Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing

Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa

DSIC, Complutense University of Madrid, Spain

Abstract. Testing concurrent systems requires exploring all possible non-deterministic interleavings that the concurrent execution may have. This is because any of the interleavings may reveal the erroneous behaviour. In testing of actor systems, we can distinguish two sources of non-determinism: (1) *actor-selection*, the order in which actors are explored and (2) *task-selection*, the order in which the tasks within each actor are explored. This paper provides new strategies and heuristics for pruning redundant state-exploration when testing actor systems by reducing the amount of unnecessary non-determinism. First, we propose a method and heuristics for actor-selection based on tracking the amount and the type of interactions among actors. Second, we can avoid further redundant interleavings in task-selection by taking into account the access to the *shared-memory* that the tasks make.

1 Introduction

Concurrent programs are becoming increasingly important as multicore and networked computing systems are omnipresent. Writing correct concurrent programs is harder than writing sequential ones, because with concurrency come additional hazards not present in sequential programs such as race conditions, data races, deadlocks, and livelocks. Therefore, software validation techniques urge especially in the context of concurrent programming. Testing is the most widely-used methodology for software validation. However, due to the non-deterministic interleavings of processes, traditional testing for concurrent programs is not as effective as for sequential programs. Systematic and exhaustive exploration of all interleavings is typically too time-consuming and often computationally intractable (see, e.g., [16] and its references).

We consider actor systems [1, 9], a model of concurrent programming that has been gaining popularity and that it is being used in many systems (such as ActorFoundry, Asynchronous Agents, Charm++, E, ABS, Erlang, and Scala). 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 *tasks*. In response to receiving 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, firstly an actor and secondly a process of its pending tasks are scheduled. As actors do not share their states, in testing

E. Ábrahám and C. Palamidessi (Eds.), FORTE 2014, LNCS 8461, pp. 49–65, 2014.

one can assume [13] that the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it releases the processor (gets to a return instruction). At this point, we must consider two levels of non-determinism: (1) *actor-selection*, the selection of which actor executes, and (2) *task-selection*, the selection of the task within the selected actor. Such non-determinism might result in different configurations, and they all need to be explored as only some specific interleavings/configurations may reveal the bugs.

A naïve exploration of the search space to reach all possible system configurations does not scale. The challenge is in avoiding the exploration of redundant states which lead to the same configuration. Partial-order reduction (POR) [6,8] is a general theory that helps mitigate the state-space explosion problem by exploring the subset of all possible interleavings which lead to a different configuration. A concrete algorithm (called DPOR) was proposed by Flanagan and Godefroid [7] which maintains for each configuration a backtrack set, which is updated during the execution of the program when it realises that a non-deterministic choice must be tried. Recently, TransDPOR [16] extends DPOR to take advantage of the transitive dependency relations in actor systems to explore fewer configurations than DPOR. As noticed in [12,16], their effectiveness highly depend on the actor selection order. Our work enhances these approaches with novel strategies and heuristics to further prune redundant state exploration, and that can be easily integrated within the aforementioned algorithms. Our main contributions can be summarized as follows:

1. We introduce a strategy for actor-selection which is based on the number and on the type of interactions among actors. Our strategy tries to find a *stable actor*, i.e., an actor to which no other actor will post tasks.
2. When temporal stability of any actor cannot be proven, we propose to use heuristics that assign a weight to the tasks according to the error that the actor-selection strategy may make when proving stability w.r.t. them.
3. We introduce a task-selection function which selects tasks based on the access to the shared memory that they make. When tasks access disjoint parts of the shared memory, we avoid non-determinism reordering among tasks.
4. We have implemented our actor-selection and task-selection strategies in aPET [2], a Test Case Generation tool for concurrent objects. Our experiments demonstrate the impact and effectiveness of our strategies.

The rest of the paper is organized as follows. Section 2 presents the syntax and semantics of the actor language we use to develop our technique. In Sec. 3, we present a state-of-the-art algorithm for testing actor systems which captures the essence of the algorithm in [16] but adapted to our setting. Section 4 introduces our proposal to establish the order in which actors are selected. In Sec. 5, we present our approach to reduce redundant state exploration in the task selection strategy. Our implementation and experimental evaluation is presented in Sec. 6. Finally, Section 7 overviews related work and concludes.

2 The Actor Model

We consider a distributed message-passing programming model in which each actor represents a processor which is equipped with a procedure stack and an unordered buffer of pending tasks. Initially all actors are idle. When an idle actor's task buffer is non-empty, some task is removed, and the task is executed to completion. Each task besides accessing its own actor's global storage, can post tasks to the buffers of any actor, including its own. When a task does complete, its processor becomes idle, chooses a next pending task to remove, and so on.

2.1 Syntax and Semantics

Actors are materialized in the language syntax by means of objects. An actor sends a message to another actor x by means of an asynchronous method call, written $x\ !\ m(\bar{z})$, being $\bar{z}$ parameters of the message or call. In response to a received message, an actor then spawns the corresponding method with the received parameters $\bar{z}$. The number of actors does not have to be known a priori, thus in the language actors can be dynamically created using the instruction **new**. Tasks from different actors execute in parallel. The grammar below describes the syntax of our programs.

$$
\begin{array}{lll}
M & ::= & \textsf{void}\ m(\bar{T}\ \bar{x})\{s;\} \\
s & ::= & s\ ;\ s \mid x = e \mid x = \textsf{this}.f \mid \textsf{this}.f = y \mid \textsf{if}\ b\ \textsf{then}\ s\ \textsf{else}\ s \mid \\
 & & \textsf{while}\ b\ \textsf{do}\ s \mid x = \textsf{new}\ C \mid x\ !\ m(\bar{z}) \mid \textsf{return}
\end{array}
$$

where x, y, z denote variables names, f a field name and s an instruction. For any entity A, the notation $\bar{A}$ is used as a shorthand for $A_1, ..., A_n$. We use the special actor identifier this to denote the current actor. For the sake of generality, the syntax of expressions e, boolean conditions b and types T is not specified. As in the object-oriented paradigm, a class denotes a type of actors including their behavior, and it is defined as a set of fields and methods. In the following, given an actor a, we denote by $class(a)$ the class to which the actor belongs. $Fields(C)$ stands for the set of fields defined in class C. We assume that there are no fields with the same name and different type. As usual in the actor model [16], we assume that methods do not return values, but rather that their computation modify the actor state. The language is deliberately simple to explain the contributions of the paper in a clearer way and in the same setting as [16]. However, both our techniques and our implementation also work in an extended language with tasks synchronization using future variables [5].

An *actor* is a term $act(a, t, h, \mathcal{Q})$ where a is the actor identifier, t is the identifier of the *active task* that holds the actor's lock or $\perp$ if the actor's lock is free, h is its local heap and $\mathcal{Q}$ is the set of tasks in the actor. A *task* is a term $tsk(t, m, l, s)$ where t is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed or ϵ if the task has terminated. A *state* or

$$(\text{MSTEP})\ \frac{selectActor(S) = act(a, \bot, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, selectTask(a) = t, S \overset{a \cdot t}{\leadsto}{}^{*}\ S'}{S \overset{a \cdot t}{\longrightarrow} S'}$$

$$(\text{SETFIELD})\ \frac{t = tsk(t, m, l, \mathsf{this}.f = y; s)}{act(a, t, h, \mathcal{Q} \cup \{t\}) \leadsto act(a, t, h[f \mapsto l(y)], \mathcal{Q} \cup \{tsk(t, m, l, s)\})}$$

$$(\text{GETFIELD})\ \frac{t = tsk(t, m, l, x = \mathsf{this}.f; s)}{act(a, t, h, \mathcal{Q} \cup \{t\}) \leadsto act(a, t, h, \mathcal{Q} \cup \{tsk(t, m, l[x \mapsto h(f)], s)\})}$$

$$(\text{NEWACTOR})\ \frac{t = tsk(t, m, l, x = \textbf{new}\ D; s), \text{fresh}(a'), h' = newheap(D), l' = l[x \to a']}{act(a, t, h, \mathcal{Q} \cup \{t\}) \leadsto act(a, t, h, \mathcal{Q} \cup \{tsk(t, m, l', s)\}) \cdot act(a', \bot, h', \{\})}$$

$$(\text{ASYNC})\ \frac{t = tsk(t, m, l, x\ !\ m_1(\overline{z}); s), l(x) = a_1,\ \text{fresh}(t_1),\ l_1 = buildLocals(\bar{z}, m_1, l)}{\begin{array}{c} act(a, t, h, \mathcal{Q} \cup \{t\}) \cdot act(a_1, \_, \_, \mathcal{Q}') \leadsto \\ act(a, t, h, \mathcal{Q} \cup \{tsk(t, m, l, s)\}) \cdot act(a_1, \_, \_, \mathcal{Q}' \cup \{tsk(t_1, m_1, l_1, body(m_1))\}) \end{array}}$$

$$(\text{RETURN})\ \frac{t = tsk(t, m, l, \textbf{return}; s)}{act(a, t, h, \mathcal{Q} \cup \{t\}) \leadsto act(a, \bot, h, \mathcal{Q})}$$

Fig. 1. Summarized Semantics for Distributed and Concurrent Execution

configuration S has the form $a_0 \cdot a_1 \cdot \cdots \cdot a_n$, where $a_i \equiv act(a_i, t_i, h_i, \mathcal{Q}_i)$. The execution of a program from a method m starts from an initial state $S_0 = \{act(0, 0, \bot, \{tsk(0, m, l, body(m))\}$. Here, l maps parameters to their initial values (null in case of reference variables), $body(m)$ is the sequence of instructions in method m, and $\bot$ stands for the empty heap.

Fig. 1 presents the semantics of the actor model. As actors do not share their states, the semantics can be presented as a macro-step semantics [13] (defined by means of the transition "$\longrightarrow$") in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to a **return** instruction. In this case, we apply rule MSTEP to select an available task from an actor, namely we apply the function $selectActor(S)$ to select non-deterministically one *active* actor in the state (i.e., an actor with a non-empty queue) and $selectTask(a)$ to select non-deterministically one task of a's queue. The transition $\leadsto$ defines the evaluation within a given actor. We sometimes label transitions with $a \cdot t$, the name of the actor a and task t selected (in rule MSTEP) or evaluated in the step (in the transition $\leadsto$). The rules GETFIELD and SETFIELD read and write resp. an actor's field. The notation $h[f \mapsto l(y)]$ (resp. $l[x \mapsto h(f)]$) stands for the result of storing $l(y)$ in the field f (resp. $h(f)$ in variable x). The remaining sequential instructions are standard and thus omitted. In NEWACTOR, an active task t in actor a creates an actor a' of class D which is introduced to the state with a free lock. Here $h' = newheap(D)$ stands for a default initialization on the fields of class D. ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier t_1. We assume $a \neq a_1$, but the case $a = a_1$ is analogous, the new task t_1 is added to $\mathcal{Q}$ of a. In what follows, a *derivation* or *execution* $E \equiv S_0 \longrightarrow \cdots \longrightarrow S_n$ is a sequence of macro-steps (applications of rule MSTEP). The derivation is *complete* if S_0 is the initial state and all actors in S_n are of the form $act(a, \bot, h, \{\})$. Since the

execution is non-deterministic, multiple derivations are possible from a state. Given a state S, $exec(S)$ denotes the set of all possible derivations starting at S.

3 A State-of-the-Art Testing Algorithm

This section presents a state-of-the-art algorithm for testing actor systems – which captures the essence of the algorithm DPOR in [7] and its extension TransDPOR [16]– but it is recasted to our setting. The main difference with [7,16] is that we use functions *selectActor* and *selectTask* that will be redefined later with concrete strategies to reduce *redundant* state exploration.

To define the notion of redundancy, we rely in the standard definition of partial order adapted to our macro-step semantics. An execution $E = S_0 \xrightarrow{a_1 \cdot t_1} \cdots \xrightarrow{a_n \cdot t_n} S_n$ defines a *partial order* [7] between the tasks of an actor. We write $t_i < t_j$, if t_i, t_j belong to the same actor a and t_i is selected before t_j in E. Given S, we say that $E_1, E_2 \in exec(S)$ are *equivalent* if they have the same partial order for all actors.

Definition 1 (redundant state exploration). *Two complete executions are redundant if they have the same partial order.*

The algorithm DPOR [7], and its extension TransDPOR [16], achieve an enormous reduction of the search space. Function *Explore* in Fig. 2 illustrates the construction of the search tree that these algorithms make. It receives as parameter a derivation E, which starts from the initial state. We use $last(E)$ to denote the last state in the derivation, $next(S, a \cdot t)$ to denote the step $S \xrightarrow{a \cdot t} S'$ and $E \cdot next(S, a \cdot t)$ to denote the new derivation $E \xrightarrow{a \cdot t} S'$. Intuitively, each node (i.e., state) in the search tree is evaluated with a backtracking set *back*, which is used to store those actors that must be explored from this node. The backtracking set *back* in the initial state is empty. The crux of the algorithm is that, instead of considering all actors, the *back* set is dynamically updated by means of function $updateBackSets(E, S)$ with the actors that need to be explored. In particular, an actor is added to *back* only if during the execution the algorithm

```
 1: procedure Explore(E)
 2:     S = last(E);
 3:     updateBackSets(E, S);
 4:     a = selectActor(S);
 5:     if a! = ϵ then
 6:         back(S) = {a};
 7:         done(S) = ∅;
 8:         while ∃(a∈back(S)\done(S)) do
 9:             done(S) = done(S) ∪ {a};
10:             for all t ∈ selectTask(a) do
11:                 Explore(E · next(S, a · t));
```

Fig. 2. A state-of-the-art algorithm for testing

```
{   /* main Block */
  Reg rg = new Reg;
  Worker1 wk1 = new Worker1();
  Worker2 wk2 = new Worker2();
  rg ! p();     //p
  wk1 ! q(rg); // q
  wk2 ! h(rg); // h
}
class Reg {
  int f=1;  int g=1;
  void p() {this.f++; return;}
  void m() {this.g*2; return;}
  void t() {this.g++; return;}
}
```

```
class Worker1 {
  void q(Reg rg) {
    rg ! m();  // m
    return;
  }
}

class Worker2 {
  void h(Reg rg) {
    rg ! t();  // t
    return;
  }
}
```

Fig. 3. Running Example

realizes that it was *needed*. Intuitively, it is *needed* when, during the execution, a new task t of an actor a previously explored, occurs. Therefore, we must try different reorderings between the tasks since according to Def. 1 they might not be redundant. In this case, the back set of the last state S in which a was used to give a derivation step might need to be updated. As a simple example, consider a state S in which an actor a with a unique task t_1 is selected. Now, assume that when the execution proceeds, a new task t_2 of a is spawned by the execution of a task t' of an actor a' and that t' was in S. This means that it is required to consider also first the execution of t_2 and, next the execution of t_1, since it represents a different partial order between the tasks of a. This is accomplished by adding a' to the back set of S, which allows exploring the execution in which a' is selected before a at S, and thus considering the partial order $t_2 < t_1$. The formal definition of *updateBackSets* (and its optimization with *freeze* flags to avoid further redundancy) can be found at [16]. Function *selectActor* at line 4 selects non-deterministically an active actor in S (or returns ϵ if there is none). The *back* set is initialized with the selected actor. The while loop at line 8 picks up an actor in the *back* set that has not been evaluated before (checked in *done* set) and explores all its tasks (lines 10-11).

Example 1. Consider the program in Fig. 3 borrowed from [16] and extended with field accesses to later explain the concepts in Sec. 5. It consists of 3 classes, one *registry* Reg and two *workers* Worker_1 and Worker_2, together with a *main* block from which the execution starts. In Fig. 4 we show the search tree built by executing $\mathit{Explore}(E_0)$, where $E_0 = S_{ini} \stackrel{main}{\longrightarrow} S_0$, and S_{ini} is the initial state from the main block. The branches in the tree show the macro-steps performed labeled with the task selected at the step (the object identifier is omitted). We distinguish three types of edges: dotted edges are introduced by the **for** loop at line 10 in Fig. 4, dashed edges are eliminated by the improvement of [16], and normal edges are introduced by the **while** loop at line 8. After executing the main block,

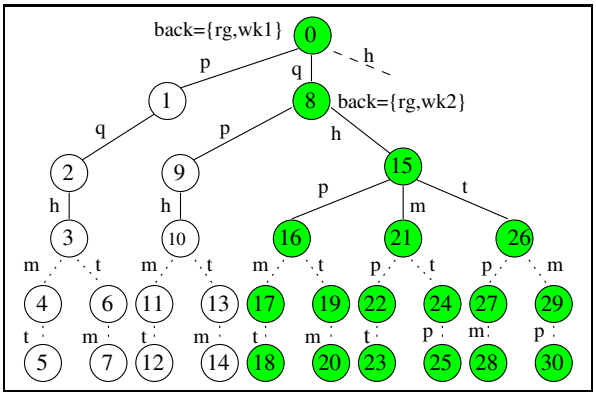

Fig. 4. Execution Tree

there are three actors S_0={rg, wk1, wk2} in node 0 and their queues of pending tasks are $\mathcal{Q}_{rg}$={p()}, $\mathcal{Q}_{wk1}$={q(rg)} and $\mathcal{Q}_{wk2}$={h(rg)} resp. Let us focus on the execution $E_2 = S_0 \xrightarrow{p} S_1 \xrightarrow{q} S_2$. The recursive call *Explore*(E_2) updates the back set of S_0 because a new task m() of rg (previously explored) occurs. Since this task has been produced by the execution of wk1 ! q(rg) and task q(rg) is in S_0, then $back(S_0) = \{\mathsf{rg}, \mathsf{wk1}\}$. The derivation continues and task wk2 ! h(rg) is selected. The execution of E_3=$E_2 \xrightarrow{h} S_3$ introduces t() in the queue of rg. The recursive call *Explore*(E_3) updates the back set of node 0 by introducing wk2 in $back(S_0)$ since it is the responsible of introducing t() on rg (dashed line in node 0). This branch, which generates 14 more (redundant) executions, can be avoided by introducing a "freeze" flag as done in [16], an optimization that we adopt but which is no relevant to explain our contributions. In S_3, the unique active actor rg is selected, and its tasks explored. The execution continues in a similar way and other nodes are added to the back sets. For instance, the back set of node 8 is updated with wk2 from node 10.

4 Actor-Selection Based on Stability Criteria

This section introduces our method to establish the order in which actors are selected based on their *stability* levels. In Sec. 4.1 we first motivate the problem. Afterwards, Sec. 4.2 introduces the notion of *temporarily stable actor* and sufficient conditions to ensure it dynamically during testing. Finally, Section 4.3 presents heuristics based on the stability level of actors.

4.1 Motivation

In Algorithm 2, function *selectActor* selects non-deterministically an active actor in the state. As noticed in [12], the pruning that can be achieved using the testing

algorithm in Sec. 3 is highly dependent on the order in which tasks are considered for processing. Consider the execution tree in Fig. 4. By inspecting the branches associated to the terminal nodes, we can see that the induced partial order $p{<}m{<}t$ occurs in the executions ending in $5, 12, 18$, $p{<}t{<}m$ in those ending in $7, 14, 20$, $m{<}p{<}t$ ending in 23, $m{<}t{<}p$ ending in node 25, $t{<}p{<}m$ ending in 28, and $t < m < p$ ending in 30. Hence, it is enough to consider the coloured subtree since the remaining executions (ending in $5, 7, 12, 14$) have the same partial order than some other execution in the coloured tree. Our work is motivated by the observation that if *selectActor* first selects an actor to which no other actors will post tasks, then we can avoid redundant computations. In particular, if *selectActor* selects wk1, the exploration will lead to the coloured search tree, which does not make any redundant state-exploration.

4.2 The Notion of Temporal Stability

The notion of temporal stability will allow us to guide the selection of actors so that the search space can be pruned further and redundant computations avoided. An actor is *stable* if there is no other actor different from it that introduces tasks in its queue. Basically, this means that the actor is autonomous since its execution does not depend on any other actor. In general, it is quite unlikely that an actor is stable in a whole execution. However, if we consider the tasks that have been spawned in a given state, it is often the case that we can find an actor that is temporarily stable w.r.t. the actors in that state.

Definition 2 (temporarily stable actor). *$act(a, t, h, \mathcal{Q})$ is* temporarily stable *in S iff, for any E starting from S and for any subtrace $S \xrightarrow{*} S' \in E$ in which the actor a is not selected, we have $act(a, t, h, \mathcal{Q}) \in S'$.*

The intuition of the definition is that an actor's queue cannot be modified by the execution of other actors (which are different from itself). E.g., actor rg in Ex. 1 is not temporarily stable in S_0 because the derivation $S_0 \xrightarrow{p} S_1 \xrightarrow{q} S_2$ introduces the task m() in the queue of rg.

Lemma 1. *Let a be a temporarily stable actor in a state S. For any execution E generated by $Explore(S)$ such that $selectActor(S){=}a$, we have $back(S){=}\{a\}$.*

The intuition of the lemma is that if *selectActor* returns a temporarily stable actor a, it is ensured that, from that state, there will be only a branch in the search tree (that corresponds to the selection of a), i.e., no other actors will be added to *back* during its exploration using the testing algorithm *Explore*.

Our goal is to come up with sufficient conditions that ensure actors stability and that can be computed during dynamic execution. To this end, given a method m_1 of class A_1, we define $Ch(A_1{::}m_1)$ as the set of all chains of method calls of the form $A_1{::}m_1{\rightarrow}A_2{::}m_2{\rightarrow}\cdots{\rightarrow}A_k{::}m_k$, with $k{\geq}2$, such that $A_i{::}m_i \neq A_j{::}m_j$, $2{\leq}i{\leq}k{-}1$, $i \neq j$ and there exists a call within $body(A_i{::}m_i)$ to method $A_{i+1}{::}m_{i+1}$, $1{\leq}i{<}k$. This captures all paths $A_2{::}m_2 \rightarrow A_{k-1}{::}m_{k-1}$, without cycles, that go from $A_1{::}m_1$ to $A_k{::}m_k$. The set $Ch(A_1{::}m_1)$ can be computed statically for all methods.

Theorem 1 (sufficient conditions for temporal stability). *We say that* $act(a, t, h, \mathcal{Q}) \in S$, $class(a)=A_n$ *is temporarily stable in* S, *if for every* $act(a', t', h', \mathcal{Q}') \in S$, $a \neq a'$, $class(a')=A_1$, *and for every* $tsk(\_, m_1, l, s) \in \mathcal{Q}'$, *one of the following conditions holds:*

1. *There is no chain* $A_1::m_1 \rightarrow \cdots \rightarrow A_n::m_n \in Ch(A_1::m_1)$; *or*
2. *For all chains* $A_1::m_1 \rightarrow \cdots \rightarrow A_n::m_n \in Ch(A_1::m_1)$, $l(x) \neq a$ *holds, for all* $x \in dom(l)$, $h'(f) \neq a$ *for all* $f \in Fields(A_1)$, *and for all* $act(a'', \_, h'', \_) \in S$ *with* $class(a'')=A_i$, $2 \leq i \leq n-1$, *then* $h''(f) \neq a$, *for all* $f \in Fields(A_i)$.

Intuitively, the theorem above ensures that a' cannot modify the queue of a. This is because (1) there is no transitive call from m_1 to any method of class A_n to which object a belongs, or (2) there are transitive calls from m_1 to some method of class A_n, but no reference to actor a can be found along the chain of objects that will lead to the potential call (that will post a task on actor a). In order to be sound, we check the second condition on all objects in the state whose type matches that of the methods considered in the chain of calls. The following example illustrates why seeking the reference in intermediate objects is required in condition (2).

Example 2. Consider $S=act(a_1, \_, h_1, \mathcal{Q}_1) \cdot act(a_2, \_, h_2, \emptyset) \cdot act(a_3, \_, h_3, \mathcal{Q}_3)$, of classes A, B and C resp., with $\mathcal{Q}_3=\{tsk(t_3, m, l_3, \{y!p(); \textsf{return};\})\}$, $l_3(y) = a_2$, $body(B :: p) = \{x = \textsf{this}.f; x!q(); \textsf{return};\}$, and $h_2(f) = a_1$. Then, even if a_3 does not have a reference to a_1, it is able to introduce the call $q()$ to $\mathcal{Q}_1$. This is because from m there is a call to $p()$ and from there to $f!q()$ with $h_2(f) = a_1$. Thus actor a_1 is not temporarily stable.

Th. 1 allows us to define *selectActor* in Fig. 2 such that it returns an actor a in S which is temporarily stable. If such actor does not exist, then it returns randomly an active object in S.

Example 3. Consider Ex. 1. At node 0 the actor rg is not temporarily stable because in the queue of wk1 there is a call q(rg) (i.e., actor rg can be reachable from q), and in the body of method q there is also a call to method m() of class Reg (i.e., rg can possibly be modified by wk1). However, actors wk1 and wk2 are temporarily stable at node 0. Thus we can select any of these actors to start the exploration. In Fig. 4, actor wk1 has been selected, resulting in the coloured subtree. Similarly, in node 8, rg is not temporarily stable but wk2 it is.

4.3 Heuristics Based on Stability Level

When we are not able to prove that there is a stable actor, then we can use heuristics to determine which actor must be explored first. In particular, we refine the definition of function *selectActor* so that it computes *stability levels* for the actors and selects the actor with highest stability level. Our heuristics tries to weight the loss of precision of the sufficient conditions in Th. 1 in the following way: (1) k_a: this is the value assigned by the heuristics to the case in

which an object is not stable due to a direct call from another object that has a reference to it, (2) k_b: it corresponds to the case in which stability is lost by a transitive (indirect) call from another object that has a reference to it, (3) k_c: this is the case in which the object that breaks its stability does not have a reference to it (instead some intermediate object will have it). It is clear that the heuristics must assign values such that $k_a > k_b > k_c$. This is because the most likely scenario in which the sufficient conditions detect an unfeasible non-stability is (3) since the loss of precision can be large when we seek references to the object within all other objects of the intermediate types in the call chain. The first scenario (1) is more likely to happen since we have both the reference and the direct call. Scenario (2) is somewhere in the middle.

Thus, we define the stability level of $a{\in}class(A_n)$ w.r.t. a $tsk(t, m_1, l, \_)$ of an actor $act(a', \_, h', \_){\in}S$ breaking its stability ($a{\neq}\ a'$, $class(a'){=}A_1$) and a chain $Ch = A_1{::}m_1 \rightarrow^* A_n{::}m_n$, denoted as $st(a, t, Ch, S)$, as follows:

(a) If $l(x){=}a$, for some $x \in dom(l)$ or $h'(f){=}a$, for some $f \in Fields(A_1)$ and $n{=}2$, then $st(a, t, Ch, S){=}k_a$.
(b) If $l(x){=}a$, for some $x \in dom(l)$ or $h'(f){=}a$, for some $f \in Fields(A_1)$ and $n > 2$, then $st(a, t, Ch, S){=}k_b$.
(c) Otherwise, i.e., $l(x) \neq a$, for all $x \in dom(l)$ and $h'(f) \neq a$, for all $f \in Fields(A_1)$, then $st(a, t, Ch, S){=}k_c$.

The *stability level of an actor* $a \in S$, $class(a){=}A_n$, *w.r.t. a task* $tsk(t, m_1, l, \_)$ from $act(a', \_, h', \_) \in S$, $class(a'){=}A_1$, denoted as $st(a, t, S)$, is defined as $\sum st(a, t, Ch, S)$ such that $Ch = A_1{::}m_1 \rightarrow^* A_n{::}m_n \in Ch(A_1{::}m_1)$.

Definition 3 (stability level of an actor). *Let a be a non temporarily stable actor in a state S. The* stability level *of a in S, denoted as $st(a, S)$, is defined as $\sum st(a, t, S)$ such that $t \in \mathcal{Q}'$, $act(a', t, h', \mathcal{Q}') \in S$, $a \neq a'$.*

Given a state $S = a_1 \cdot \ldots \cdot a_n$, the above definition allows us to define the function $selectActor(S)$ in Fig. 2 such that, in case of finding an active actor, it returns a temporarily stable actor a if it exists, and otherwise it returns a_i, where a_i satisfies $st(a_i, S) \geq st(a_j, S)$, for all $1 \leq i, j \leq n$, $i \neq j$.

Example 4. Let us consider the program in Fig. 5, borrowed from [16], which computes the nth element in the Fibonacci sequence in a distributed fashion. The computation starts with the execution of a task $\mathsf{fib}(3)$ on actor a_1, which in turn generates two actors a_2 and a_3 with $Q_{a_2} = \{\mathsf{fib}(2)\}$ and $Q_{a_3} = \{\mathsf{fib}(1)\}$. Both a_2 and a_3 are clearly temporarily stable since there is no reference pointing to them. Let us select a_2 and therefore execute its task $\mathsf{fib}(2)$. This generates two more actors a_4 and a_5 with $Q_{a_4} = \{\mathsf{fib}(1)\}$ and $Q_{a_5} = \{\mathsf{fib}(0)\}$. Again a_4 and a_5 are clearly temporarily stable. After selecting successively a_3, a_4 and a_5 we reach a state S, where a_3, a_4 and a_5 have an empty queue, $Q_{a_1} = \{\mathsf{res}(1)\}$, and $Q_{a_2} = \{\mathsf{res}(1), \mathsf{res}(0)\}$. At this point, our sufficient condition for temporal stability is not able to determine a stable actor. Namely, a_1 is clearly non-stable since the execution of task res on a_2 can, and will, eventually launch a task res

```
class Fib {
  Fib parent;
  Int n = 0;
  Int r = 0;
  Fib(Fib p){
    parent = p;
  }
  void fib(Int v) {
    if (v <= 1) then parent!res(v);
    else {
      Fib child1 = new Fib(this);
      child1!fib(v-1);
      Fib child2 = new Fib(this);
      child2!fib(v-2);
    }
    return;
  }
  void res(Int v) {
    if (n == 0) then {
    n++;
    r = v;
    }
    else {
      r = r + v;
      if (parent ≠ null) then parent!res(r);
    }
    return;
  }
}

{// Main block
  Fib a1 = new Fib(null);
  a1!fib(3);
}
```

Fig. 5. Distributed Fibonacci

on it. However, a_2 is stable, but we cannot determine it syntactically since there is a call chain $Fib::res \rightarrow Fib::res \rightarrow Fib::res$ (i.e. we can reach from $Fib::res$ to $Fib::res$ through $Fib::res$), which forces us to look for a reference to a_2 within all actors of type Fib (cond. 2 of Th. 1). That includes a_4 and a_5 whose parent field points to a_2. Interestingly, our heuristics assigns a much lower non-stability factor to a_2 than to a_1, making it being selected first. Specifically, $st(a_2, S) = k_c$ whereas $st(a_1, S) = 2 * k_a + 2 * k_c$. The latter is because we find 4 tasks that break the stability, 2 of them fulfill condition (a) and the two others condition (c). A wrong selection of a_1 would cause a backtracking at S which produces the exploration of redundant executions. In this concrete example, 8 executions would be explored, whereas with our right selection we explore 4.

We have defined a heuristics which according to our experiments works very well in practice. However, there are other factors to be taken into account to define other heuristics. For instance, it is relevant to consider if the calls appear within conditional instructions (and thus they may finally not hold). This can be easily detected from the control flow graph of the program, where we can define the "depth" of the calls according to the number of conditions that need to be checked to perform the call. In the absence of a stable object, it is also sensible to select the object that is breaking most stabilities, since once it is explored, those objects whose stability it was breaking might become stable.

5 Task Selection Based on Shared-Memory Access

In the section, we present our approach to reduce redundant state exploration within task selection. In Sec. 5.1, we first motivate the problem and characterize the notion of task independence. In Sec. 5.2 we provide sufficient conditions to ensure it. Finally, Sec. 5.3 presents our task selection function.

5.1 Motivation

Let us observe that there can be executions with different partial-orders which lead to the same state, which according to a stronger notion of redundancy could be considered as redundant executions. Consider node 15 in the search tree of Fig. 4. At this point, only tasks of actor rg are available. The derivations ending in nodes 18, 23, 25 result in the same state (namely fields of object rg are f=2, g=3) and the derivations to nodes 20, 28 and 30 also result in the same state (f=2, g=4). The reason for this redundancy is that the execution of p is independent from the executions of m and t because they access disjoint areas of the shared memory. However tasks m and t are not independent and the order in which they are executed affects the final result.

Definition 4. *Tasks t_1 and t_2 are independent, written $indep(t_1, t_2)$, if for any complete execution $S_0 \longrightarrow \cdots \longrightarrow S_n$ with $t_1 < t_2$, there exists another execution $S_0 \longrightarrow \cdots \longrightarrow S_n$ with $t_2 < t_1$.*

Observe that according to Def. 1, the above two derivations are not redundant (as they have a different partial order). However, they are redundant because they lead to the same state, which is a stronger notion of redundancy.

5.2 The Notion of Task Independence

The notion of independence between tasks is well-known in concurrent programming [3]. Basically, tasks t and t' are independent if t does not write in the shared locations that t' accesses, and viceversa. The following definition provides a syntactic way of ensuring task independence by checking the fields that are read and written. Let $act(a, \_, \_, \mathcal{Q}) \in S$ and $tsk(t, m, \_, s) \in \mathcal{Q}$. We define the set $W(t)$ as $\{f \mid \mathsf{this}.f = y \in s\}$. Similarly, the set $R(t)$ is defined as $\{f \mid x = \mathsf{this}.f \in s\}$. The following theorem is an immediate consequence of the definition of independent task above. We denote by $indep(t_1, t_2)$ that t_1 and t_2 are independent.

Theorem 2 (sufficient condition for tasks independence). *Given a state S, an actor $act(a, \_, \_, \mathcal{Q}) \in S$ and two tasks $t_1, t_2 \in \mathcal{Q}$. If $R(t_1) \cap W(t_2) = \emptyset$, $R(t_2) \cap W(t_1) = \emptyset$ and $W(t_1) \cap W(t_2) = \emptyset$, then $indep(t_1, t_2)$ holds.*

Note that since the actor state is local, i.e., fields cannot be accessed from other actors. Thus, all accesses to the heap are on the actor this.

```
9:   for all t ∈ selectTask(a) do
10:    unmark(a); mark(t, a);
11:    Explore(E · next(S, a · t))
```

Fig. 6. Refining Algorithm 2 with Task Selection

5.3 A Task-Selection Function Based on Task-Independence

We now introduce in Alg. 2 a task selection function which avoids unnecessary reorderings among independent tasks. To this end, we introduce marks in the tasks such that the elements in the queues have the form $\langle t, \mathit{flag}\rangle$, where t is a task and *mark* is a boolean flag which indicates if the task can be selected. Furthermore, we treat queues as lists and assume that its elements appear in the order in which they were added to the queue during execution. In order to implement task independence in Alg. 2, we replace lines 10 and 11 of Alg. 2 by those in Fig. 6 where we have that: (1) function $\mathit{selectTask}(a)$ returns the list of unmarked tasks in the queue $\mathcal{Q}$ of a, i.e, those tasks of the form $\langle t, \mathit{false}\rangle$; (2) procedure $\mathit{unmark}(a)$ traverses $\mathcal{Q}$ and changes the flag *mark* to *false*; and (3) procedure $\mathit{mark}(t, a)$ sets the flag *mark* to *true* for all tasks which are independent with t and occur in $\mathcal{Q}$ after t.

Intuitively the task selection process works as follows. Given $\mathit{act}(a, \_, \_, \mathcal{Q}) \in S$, $\mathcal{Q}$ contains a list $[t_1, \ldots, t_n]$ of tasks. These tasks are selected one by one traversing $\mathcal{Q}$ (line 10 of Alg, 2). This means that if t_i is selected by $\mathit{selectTask}(a)$ and t_i is independent from t_j, then $i < j$, i.e., the task t_i is selected before t_j. Furthermore, procedure $\mathit{mark}(t_i, a)$ puts the flag *mark* of t_j to *true*. Thus, in the following step in which actor a is selected, task t_j cannot be chosen, i.e., the *direct* order $t_i < t_j$ is pruned. By direct order, we mean that t_j is selected immediately after t_i. However, when t_j is selected from S, as it occurs after t_i, then t_i will not be marked. This branch will capture the direct order $t_j < t_i$. Since both orders generate equivalent states, no solution is missed.

Example 5. Consider the execution tree in Fig. 4, and the subtree from node 15 in Fig. 7, where $\bar{t}$ denotes that the flag *mark* of t is *true*. At this point, all tasks in rg have the flag *mark* set to *false*. Thus $\mathit{selectTask}(\mathsf{rg})$ returns the list $[p, m, t]$. Procedure *unmark* does nothing. The execution of $\mathit{mark}(p, \mathsf{rg})$ then sets the flag *mark* of m and t to *true* since $\mathit{indep}(p, m)$ and $\mathit{indep}(p, t)$. This branch is therefore cut at node 16 ($\mathit{selectTask}(\mathsf{rg})$ returns the empty list). Afterwards, the selection of m from node 15 does not mark any task. However, when selecting p from node 21, procedure $\mathit{mark}(p, \mathsf{rg})$ sets the flag of t to *true* since we have the independence relation $\mathit{indep}(p, t)$. Hence at node 22 the branch is cut ($\mathit{selectActor}(\mathsf{rg})$ returns the empty list). Similarly, at node 27 the branch is cut because of $\mathit{indep}(p, m)$. The only derivations are those ending in nodes 25 and 30 which correspond to the order of tasks $m < t < p$ and $t < m < p$, resp.

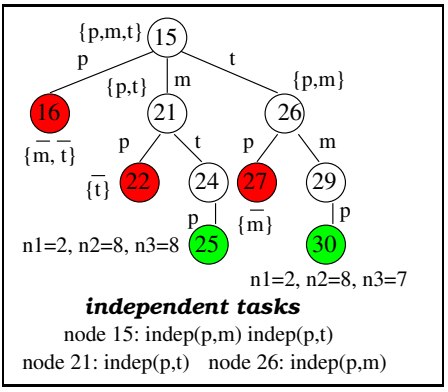

Fig. 7. Pruning due to task-selection

6 Implementation and Experimental Evaluation

We have implemented and integrated all the techniques presented in the paper within the tool aPET [2], a test case generator for ABS programs which is available at `http://costa.ls.fi.upm.es/apet`. ABS [10] is a concurrent, object-oriented, language based on the *concurrent objects* model, an extension of the actors model which includes *future variables* and synchronization operations. Handling those features within our techniques does not pose any technical complication. This section reports on experimental results which aim at demonstrating the applicability, effectiveness and impact of the proposed techniques during testing. The experiments have been performed using as benchmarks: (i) a set of classical actor programs borrowed from [12,13,16] and rewritten in ABS from ActorFoundry, and, (ii) some ABS models of typical concurrent systems. Specifically, *QSort* is a distributed version of the Quicksort algorithm, *Fib* is an extension of the example at Fig. 5, *PI*, computes an approximation of π distributively, *PSort* is a modified version of the sorting algorithm used in the dCUTE study [13], *RegSim* is a server registration simulation, *DHT* is a distributed hash table, *Mail* is an email client-server simulation, and *BB* is a classical producer-consumer. All sources are available at the above website. For each benchmark, we consider two different tests with different input parameters. Table 1 shows the results obtained for each test. After the name, the first (resp. second) set of columns show the result with (resp. without) our task selection function. For each run, we measure: the number of finished executions (column *Execs*); the total time taken and number of states generated by the whole exploration (columns *Time* and *States*); and the number of states at which no stable actor is found and the heuristics is used for actor selection (column H). Times are in milliseconds and are obtained on an Intel(R) Core(TM) i5-2300 CPU at 2.8GHz with 8GB of RAM, running Linux Kernel 2.6.38.

Table 1. Experimental evaluation

	No task sel. reduction				With task. sel. reduct.				Speedup	
Test	**Execs**	**Time**	**States**	**H**	**Execs**	**Time**	**States**	**H**	**Execs**	**Time**
QSort(5)	16	14	72	23	16	15	72	23	1.0x	1.0x
QSort(6)	32	29	146	55	32	29	146	55	1.0x	1.0x
Fib(5)	16	17	72	23	16	15	72	23	1.0x	1.0x
Fib(7)	4096	5425	16760	6495	4096	5432	16760	6495	1.0x	1.0x
Pi(3)	6	10	38	3	6	10	38	3	1.0x	1.0x
Pi(5)	120	65	932	5	120	66	932	5	1.0x	1.0x
PSort(4,1)	288	70	1294	144	288	71	1294	144	1.0x	1.0x
PSort(4,2)	5760	1389	25829	2880	288	71	1304	144	20.0x	19.6x
RegSim(6,1)	10080	804	27415	0	720	135	3923	0	14.0x	6.0x
RegSim(4,2)	11520	860	31576	0	384	70	2132	0	30.0x	12.3x
DHT(a)	1152	132	3905	0	36	5	141	0	32.0x	26.4x
DHT(b)	480	97	2304	0	12	4	85	0	40.0x	24.2x
Mail(2,2)	2648	553	11377	0	460	119	2270	0	5.8x	4.6x
Mail(2,3)	1665500	>200s	5109783	0	27880	4022	94222	0	>60x	>50x
BB(3,1)	155520	23907	475205	0	4320	674	13214	0	36.0x	35.5x
BB(4,2)	1099008	165114	3028298	0	45792	6938	126192	0	24.0x	23.8x

A relevant point to note, which is not shown in the table, is that no backtracking due to actor selection is performed at any state of any test. The number of executions is therefore induced by the non-determinism at task selection. In most states, overall in 99.9% of them, our sufficient condition for temporal stability is able to determine a stable actor (compare column *H* against *States*). Interestingly, at all states where no stable actor can be found, the heuristics for temporal stability guides the execution towards selections of actors which are indeed temporarily stable. This demonstrates that, even though our sufficient condition for stability and heuristics are syntactic, they are very effective in practice since they are computed dynamically on every state. Another important point to observe is the huge pruning of redundant executions which our task selection function is able to achieve for most benchmarks. Last two columns show the gain of the task selection function in number of executions and time. In most benchmarks, the speedup ranges from one to two orders of magnitude (the more complex the programs the bigger the speedup). There is however no reduction in the first three benchmarks. This is because they only generate actors of the same type, and at most two kinds of tasks, usually recursive, which are dependent. This is also the main reason of the loss of precision of our sufficient condition for temporal stability for these benchmarks (namely, cond. 2 of Th. 1 needs to consider all actors in the state).

There are two more benchmarks, *Chameneos* and *Shortpath*, also borrowed from [16], that have been used in our evaluation. We do not provide concrete data for them in the table since they cannot be handled yet by our current implementation. In *Chameneos* the heuristics needs to be used at many states in

order to select an actor. The heuristics of Sec. 4.3 enriched to take into account calls affected by conditional instructions (as described at the end of Sec. 4) would always be able to select actors which are indeed temporarily stable. The *ShortPath* benchmark poses new challenges. It builds a cyclic graph of actors, all of the same type, which interact through a recursive task. An intelligent actor selection heuristics able to prune redundant executions in this case would require detecting tasks which execute their base case. This could be done by computing *constrained call-chains*, and checking dynamically that the constraints hold in order to sum-up the effect of the call-chain when computing the non-stability factor.

7 Related Work and Conclusions

We have proposed novel techniques to further reduce state-exploration in testing actor systems which have been proven experimentally to be both efficient and effective. Whereas in [12, 16] the optimal redundancy reduction can only be accomplished by trying out different selection strategies, our heuristics is able to generate the most intelligent strategy on the fly. Additionally, our task selection reduction has been shown to be able to reduce the exploration in up to two orders of magnitude. Our techniques can be used in combination with the testing algorithms proposed in [7, 16]. In particular, the method in [16] makes a blind selection on the actor which is chosen for execution first. While in some cases, such selection is irrelevant, it is known that the pruning that can be achieved is highly dependent on the order in which tasks are considered for processing (see [12]). Sleep sets, as defined in [8], can be used as well to guide actor-selection by relying on different criteria than ours (in particular, they use a notion of independence different from ours). However, we have not found practical ways of computing them, while we can syntactically detect stable actors by some inspections in the state. Also, we define actor selection strategies based on the stability level of actors. The accuracy of such strategies can be improved by means of static analysis. In particular, points-to analysis [15] can be useful in Th. 2 to detect more accurately if there is a reference to an object from another one and also to know from which object a method is invoked. Another novelty of our approach to reduce useless state-exploration is to consider the access to the shared memory that tasks make. This allows us to avoid non-deterministic task-selection among independent tasks. A strong aspect of our work is that it can be used in symbolic execution [4, 11] directly. In symbolic execution, it is even more crucial to reduce state-exploration, since we already have non-deterministic choices due to branching in the program and due to aliasing of reference variables. In aPET, we use our method to prune the state-exploration of useless interleavings in the context of symbolic execution of actor programs. Recently, the project Setak [14] has developed a new testing framework for actor programs. Differently to us, where everything is automatic, part of the testing is doing manually, and programmers may specify the order of tasks during the execution of a test.

Acknowledgements. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137.

References

1. Agha, G.A.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Albert, E., Arenas, P., Gómez-Zamalloa, M., Wong, P.Y.H.: aPET: A Test Case Generation Tool for Concurrent Objects. In: Proc. of ESEC/FSE 2013, pp. 595–598. ACM (2013)
3. Andrews, G.R.: Concurrent Programming: Principles and Practice. Benjamin/Cummings (1991)
4. Clarke, L.A.: A System to Generate Test Data and Symbolically Execute Programs. IEEE Transactions on Software Engineering 2(3), 215–222 (1976)
5. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
6. Esparza, J.: Model checking using net unfoldings. Sci. Comput. Program. 23(2-3), 151–195 (1994)
7. Flanagan, C.: Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In: Proc. of POPL 2005, pp. 110–121. ACM (2005)
8. Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 176–185. Springer, Heidelberg (1992)
9. Haller, P., Odersky, M.: Scala actors: Unifying Thread-Based and Event-Based Programming. Theor. Comput. Sci. 410(2-3), 202–220 (2009)
10. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
11. King, J.C.: Symbolic Execution and Program Testing. Commun. ACM 19(7), 385–394 (1976)
12. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 308–322. Springer, Heidelberg (2010)
13. Sen, K., Agha, G.: Automated Systematic Testing of Open Distributed Programs. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer, Heidelberg (2006)
14. Setak, A.: Framework for Stepwise Deterministic Testing of Akka Actors, `http://mir.cs.illinois.edu/setak`
15. Steensgaard, B.: Points-to Analysis in almost Linear Time. In: Proc. of POPL 1991, pp. 32–41. ACM Press (1996)
16. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE 2012. LNCS, vol. 7273, pp. 219–234. Springer, Heidelberg (2012)