*, 1998-09.
22. D. Peled. All from one, one for all: On model checking using representatives. In *CAV*, pages 409–423, 1993.
23. Pi code. `www-unix.mcs.anl.gov/mpi/usingmpi/examples/simplempi/main.htm`.
24. K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FASE 2006*, volume 3922 of *LNCS*, pages 339–356, 2006.
25. K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. *Lecture Notes in Computer Science*, 4383:166, 2007.
26. S. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *CAV*, pages 66–79, 2008.
27. A. Valmari. Stubborn sets for reduced state space generation. In *ATPN*, pages 491–515, 1991.
28. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
29. C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *TACAS*, pages 382–396, 2008.