Test Case Generation of Actor Systems

Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa

DSIC, Complutense University of Madrid, Spain

Abstract. Testing is a vital part of the software development process. It is even more so in the context of concurrent languages, since due to undesired task interleavings and to unexpected behaviours of the underlying task scheduler, errors can go easily undetected. Test case generation (TCG) is the process of automatically generating *test inputs* for interesting *coverage criteria*, which are then applied to the system under test. This paper presents a TCG framework for *actor systems*, which consists of three main elements, which are the original contributions of this work: (1) a symbolic execution calculus, which allows symbolically executing the program (i.e., executing the program for unknown input data), (2) improved techniques to avoid performing redundant computations during symbolic execution, (3) new termination and coverage criteria, which ensure the termination of symbolic execution and guarantee that the test cases provide the desired degree of code coverage. Finally, our framework has been implemented and evaluated within the aPET system.

1 Introduction

Concurrent programs are becoming increasingly important as multicore and networked computing systems are omnipresent. Writing correct concurrent programs is more difficult than writing sequential ones, because with concurrency come additional hazards not present in sequential programs such as race conditions, 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 in industry. It typically requires at least half of the total cost of a software project. *Test Case Generation* (TCG) is a key component to automate testing. It consists in generating *test inputs* for interesting *coverage criteria*, which are then applied to the system under test. Examples of coverage criteria for sequential code are: *statement coverage*, which requires that each instruction of the code is executed; *path coverage*, which requires that each possible path of the execution is tried; etc.

We consider actor systems [2, 14], a model of concurrent programming that has been gaining popularity and that 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.

The aim of this work is to develop a framework for TCG of actor systems. A standard approach to generating test cases statically is to perform a *symbolic execution* of the program [6–8,12,17,19,20], where the contents of variables are expressions rather than concrete values. Symbolic execution produces a system of constraints over the input variables and the actor's fields containing the conditions to execute the different paths. The conjunction of these constraints represents the equivalence class of inputs that would take this path. This produces, by construction, a (possibly infinite) set of test cases, which satisfy the path-coverage criterion. Briefly, the TCG framework that we propose has three main components, which are the contributions of this work: (1) in Sec. 3, we leverage the semantics used for testing in [3] to the more general setting of symbolic execution; (2) in Sec. 4, we extend and improve the techniques to avoid redundant computation of [3] to eliminate redundancies in symbolic execution and; (3) in Sec. 5, we propose novel termination and coverage criteria, which guarantee termination of the process. We have implemented our framework in aPET [4], a TCG tool for concurrent objects. Our experiments demonstrate the usefulness, impact and effectiveness of the proposed techniques.

2 The Language

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 and chooses a next pending task to execute.

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 priory, thus in the language actors can be dynamically created using the instruction **new**. Tasks from different actors execute in parallel. As in the object-oriented paradigm, a class C denotes a type of actors and it is defined as a set of fields $\mathcal{F}(C)$ and methods **void** $m(\bar{T}\ \bar{x})\{s;\}$. The grammar for an instruction s is:

$$\begin{aligned} s ::= s\ ;\ s \mid x = e \mid \textbf{while}\ b\ \textbf{do}\ s \mid \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ s \mid \\ \mathsf{this}.f = y \mid x = \mathsf{this}.f \mid x = \textbf{new}\ C \mid x\ !\ m(\bar{z}) \end{aligned}$$

where x, y, z denote variables names and f a field name. 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. We assume

$$(\mathsf{mstep})\ \frac{\mathit{selectA}(S) = \mathsf{ac}(\mathtt{r}, \bot, h, Q), Q \neq \emptyset, \mathit{selectT}(\mathtt{r}) = t, S \overset{\mathtt{r}\cdot t}{\rightsquigarrow^*} S'}{S \xrightarrow{\mathtt{r}\cdot t} S'}$$

$$(\mathsf{setf})\ \frac{t = tk(t, m, l, \mathsf{this}.f = y; s)}{\mathsf{ac}(\mathtt{r}, t, h, Q \cup \{t\}) \rightsquigarrow \mathsf{ac}(\mathtt{r}, t, h[f \mapsto l(y)], Q \cup \{tk(t, m, l, s)\})}$$

$$(\mathsf{getf})\ \frac{t = tk(t, m, l, x = \mathsf{this}.f; s)}{\mathsf{ac}(\mathtt{r}, t, h, Q \cup \{t\}) \rightsquigarrow \mathsf{ac}(\mathtt{r}, t, h, Q \cup \{tk(t, m, l[x \mapsto h(f)], s)\})}$$

$$(\mathsf{new})\ \frac{t = tk(t, m, l, x = \textbf{new } D; s), n = \text{fresh}(), h' = \mathit{newheap}(D), l' = l[x \to \mathtt{r}_n^D]}{\mathsf{ac}(\mathtt{r}, t, h, Q \cup \{t\}) \rightsquigarrow \mathsf{ac}(\mathtt{r}, t, h, Q \cup \{tk(t, m, l', s)\}) \cdot \mathsf{ac}(\mathtt{r}_n^D, \bot, h', \{\})}$$

$$(\mathsf{asy})\ \frac{t = tk(t, m, l, x \ !\ m_1(\bar{z}); s), l(x) = \mathtt{r}_1, t_1 = \text{fresh}(), \mathtt{r} \neq \mathtt{r}_1, l_1 = \mathit{newlocals}(\bar{z}, m_1, l)}{\begin{array}{c}\mathsf{ac}(\mathtt{r}, t, h, Q \cup \{t\}) \cdot \mathsf{ac}(\mathtt{r}_1, t', h', Q') \rightsquigarrow \\ \mathsf{ac}(\mathtt{r}, t, h, Q \cup \{tk(t, m, l, s)\}) \cdot \mathsf{ac}(\mathtt{r}_1, t', h', Q' \cup \{tk(t_1, m_1, l_1, bd(m_1))\})\end{array}}$$

$$(\mathsf{return})\ \frac{t = tk(t, m, l, \epsilon)}{\mathsf{ac}(\mathtt{r}, t, h, Q \cup \{t\}) \rightsquigarrow \mathsf{ac}(\mathtt{r}, \bot, h, Q)}$$

Fig. 1. Summarized Semantics for Distributed and Concurrent Execution

that there are no fields with the same name and different type. As usual in the actor model [2, 14, 22], we suppose that a method does not return a value, but rather that its computation modifies the actor state. The language is simple to explain the contributions of the paper in a clear way, as done in [3, 22].

An *actor* is a term $\mathsf{ac}(\mathtt{r}_n^C, t, h, Q)$ where $\mathtt{r}$ stands for reference, n is the actor identifier, C is the class name, t is the identifier of the *active task* that holds the actor's lock or $\bot$ if the actor's lock is free, h is its local heap and Q is the set of tasks in the actor. A *heap* h is a mapping $h : \mathcal{F}(C) \mapsto \mathbb{V}$, where $\mathbb{V} = \mathbb{Z} \cup \mathit{Ref} \cup \{\textbf{null}\}$ and Ref stands for the set of references of the form $\mathtt{r}_n^C$. Whenever it is clear from the context, we will omit n and C from actor identifiers by using only $\mathtt{r}$. A *task* is a term $tk(t, m, l, s)$ where t is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to $\mathbb{V}$, and s is the sequence of instructions to be executed. Sometimes we use the identifier t to refer to entire task and we use ϵ to denote an empty sequence of instructions. A *state* S has the form $\mathtt{r}_0 \cdot \mathtt{r}_1 \cdot \ldots \cdot \mathtt{r}_n$, where $\mathtt{r}_i$ is used to refer to the whole actor $\mathsf{ac}(\mathtt{r}_i^{C_i}, t_i, h_i, Q_i)$ and $\mathtt{r}_i \neq \mathtt{r}_j$, $1 \leq i, j \leq n$, $i \neq j$.

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 [21] (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 the end of the method. In this case, we apply rule (mstep) to select an available task from an actor, namely we apply the function $\mathit{selectA}(S)$ to select non-deterministically one *active* actor in the state (i.e., an actor with a non-empty queue) and $\mathit{selectT}(\mathtt{r})$ to select non-deterministically one task of $\mathtt{r}$'s queue. The transition $\rightsquigarrow$ defines the evaluation within a given actor. We sometimes label transitions with $\mathtt{r}\cdot t$, the name of the actor $\mathtt{r}$ and task t selected in (mstep) The rules (getf) and (setf) 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

```
void ft(int n) {
   if (n > this.mx) {
      this ! wk(n,this.mx);
      this ! dg(n-this.mx);
   } else {
      this ! wk(n,n);
      this ! rp(1);
   }
}
```

```
void wk(int n,int h) {
  while (h > 0){
     this.r = this.r * n;
     n = n - 1;
     h = h - 1;
  }
}
```

```
void dg(int n) {
   Fact wkr = new Fact(this,this.mx);
   wkr ! ft(n);
}
void rp(int x) {
   this.r = this.r * x;
   if (this.b != null) this.b ! rp(this.r);
}
```

Fig. 2. Running Example with **class** Fact(Fact b, int mx) {int r = 1; . . . }

and thus omitted. In (new), an active task t in actor $\mathtt{r}$ creates an actor $\mathtt{r}_n^D$ of class D with a fresh identifier $n = \text{fresh}()$, 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. (asy) spawns a new task (the initial state is created by *newlocals*) with a fresh task identifier $t_1 = \text{fresh}()$. We assume $\mathtt{r}{\neq}\mathtt{r}_1$, but the case $\mathtt{r}{=}\mathtt{r}_1$ is analogous, the new task t_1 is added to Q of $\mathtt{r}$. In what follows, a *derivation* E is a sequence $S_0 \longrightarrow \cdots \longrightarrow S_k$ of macro-steps (applications of (mstep)) starting from an initial state $\mathsf{ac}(\mathtt{r}_0^C, \bot, h, \{tk(0, m, l, bd(m))\})$, where l (resp. h) maps parameters (resp. fields) to elements in $\mathbb{V}$ and $bd(m)$ is the sequence of instructions in the body of m. The derivation is *complete* if all actors in S_k are of the form $\mathsf{ac}(\mathtt{r}, \bot, h, \{\})$. Since the execution is non-deterministic in the selection of actor and task, multiple derivations are possible from a state.

Example 1. Consider the class Fact in Fig. 2, which contains three fields Fact b, int mx and int r. Fields b and mx can be initialized in the constructor Fact(Fact b, int mx) whereas r is always initialized to 1. Let us suppose an actor a is asked to compute the factorial of n (by means of call a ! ft(n)). Actor a computes n∗(n−1)∗ . . . ∗(n−a.mx+1) by means of task wk(n, a.mx), and *delegates* to another actor the rest of the computation, by means of task dg(n−a.mx). When an actor is asked to compute the factorial of an n, which is smaller than its mx, then the call this ! wk(n, n) computes directly the factorial of n and the result is *reported* to its caller by means of task rp. The result is then reported back to the initial actor in a chain of rp tasks using field b, which stores the caller actor. The computed result of each actor is stored in field r. The program has a bug, which is only exploited in a concrete sequence of interleavings when at least three actors are involved. Let us consider two derivations that may arise among others from the initial state $S_0{=}\mathsf{ac}(\mathtt{r}_0, \bot, h_0, \{tk(0, \mathsf{ft}, l_0, bd(\mathsf{ft}))\})$, where $h_0(\mathsf{r}){=}1$, $h_0(\mathsf{b}){=}\textbf{null}$, $h_0(\mathsf{mx})$ $=2$ and $l_0(\mathsf{n}){=}5$, i.e., we want to compute factorial of 5 with this.mx equals 2. Arrows are labeled with the identifier of the task(s) selected and it is executed entirely. The contents of the heap and local variables are showed when it is relevant or updated (we only show the new updates). We use h_i, l_i to denote the heap of actor $\mathtt{r}_i$ and the local variables of task t_i respectively.

(a) $S_0 \xrightarrow{0} \mathsf{ac}(\mathtt{r}_0, 0, h_0, \{tk(1, \mathsf{wk}, [\mathsf{n}\mapsto 5, \mathsf{h}\mapsto 2], bd(\mathsf{wk})), tk(2, \mathsf{dg}, [\mathsf{n}\mapsto 3], bd(\mathsf{dg}))\})$ $\xrightarrow{(1,2)^*}$

(b) $\mathsf{ac}(\mathtt{r}_0, \bot, [\mathsf{r}\mapsto 5{*}4], \{\}) \cdot \mathsf{ac}(\mathtt{r}_1, \bot, [\mathsf{r}\mapsto 1, \mathsf{b}\mapsto \mathtt{r}_0], \{tk(3, \mathsf{ft}, [\mathsf{n}\mapsto 3], bd(\mathsf{ft}))\})$ $\xrightarrow{(3)^*}$

$$(c)\ \left.\begin{array}{l}\mathsf{ac}(\mathtt{r}_0,\bot,h_0,\{\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,h_1,\{\})\cdot\mathsf{ac}(\mathtt{r}_2,\bot,h_2,\{tk(4,\mathsf{ft},l_4,bd(\mathsf{ft}))\})\\ \quad h_1(\mathsf{r}){=}3{*}2, l_4(n){=}1, h_2(\mathsf{r}){=}1, h_2(\mathsf{b}){=}\mathtt{r}_1\end{array}\right\}\xrightarrow{(4)^*}$$
$$(d)\ \mathsf{ac}(\mathtt{r}_0,\bot,h_0,\{\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,h_1,\{tk(5,\mathsf{rp},[\mathsf{x}{\mapsto}1],bd(\mathsf{rp}))\})\cdot\mathsf{ac}(\mathtt{r}_2,\bot,h_2,\{\})\xrightarrow{(5)^*}$$
$$(e)\ \mathsf{ac}(\mathtt{r}_0,\bot,[\mathsf{r}=5{*}4{*}3{*}2],\{\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,h_1,\{\})\cdot\mathsf{ac}(\mathtt{r}_2,\bot,h_2,\{\})$$

Note that after executing task 5 we compute the final state (e), where h_0 stores in the field r the value of factorial of 5. Suppose now that, in the above trace, from (b), we first select method dg but we do not execute method wk, and all calls to method rp are executed before method wk. Then:

$$(c)\ \left\{\begin{array}{l}\mathsf{ac}(\mathtt{r}_0,\bot,h_0,\{\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,h_1,\{tk(4,\mathsf{wk},l_4,bd(\mathsf{wk}))\})\cdot\\ \mathsf{ac}(\mathtt{r}_2,\bot,[\mathsf{b}{\mapsto}\mathtt{r}_1],\{tk(5,\mathsf{ft},[\mathsf{n}{\mapsto}1],bd(\mathsf{ft}))\})\end{array}\right\}\xrightarrow{(5)^*}$$
$$(d)\ \left\{\begin{array}{l}\mathsf{ac}(\mathtt{r}_0,\bot,h_0,\{\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,h_1,\{tk(4,\mathsf{wk},l_4,bd(\mathsf{wk}))\})\cdot\\ \mathsf{ac}(\mathtt{r}_2,\bot,[\mathsf{b}{\mapsto}\mathtt{r}_1],\{tk(6,\mathsf{wk},l_6,bd(\mathsf{wk})),tk(7,\mathsf{rp},l_7,bd(\mathsf{rp}))\})\end{array}\right\}\xrightarrow{(7)^*}$$
$$(e)\ \left\{\begin{array}{l}\mathsf{ac}(\mathtt{r}_0,\bot,h_0,\{\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,h_1,\{tk(4,\mathsf{wk},l_4,bd(\mathsf{wk})),tk(8,\mathsf{rp},l_8,bd(\mathsf{rp})\})\cdot\\ \mathsf{ac}(\mathtt{r}_2,\bot,h_2,\{tk(6,\mathsf{wk},l_6,bd(\mathsf{wk}))\})\end{array}\right\}\xrightarrow{(8)^*}$$
$$(f)\ \left\{\begin{array}{l}\mathsf{ac}(\mathtt{r}_0,\bot,h_0,\{tk(9,\mathsf{rp},l_9,bd(\mathsf{rp}))\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,h_1,\{tk(4,\mathsf{wk},l_4,bd(\mathsf{wk}))\})\cdot\\ \mathsf{ac}(\mathtt{r}_2,\bot,h_2,\{tk(6,\mathsf{wk},l_6,bd(\mathsf{wk}))\})\end{array}\right\}\xrightarrow{(9)^*}$$
$$(g)\ \mathsf{ac}(\mathtt{r}_0,\bot,[\mathsf{r}{\mapsto}5{*}4],\{\})\cdot\mathsf{ac}(\mathtt{r}_1,\bot,[\mathsf{r}{\mapsto}3{*}2],\{\})\cdot\mathsf{ac}(\mathtt{r}_2,\bot,[\mathsf{r}{\mapsto}1],\{\})\ \}$$

In the last step we have computed $h_0(\mathsf{r}){=}5{*}4$, which is an incorrect result, hence exploiting the above-mentioned bug. Although the execution at this point is not finished, none of the pending tasks will modify the value of field r in $\mathtt{r}_0$. □

3 Symbolic Execution

The main component of our TCG framework is *symbolic execution* [12, 17, 19, 20, 23], whereby instead of on actual values, programs are executed on symbolic values, represented as *constraint variables*. The outcome is a set of equivalence classes of inputs, each of them consisting of the *constraints* that characterize a set of feasible concrete executions of a program that takes the same path and, optionally constraints, which characterize the output of the execution. For instance, consider method int abs(int x){if (x<0) return -x; else return x;}. The outcome is the set $\{\langle \mathsf{X}{<}0, \mathsf{Y}{\doteq}-\mathsf{X}\rangle, \langle \mathsf{X}{\geq}0, \mathsf{Y}{\doteq}\mathsf{X}\rangle\}$ where Y refers to the return value. Essentially, there are two elements in the set which will lead to two *test inputs*, the first one captures the execution of the then branch, with the constraint $\mathsf{X}{<}0$ on the input and $\mathsf{Y}{=}{-}\mathsf{X}$ on the output. The second element captures the execution of the else branch. Symbolic execution thus produces a set of test cases satisfying the path coverage criterion. We use uppercase characters to syntactically distinguish constraint variables from ordinary program variables. In our simplified language, we consider two types of equality and inequality constraints, those that involve integer values and those that involve references (the latter refer to aliasing conditions between references). The constraint variables can represent field or variable names. Given an infinite set of constraint variable names $X, Y, F, G, \ldots \in \mathcal{V}$, an *atomic constraint* φ is of the form:

$$\varphi ::= X{\doteq}n \mid X{\doteq}Y \mid X{>}Y \mid X{\doteq}ref$$

where $ref{\in}Ref^*$ can be either $\mathtt{r}_n^C$ or $\mathtt{s}_n^C$, $n{\in}\mathbb{N}$ and C is a class name. Each element of the form $\mathtt{r}_n^C$ stands for a reference of class C created by using a **new**

$$(\mathsf{mstep})_{\Phi}\ \frac{selectA(\mathcal{S}) = \mathsf{ac}(ref, \bot, \_, \mathcal{Q}, \_), \mathcal{Q} \neq \emptyset, selectT(ref) = t, \mathcal{S}\Box\mathcal{I} \overset{ref\cdot t}{\leadsto^{*}}_{\Phi} \mathcal{S}'\Box\mathcal{I}'}{\mathcal{S}\Box\mathcal{I} \xrightarrow{ref\cdot t}_{\Phi} \mathcal{S}'\Box\mathcal{I}'}$$

$$(\mathsf{setf})_{\Phi}\ \frac{t = tk(t, m, \rho, \mathsf{this}.f = y; s), \theta' = \theta[f \mapsto F], \varphi = \{F \doteq \rho(y)\}}{\mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t\}, \Phi) \leadsto_{\Phi} \mathsf{ac}(ref, t, \theta', \mathcal{Q}\cup\{t'\}, \Phi \cup \varphi)}$$

$$(\mathsf{getf})_{\Phi}\ \frac{t = tk(t, m, \rho, x = \mathsf{this}.f; s), \rho_1 = \rho[x \mapsto X], \varphi = \{X \doteq \theta(f)\}}{\mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t\}, \Phi) \leadsto_{\Phi} \mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{tk(t, m, \rho_1, s)\}, \Phi \cup \varphi)}$$

$$(\mathsf{new})_{\Phi}\ \frac{t = tk(t, m, \rho, x = \textbf{new}\ D; s), n{=}\mathrm{fresh}(), \rho_1{=}\rho[x \mapsto X], \Phi_1{=}init(D), \varphi{=}\{X \doteq \mathtt{r}_n^D\}}{\begin{array}{c}\mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t\}, \Phi) \leadsto_{\Phi} \\ \mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{tk(t, m, \rho_1, s)\}, \Phi \cup \varphi) \cdot \mathsf{ac}(\mathtt{r}_n^D, \bot, \theta_{\mathsf{s}}, \{\}, \Phi_1)\end{array}}$$

$$(\mathsf{asy})_{\Phi_1}\ \frac{\begin{array}{c} t{=}tk(t, m, \rho, x\ !\ m_1(\bar{z}); s), \rho(x) \text{ is object-bounded in } \Phi \\ \Phi \models \rho(x) \doteq ref', t_1{=}\mathrm{fresh}(), \mathrm{fresh}(m_1(\bar{w})\{s_1;\}), \Phi'{=}\Pi_{\rho(\bar{z})}\Phi \cup \{\rho_{\mathsf{s}}(\bar{w}) \doteq \rho(\bar{z})\}\end{array}}{\begin{array}{c}\mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t\}, \Phi) \cdot \mathsf{ac}(ref', \_, \theta', \mathcal{Q}_1, \Phi_1) \leadsto_{\Phi} \\ \mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t'\}, \Phi) \cdot \mathsf{ac}(ref', \_, \theta', \mathcal{Q}_1\cup\{tk(t_1, m_1, \rho_{\mathsf{s}}, s_1)\}, \Phi_1 \cup \Phi')\end{array}}$$

$$(\mathsf{asy})_{\Phi_2}\ \frac{\begin{array}{c} t{=}tk(t, m, \rho, x\ !\ m_1(\bar{z}); s), class(x){=}D, \rho(x) \text{ is not object-bounded in } \Phi \\ t_1{=}\mathrm{fresh}(), \mathrm{fresh}(m_1(\bar{w})\{s_1;\}), \Phi'{=}\Pi_{\rho(\bar{z})}\Phi \cup \{\rho_{\mathsf{s}}(\bar{w}) \doteq \rho(\bar{z})\}\end{array}}{\begin{array}{c}\mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t\}, \Phi) \cdot \mathsf{ac}(\mathsf{s}_n^D, \_, \theta', \mathcal{Q}_1, \Phi_1) \leadsto_{\Phi} \\ \mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t'\}, \Phi \cup \{\rho(x) \doteq \mathsf{s}_n^D\}) \cdot \mathsf{ac}(\mathsf{s}_n^D, \_, \theta', \mathcal{Q}_1\cup\{tk(t_1, m_1, \rho_{\mathsf{s}}, s_1)\}, \Phi_1 \cup \Phi')\end{array}}$$

$$(\mathsf{asy})_{\Phi_3}\ \frac{\begin{array}{c} t{=}tk(t, m, \rho, x\ !\ m_1(\bar{z}); s), class(x){=}D, \rho(x) \text{ is not object-bounded in } \Phi, n{=}\mathrm{fresh}() \\ t_1{=}\mathrm{fresh}(), \mathrm{fresh}(m_1(\bar{w})\{s_1;\}), \Phi'{=}\{\theta_{\mathsf{s}}(\mathsf{this}) \doteq \mathsf{s}_n^D\} \cup \Pi_{\rho(\bar{z})}\Phi \cup \{\rho_{\mathsf{s}}(\bar{w}) \doteq \rho(\bar{z})\}\end{array}}{\begin{array}{c}\mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t\}, \Phi)\Box\mathcal{I} \leadsto_{\Phi} \mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t'\}, \Phi \cup \{\rho(x) \doteq \mathsf{s}_n^D\}) \cdot \\ \mathsf{ac}(\mathsf{s}_n^D, \bot, \theta_{\mathsf{s}}, \{tk(t_1, m_1, \rho_{\mathsf{s}}, s_1)\}, \Phi')\Box\mathcal{I}\cup\{\langle \mathsf{s}_n^D, \theta_{\mathsf{s}}\rangle\}\end{array}}$$

$$(\mathsf{return})_{\Phi}\ \frac{t = tk(t, m, \rho, \epsilon)}{\mathsf{ac}(ref, t, \theta, \mathcal{Q}\cup\{t\}, \Phi) \leadsto_{\Phi} \mathsf{ac}(ref, \bot, \theta, \mathcal{Q}, \Phi)}$$

Fig. 3. Symbolic Execution Calculus. t' stands for $tk(t, m, \rho, s)$

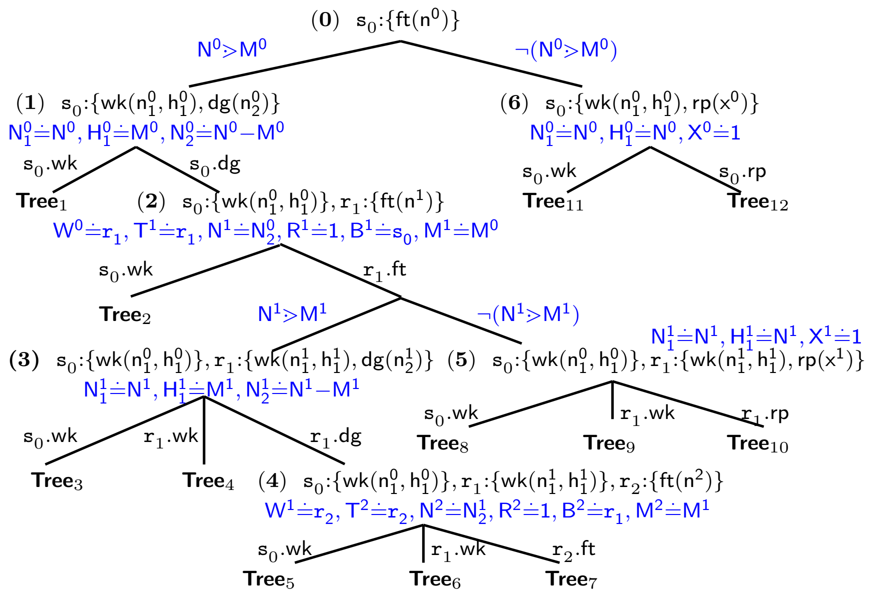
instruction. References of the form $\mathtt{s}_n^C$ refer to actors not created with **new** but arising from asynchronous calls in which the calling actor is unknown at the time of the call. We denote by Φ a conjunction of atomic constraints. We use simply $\mathtt{r}$ (resp. $\mathtt{s}$) instead of $\mathtt{r}_n^C$ (resp. $\mathtt{s}_n^C$) when the values of C and n are irrelevant. We use ref to refer either to $\mathtt{r}$ or $\mathtt{s}$ and sometimes set notations to refer to Φ.

Fig. 3 presents the operational semantics of symbolic execution for the concurrent instructions (the sequential ones are standard). A *symbolic state* has the form $\mathcal{S}\Box\mathcal{I}$, where $\mathcal{S}$ is a collection of *symbolic actors* and $\mathcal{I}$ is a set of actors required to know the actors that must be in the initial state to get to the final state, and thus be able to build the test cases in Sec. 5.2. For simplicity, we omit $\mathcal{I}$ in all rules except for $(\mathsf{asy})_{\Phi_3}$, since it is the only rule that modifies $\mathcal{I}$. A *symbolic actor* is represented as $\mathsf{ac}(ref, t, \theta, \mathcal{Q}, \Phi)$, where $ref \in Ref^*$ is the actor identifier, t is the identifier of the active task, $\mathcal{Q}$ is the queue of symbolic pending tasks, Φ is a set of constraints involving the fields of the actor and the variables of its tasks, and $\theta : \mathcal{F}(C) \cup \{\mathsf{this}\} \mapsto \mathcal{V}$ is called the *field renaming* that maps fields of class C and the this actor to $\mathcal{V}$. In particular, if f is a field of class C, then $\theta(f)$ is the current constraint variable F representing f in Φ. A *symbolic task* $tk(t, m, \rho, s)$ of a method m in a class C contains the sequence of instructions s

to be executed together with the current renaming $\rho : \mathit{vars}(\mathit{bd}(m)) \mapsto \mathcal{V}$ of those variables in m, where $\mathit{vars}(A)$ stands for the set of variables occurring at any entity A. The constraints associated to these variables are stored in the actor in Φ. As for fields, the renaming is required to build correctly the set of atomic constraints Φ and keep the relation between these constraints and the original variables of method m. An initial state to symbolically execute $m(\bar{x})$ on s_n^C has the form $\mathcal{S}_0 \square \mathcal{I}_0$, where $\mathcal{S}_0 = \mathsf{ac}(\mathsf{s}_0^C, \bot, \theta_\mathsf{s}, \{\mathit{tk}(0, m, \rho_\mathsf{s}, \mathit{bd}(m))\}, \{\theta_\mathsf{s}(\mathsf{this}) \doteq \mathsf{s}_0^C\})$, θ_s (resp. ρ_s) is a starting fresh mapping, i.e., $\theta_\mathsf{s}(f)$ (resp. $\rho_\mathsf{s}(x)$) are mapped to fresh variables and $\mathcal{I}_0 = \langle \mathsf{s}_0^C, m(\bar{x}), \theta_\mathsf{s}, \rho_\mathsf{s} \rangle$.

The different rules of the symbolic semantics in Fig. 3 extend those in Fig. 1 with constraint handling as follows. As notation, $\rho_1 = \rho[x \mapsto X]$ maps in ρ variable x to the fresh variable X. Rule $(\mathsf{setf})_\Phi$ updates the field mapping θ with the fresh variable F, and stores the new constraint $F \doteq \rho(y)$ in Φ. Since a field is modified, the mapping θ in the actor must be updated. However, in rule $(\mathsf{getf})_\Phi$, the field f is read and thus, it is not required to update θ but ρ. Rule $(\mathsf{new})_\Phi$ adds the constraint $X \doteq \mathsf{r}_n^D$ to Φ and updates ρ. The function $\Phi_1 = \mathit{init}(D)$ initializes Φ_1 with the corresponding initialization of the fields in D and the this actor, i.e., $\theta_\mathsf{s}(\mathsf{this}) \doteq \mathsf{r}_n^D \in \Phi_1$ and if a field f in D is initialized to a value v, then $\theta_\mathsf{s}(f) \doteq v$ will be in Φ_1. Rule $(\mathsf{return})_\Phi$ allows us to apply rule $(\mathsf{mstep})_\Phi$. A main aspect is the treatment of asynchronous calls $x \ ! \ m_1(\bar{z})$, which distinguishes three cases:

1. $(\mathsf{asy})_{\Phi_1}$: *Object x exists in the store.* This condition is checked by seeing if x is bounded in Φ. Formally we say that x is *object-bounded* in Φ if $\rho(x) \in \mathit{vars}(\Phi)$ and $\Phi \models \rho(x) \doteq \mathit{ref}'$, for some actor ref'. In this case, the task m_1 is introduced in the queue of actor ref'. Here $\mathrm{fresh}(m_1(\bar{w})\{s_1;\})$ is a fresh renaming of the variables in m_1. We use $\Pi_{\rho(\bar{z})}\Phi$ to denote the projection of Φ on the variables $\rho(\bar{z})$, i.e., the constraints in Φ involving the input parameters $\bar{z}$. Note that the constraint Φ' is added to Φ_1 in actor ref' in order to store the relation between the formal and actual parameters of method m_1.
2. $(\mathsf{asy})_{\Phi_2}$: *Object x is compatible with objects in the state but $\rho(x)$ is not bounded in Φ.* If $\rho(x)$ is not bounded in Φ, this means either that $\rho(x) \notin \mathit{vars}(\Phi)$, or that $\Phi \not\models \rho(x) \doteq \mathit{ref}'$. Then we need to consider all possible aliasings with actors of compatible type that are in $\mathcal{S}$ whose actor identifiers have not been created using **new**, i.e., those whose actor identifier has the form s^D. For example, for the instructions $\mathsf{y} = \textbf{new}\ \mathsf{D}; \mathsf{x}\ !\ \mathsf{m_1}(\bar{\mathsf{z}})$ and assuming that x is not bounded, it is incorrect to bound variable x to y, as x must be a different reference. Then if $\mathcal{S}$ contains some actor of class D not created with **new**, then we can assume that $\rho(x)$ and such actor are aliased, and thus store the call in the queue of s_n^D and $\rho(x) \doteq \mathsf{s}_n^D$ in Φ. Function $\mathit{class}(x)$ returns the class of actor x.
3. $(\mathsf{asy})_{\Phi_3}$: *Actor x corresponds to an actor not yet created.* Then, a new actor is created, forcing $\rho(x)$ to be equals to it. Importantly, this situation requires that an actor of class D be in the initial state. Hence, a new identifier s_n^D corresponding to an actor not created with **new**, must be introduced in the set $\mathcal{I}$ in order to be able to reconstruct the initial state at the end of the computation. Note that rules $(\mathsf{asy})_{\Phi_2}$ and $(\mathsf{asy})_{\Phi_3}$ are both applicable under the same conditions what generates non-determinism in symbolic execution.



(0) s₀:{ft(n⁰)}
N⁰>M⁰
¬(N⁰>M⁰)
(1) s₀:{wk(n⁰₁, h⁰₁), dg(n⁰₂)}
N⁰₁≐N⁰, H⁰₁≐M⁰, N⁰₂≐N⁰−M⁰
(6) s₀:{wk(n⁰₁, h⁰₁), rp(x⁰)}
N⁰₁≐N⁰, H⁰₁≐N⁰, X⁰≐1
s₀.wk
s₀.dg
s₀.wk
s₀.rp
Tree₁
(2) s₀:{wk(n⁰₁, h⁰₁)}, r₁:{ft(n¹)}
Tree₁₁
Tree₁₂
W⁰≐r₁, T¹≐r₁, N¹≐N⁰₂, R¹≐1, B¹≐s₀, M¹≐M⁰
s₀.wk
r₁.ft
Tree₂
N¹>M¹
¬(N¹>M¹)
N¹₁≐N¹, H¹₁≐N¹, X¹≐1
(3) s₀:{wk(n⁰₁, h⁰₁)}, r₁:{wk(n¹₁, h¹₁), dg(n¹₂)}
(5) s₀:{wk(n⁰₁, h⁰₁)}, r₁:{wk(n¹₁, h¹₁), rp(x¹)}
N¹₁≐N¹, H¹₁≐M¹, N¹₂≐N¹−M¹
s₀.wk
r₁.wk
r₁.rp
Tree₈
Tree₉
Tree₁₀
s₀.wk
r₁.wk
r₁.dg
Tree₃
Tree₄
(4) s₀:{wk(n⁰₁, h⁰₁)}, r₁:{wk(n¹₁, h¹₁)}, r₂:{ft(n²)}
W¹≐r₂, T²≐r₂, N²≐N¹₂, R²≐1, B²≐r₁, M²≐M¹
s₀.wk
r₁.wk
r₂.ft
Tree₅
Tree₆
Tree₇

Fig. 4. Symbolic Execution for the Running Example

Example 2. Fig. 4 shows an excerpt of the symbolic execution tree of method ft. We write $\textbf{Tree}_i$, $1 \leq i \leq 12$, to denote partial execution trees, which are not shown due to space limitations. The nodes contain the actor identifiers and their queues of tasks in braces. A superscript in a variable corresponds to the identifier of the actor to which it belongs, e.g., R^2 refers to field r of actor $\mathtt{s}_2$. Subscripts are used to generate fresh variables consecutively. The initial field renaming in the root node is $\theta^0_\mathsf{s} = \{\mathsf{this}{\mapsto}\mathsf{T}^0, \mathsf{r}{\mapsto}\mathsf{R}^0, \mathsf{mx}{\mapsto}\mathsf{M}^0, \mathsf{b}{\mapsto}\mathsf{B}^0\}$, the constraint attached to $\mathtt{s}_0$ is $\Phi^0 = \{\mathsf{T}^0 \doteq \mathtt{s}_0\}$ and the initial renaming for local variables in $\mathsf{ft}(\mathsf{n}^0)$ is $\rho^0_\mathsf{s} = \{\mathsf{n}^0{\mapsto}\mathsf{N}^0\}$. The left branch from node (**0**) corresponds to the **if** instruction for the call $\mathsf{ft}(\mathsf{n}^0)$. The condition $\mathsf{n}^0{>}\mathsf{this.mx}$ produces the constraint $\rho^0_\mathsf{s}(\mathsf{n}^0){>}\theta^0_\mathsf{s}(\mathsf{mx})$, i.e., $\mathsf{N}^0{>}\mathsf{M}^0$. Since $\Phi^0 \models \theta^0_\mathsf{s}(\mathsf{this}){\doteq}\mathtt{s}_0$ holds, rule $(\mathsf{asy})_{\Phi_1}$ can be applied to both asynchronous calls (the applications of $(\mathsf{asy})_{\Phi_2}$ and $(\mathsf{asy})_{\Phi_3}$ will be illustrated in the tree in Fig. 5). For the call $\mathsf{this} \,!\, \mathsf{wk}(\mathsf{n}^0, \mathsf{this.mx})$, we generate $\mathsf{wk}(\mathsf{n}^0_1, \mathsf{h}^0_1)$ as fresh renaming for the method, together with the initial renaming $\rho^0_1{=}\{\mathsf{n}^0_1 \mapsto \mathsf{N}^0_1, \mathsf{h}^0_1 \mapsto \mathsf{H}^0_1\}$. Hence, the constraints $\mathsf{N}^0_1{\doteq}\mathsf{N}^0$ and $\mathsf{H}^0_1{\doteq}\mathsf{M}^0$ are added to the constraints for $\mathtt{s}_0$. Similarly, the constraint $\mathsf{N}^0_2{\doteq}\mathsf{N}^0{-}\mathsf{M}^0$ originates from using $\mathsf{dg}(\mathsf{n}^0_2)$ as renaming for $\mathsf{dg}(\mathsf{n}{-}\mathsf{this.mx})$. The right branch of node (**0**) is associated to the **else** of method ft. In this case, the renaming ρ^0_1 associated to task $\mathtt{s}_0.\mathsf{wk}$ maps n^0_1 (resp. h^0_1) to N^0_1 (resp. H^0_1). Similarly, the initial renaming ρ^0_2 for task $\mathtt{s}_0.\mathsf{rp}$ maps x^0 to X^0. Since in (**1**) we have two tasks, a new branching is required to try the two reorderings. The branch $\mathtt{s}_0.\mathsf{dg}$ executes the call to dg and thus, after applying rules $(\mathsf{new})_\Phi$ and $(\mathsf{asy})_{\Phi_1}$, a new actor $\mathtt{r}_1$ appears in (**2**) with a corresponding call to ft in its queue. The constraint $\mathsf{W}^0{\doteq}\mathtt{r}_1$ is added to the constraints of actor $\mathtt{s}_0$, where W^0 is the fresh renaming for variable wkr.

From (**2**), if we execute $\mathtt{r}_1$.ft, branches (**3**) and (**5**) are generated. From (**3**) the execution of $\mathtt{r}_1$.dg creates a new actor $\mathtt{r}_2$ (node (**4**)) as in (**2**). $\square$

4 Less Redundant Exploration in Symbolic Execution

Already in the context of dynamic execution, a naïve exploration of the search space to reach all possible system configurations does not scale. The problem is exacerbated in the context of symbolic execution due to the additional non-determinism introduced by the use of constraint variables instead of concrete values. There has been intensive work to avoid the exploration of redundant states, which lead to the same configuration. Partial-order reduction (POR) [9,11] is a general theory that helps mitigate the problem by exploring the subset of all possible interleavings, which lead to a different configuration. Concrete algorithms have been proposed in [3,10,22] for dynamic testing.

In this section, we adapt to the context of symbolic execution and improve the notion of temporal stability of an actor introduced in [3] to avoid redundant exploration. This notion states that, at a given state, if we first select a *temporarily stable actor*, i.e., an actor to which no other actors will post tasks, unless it executes, it is guaranteed that it is not necessary to try executions in which the other actors in the state execute before this one, thus, avoiding such redundant explorations. Note that a temporarily stable actor at a state, might become non-stable in a subsequent state if tasks are added to it after it executes again, hence the temporal nature of the property. This notion is of general applicability and can be used within the algorithms of [10,22]. The original notion of [3] is here extended to consider symbolic states and strengthened to allow the case in which an actor receives a task, which is *independent* of those in the queue of the actor. As it is well-known in concurrent programming [5], tasks t and t' *are independent* if t does not write in the shared locations that t' accesses, and viceversa. We say that t is independent of $\mathcal{Q}$, denoted as $indep(t,\mathcal{Q})$, if t and t' are independent for all $t' \in \mathcal{Q}$.

Definition 1 (temporarily stable actor). $\mathsf{ac}(ref, t, \theta, \mathcal{Q}, \Phi)$ *is* temporarily stable *in $\mathcal{S}_0$ iff, for any $\mathcal{E}$ starting from $\mathcal{S}_0$ and for any subtrace $\mathcal{S}_0 \xrightarrow{*}_{\Phi} \mathcal{S}_n \in \mathcal{E}$ in which the actor ref is not selected, we have $\mathsf{ac}(ref, t, \theta, \mathcal{Q}', \Phi) \in \mathcal{S}_n$ and for all $t' \in \mathcal{Q}'{-}\mathcal{Q}$ it holds that $indep(t', \mathcal{Q})$.*

Our goal is to define sufficient conditions that ensure actors stability and can be computed during symbolic execution. To this end, given a method m_1 of class C_1, we define $Ch(C_1{:}m_1)$ as the set of all chains of method calls of the form $C_1{:}m_1{\to}C_2{:}m_2{\to}\cdots{\to}C_k{:}m_k$, with $k{\geq}2$, s.t. $C_i{:}m_i \neq C_j{:}m_j$, $2{\leq}i{\leq}k{-}1$, $i \neq j$ and there exists a call within $bd(C_i{:}m_i)$ to method $C_{i+1}{:}m_{i+1}$, $1{\leq}i{<}k$. This captures all paths $C_2{:}m_2 \to C_{k-1}{:}m_{k-1}$, without cycles, that go from $C_1{:}m_1$ to $C_k{:}m_k$. The set $Ch(C_1{:}m_1)$ can be computed statically for all methods.

Theorem 1 (sufficient conditions for temporal stability). $\mathsf{ac}(ref^{C_n}, \_, \_, \mathcal{Q}, \_){\in}\mathcal{S}$*, is temporarily stable in $\mathcal{S}$, if for every $\mathsf{ac}(ref^{C_1}, \_, \_, \mathcal{Q}_1, \_) \in\mathcal{S}$, $ref^{C_n} \neq ref^{C_1}$ and for every $tk(\_, m_1, \_, \_){\in}\mathcal{Q}_1$, one of the following conditions holds:*

1. *There is no chain* $C_1{:}m_1 \to \cdots \to C_n{:}m_n \in \mathit{Ch}(C_1{:}m_1)$*; or*
2. *For all chains* $C_1{:}m_1{\to}\cdots{\to}C_n{:}m_n{\in}\mathit{Ch}(C_1{:}m_1)$*,* m_n *is independent of* $\mathcal{Q}$*; or*
3. *For all chains* $C_1{:}m_1 \to \cdots \to C_n{:}m_n{\in}\mathit{Ch}(C_1{:}m_1)$*, for all* $\mathsf{ac}(\mathit{ref}^{C_i}, \_, \theta, \mathcal{Q}_2, \Phi) \in \mathcal{S}$*,* $1{\le}i{\le}n{-}1$*, and for all* $\mathit{tk}(\_, \_, \rho, \_) \in \mathcal{Q}_2$*, it holds that* $\Phi{\cup}\theta(f){\doteq}\mathtt{r}^{C_n}$ *is unsatisfiable, for all* $f{\in}\mathcal{F}(C_i)$ *and* $\Phi{\cup}\rho(x){\doteq}\mathtt{r}^{C_n}$ *is unsatisfiable, for all* x *occurring in* $\mathit{vars}(\Phi){-}\mathcal{F}(C_i)$*.*

Intuitively, the theorem above ensures that no ref^{C_1} can modify the queue of ref^{C_n}. This is because (1) there is no transitive call from m_1 to any method of class C_n, or (2) if there is, the call is independent of those in ref^{C_n}, or (3) there are transitive (non-independent) calls from m_1 to some method of class C_n, but no reference to actor ref^{C_n} can be found.

Example 3. Node (**2**) has two actors and the initial mapping for $\mathtt{r}_1$ contains $\theta^1_{\mathsf{s}}(\mathsf{b}){=}\mathsf{B}^1$. Points 1 and 2 of Th. 1 does not hold, since from ft (in the queue of $\mathtt{r}_1$) there exists a call to rp and rp and wk are not independent. Besides, $\mathsf{B}^1{\doteq}\mathtt{s}_0$ occurs as constraint in (**2**) and thus Point 3 of Th. 1 neither holds. Hence, actor $\mathtt{s}_0$ is not temporarily stable. However, actor $\mathtt{r}_1$ is temporarily stable in (**2**), since task wk in the queue of $\mathtt{s}_0$ does not call any method of class Fact. This means that $\mathbf{Tree}_2$ in Fig. 4 is redundant and hence not expanded. A similar reasoning allows us to conclude that trees $\mathbf{Tree}_3$, $\mathbf{Tree}_5$, $\mathbf{Tree}_6$ and $\mathbf{Tree}_8$ are redundant. To illustrate the need of condition 2, consider a state with two actors $\mathtt{r}_0$, $\mathtt{r}_1$, with task dg resp. ft in the queue of $\mathtt{r}_0$ resp. $\mathtt{r}_1$ and no associated constraints. Then, for both actors neither condition 1 nor 3 in Th. 1 hold. However, since method dg is independent of the remaining methods of class Fact, condition 2 holds and both actors are temporarily stable. Finally note that, as explained in [3], $\mathbf{Tree}_1$ and $\mathbf{Tree}_4$ are detected redundant, since tasks wk and dg are independent. □

5 Generation of Test Cases

An important problem for the generation of test cases for a given method (without knowledge on the input values) is that the execution tree to be traversed by symbolically executing the method is in general infinite. Hence, it is required to fix a *coverage and termination criterion* (CTC) to guarantee that the number of paths traversed remains finite, while at the same time an interesting set of test-cases is generated.

5.1 Coverage and Termination Criteria for Actor Systems

Given a task executing on an actor, we can ensure its local termination by using existing CTC developed in the sequential setting. For instance, we can use the loop-count criteria [15], which limits the number of times we iterate on loops (and the number of recursive calls) to a threshold k_l. Other existing criteria defined for sequential programs would be valid as well. Unfortunately, the application of these CTC criteria to all tasks of a state does not guarantee termination of the whole TCG process. There are two factors that threaten termination: (1) we can

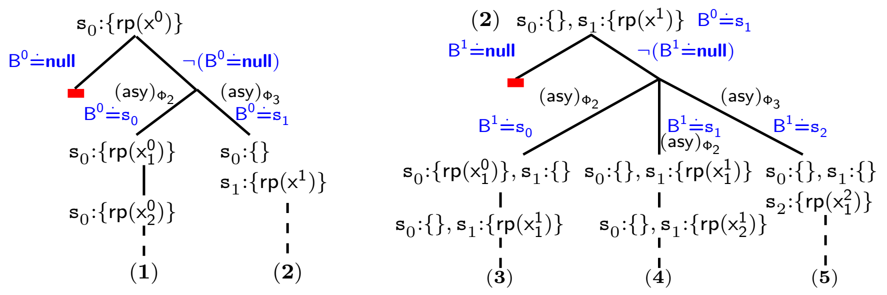


Fig. 5. Task-level and Object-number CTC Criteria

switch from one task to another an infinite number of times, (2) we can create an unbounded number of actors. The following example shows the first factor, a program for which the loop-k criterion does not guarantee termination unless we limit the number of task switches per actor.

Example 4. The execution tree for rp(x) in showed in two fragments in Fig. 5 (the right part corresponds to the execution from node (**2**) in the left part). The branch marked with (**1**) is infinite due to the task rp is continuously introduced and extracted from the queue of s_0. By limiting the number of task switches per actor it is possible to prune branch (**1**) and to continue the execution by exploring the branch corresponding to $B^0 \doteq s_1$. □

In our symbolic semantics, we can easily track the number of task switches per actor *ref* by counting the number of applications of rule $(\mathsf{mstep})_\Phi$ on *ref*.

Definition 2 (task-level CTC). *Let* $k \in \mathbb{N}^+$. *A symbolic execution* $\mathcal{E} \equiv \mathcal{S}_0 \square \mathcal{I}_0 \longrightarrow^*_\Phi \mathcal{S}_n \square \mathcal{I}_n$ *satisfies* the task-level CTC *iff for all actor ref*$\in \mathcal{S}_n$, *it holds that ref has been selected at most k times in* $\mathcal{E}$ *by rule* $(\mathsf{mstep})_\Phi$.

Even by limiting the number of task switches per actor, a second factor for non-termination arises when we create an unbounded number of actors for which, the number of task switches does not exceed the limit allowed. New actors arise when applying either $(\mathsf{new})_\Phi$ or $(\mathsf{asy})_{\Phi_3}$ in Fig. 3. Next example illustrates it.

Example 5. Using the task-level CTC, the branch marked with (**2**) in Fig. 5 can be explored. Such branch comes from the application of rule $(\mathsf{asy})_{\Phi_3}$, which generates a new actor s_1 when executing the asynchronous call this.b ! rp(this.r). The continuation of the branch is detailed in Fig. 5 (right). Branch (**3**) will be pruned by using the task-level CTC because field b in actor s_0 points to s_1 ($B^0 \doteq s_1$) and viceversa, such field points to s_0 in actor s_1 ($B^1 \doteq s_0$). Similarly, branch (**4**) is pruned by the task-level criteria, since the field b in actor s_1 points to s_1 ($B^1 \doteq s_1$). Branch (**5**) behaves differently to the others, since the application of rule $(\mathsf{asy})_{\Phi_3}$ generates a new actor in each execution step and thus the number of new actors grows infinitely. By annotating the instruction this.b ! rp(this.r) in method rp with a counter initialized to 0, it is possible to count the number of times that such instruction is executed. When such counter exceeds a fixed limit, the branch can be pruned. □

The idea of the next CTC is to limit the application of the instructions, which introduce new actors (**new** and $(\mathsf{asy})_{\Phi_3}$) to a threshold k_o. Such k_o cannot be global, since the key is not to limit the total number of created actors but instead the number of actors created at the same program point. Thus, we consider that program points at which actors are introduced in the state are *annotated* with a counter c_o initialized to 0. In particular, each **new** and asynchronous call have the form $\langle x = \textbf{new}\ C, c_o\rangle$, and $\langle x = y\ !\ m(\bar{z}), c_o\rangle$ respectively. When a task executes a **new** or $(\mathsf{asy})_{\Phi_3}$ instruction, the counter c_o associated to such instruction in the program is increased by one.

Definition 3 (actor-number CTC). *Let $k_o \in \mathbb{N}^+$. A symbolic execution $\mathcal{E}$ for an annotated program P satisfies the* actor-number CTC *iff for all instructions $\langle \_, c_o\rangle$ in P it holds that $c_o \leq k_o$.*

5.2 Test Cases for Actor Systems

The generation of *test cases* for a method $m(\bar{x})$ using the above CTC is as follows. We start the symbolic execution of $m(\bar{x})$ using the rules in Fig. 3 such that each derivation is expanded until (a) it is *complete* (i.e., all actors are idle and have empty queues) or (b) one of the CTC in Sec. 5.1 is not satisfied. In case (a), we produce a test case associated to the complete derivation, which defines the initial and final states of such execution. In the context of actor systems, the state is given by the constraints gathered along symbolic execution on the fields of the different actors, denoted as $\mathsf{fields}(\mathit{ref}, \Phi, \theta)$, where ref is the reference of the actor, Φ are the constraints for its field values and θ is the renaming relating constraint variables in Φ with fields. Besides, in the initial state we want to obtain also the constraints gathered for the arguments $\bar{x}$ of the method $m(\bar{x})$. We use the notation $\mathsf{args}(m(\bar{x}), \Phi, \rho)$ to denote the constraints Φ imposed on $\bar{x}$, together with the initial renaming ρ, which keeps the association between $\bar{x}$ and Φ. Due to the non-determinism in symbolic execution, the execution of $m(\bar{x})$ produces a symbolic tree such that a test case is obtained from each of its complete derivations (or branches). The following definition presents the notion of test case associated to a given complete derivation.

Definition 4 (test case). *Let $\mathcal{E} \equiv \mathcal{S}_0 \Box \mathcal{I}_0 \xrightarrow{*}_{\Phi} \mathcal{S}_n \Box \mathcal{I}_n$ be a complete symbolic execution such that $\mathcal{S}_0 \Box \mathcal{I}_0$ is an initial state, where $\mathcal{I}_0 = \{\langle \mathsf{s}_0^C, m(\bar{x}), \theta_\mathsf{s}, \rho_\mathsf{s}\rangle\}$. The test case for $\mathcal{E}$ is defined as the tuple $\langle \mathcal{A}_I, \mathcal{A}_O\rangle$, where:*

$$\begin{array}{lll}
\mathcal{A}_I & = \{\mathsf{args}(m(\bar{x}), \Phi_I, \rho_\mathsf{s}) & \mid \mathsf{ac}(\mathsf{s}_0^C, \_, \_, \_, \Phi) \in \mathcal{S}_n, \Phi_I = \Pi_{\rho_\mathsf{s}(\bar{x})}\Phi\} \cup \\
 & \{\mathsf{fields}(\mathsf{s}_0^C, \Phi_I, \theta_\mathsf{s}) & \mid \mathsf{ac}(\mathsf{s}_0^C, \_, \_, \_, \Phi) \in \mathcal{S}_n, \Phi_I = \Pi_{\theta_\mathsf{s}(\mathcal{F}(C))}\Phi\} \cup \\
 & \{\mathsf{fields}(\mathsf{s}_k^D, \Phi_I, \theta'_\mathsf{s}) & \mid \mathsf{ac}(\mathsf{s}_k^D, \_, \_, \_, \Phi) \in \mathcal{S}_n, \langle \mathsf{s}_k^D, \theta'_\mathsf{s}\rangle \in \mathcal{I}_n, \Phi_I = \Pi_{\theta'_\mathsf{s}(\mathcal{F}(D))}\Phi\} \\
\mathcal{A}_O & = \{\mathsf{fields}(\mathit{ref}_k^D, \Phi_O, \theta) & \mid \mathsf{ac}(\mathit{ref}_k^D, \_, \theta, \_, \Phi) \in \mathcal{S}_n, \Phi_O = \Pi_{\theta(\mathcal{F}(C))}\Phi\}
\end{array}$$

In the above definition, we can observe that the test cases are given in terms of the constraints in Φ. An essential aspect is that the renamings ρ_s and θ_s allow us to establish the relation between the names for fields and variables in the program and their corresponding constraint ones in order to generate a correct test case. In particular, the initial state of the test case $\mathcal{A}_I$ contains two types of information: (1) in args we store the information about the constraints gathered for the method arguments $\bar{x}$ that is obtained by projecting the constraints Φ on the original names for the input arguments that were stored in ρ_{s}, (2) in fields the constraints for the actor fields that are obtained by projecting Φ on the initial names for the actor fields that are stored in θ_{s}. The final state contains the constraints for the fields gathered in the final state of the computation and applying the renamings that have been computed in θ until the last state.

Example 6. Let us consider the TCG of method $\mathsf{ft}(\mathsf{n}^0)$ with limits 1, 5 and 2 resp. for the constants k in criteria loop-k, task-level and actor-number. The following two test cases $\mathcal{A}_1 = \langle \mathcal{A}_I, \mathcal{A}_O \rangle$ and $\mathcal{A}_2 = \langle \mathcal{A}_I, \mathcal{A}'_O \rangle$, are generated from two of the derivations in $\mathsf{\textbf{Tree}}_7$ of Fig. 4, where:

$$\begin{aligned}
\mathcal{A}_I &= \{\mathsf{args}(\mathsf{ft}(\mathsf{n}^0), \{\mathsf{N}^0 \doteq 3\}, \rho^0_{\mathsf{s}}), \mathsf{fields}(\mathsf{s}_0^{\mathsf{Fact}}, \{\mathsf{M}^0 \doteq 1, \mathsf{B}^0 \doteq \mathbf{null}\}, \theta^0_{\mathsf{s}})\}\\
\mathcal{A}_O &= \{\mathsf{fields}(\mathsf{s}_0^{\mathsf{Fact}}, \{\mathsf{M}^0_{\mathsf{a}} \doteq 1, \mathsf{B}^0_{\mathsf{b}} \doteq \mathbf{null}, \mathsf{R}^0_{\mathsf{c}} \doteq \mathsf{R}^0 {*} 3 {*} 2 {*} 1\}, \theta^0_{\mathsf{f}}),\ \mathsf{fields}(\mathsf{s}_1^{\mathsf{Fact}}, \{\mathsf{M}^1_{\mathsf{d}} \doteq 1,\\
&\qquad \mathsf{B}^1_{\mathsf{e}} \doteq \mathsf{s}_0^{\mathsf{Fact}}, \mathsf{R}^1_{\mathsf{f}} \doteq 2\}, \theta^1_{\mathsf{f}}), \mathsf{fields}(\mathsf{s}_2^{\mathsf{Fact}}, \{\mathsf{M}^2_{\mathsf{g}} \doteq 1, \mathsf{B}^2_{\mathsf{h}} \doteq \mathsf{s}_1^{\mathsf{Fact}}, \mathsf{R}^2_{\mathsf{i}} \doteq 1\}, \theta^2_{\mathsf{f}})\}
\end{aligned}$$

being $\mathcal{A}'_O$ as $\mathcal{A}_O$ but replacing the first entry for $\mathsf{s}_0^{\mathsf{Fact}}$ by $\mathsf{fields}(\mathsf{s}_0^{\mathsf{Fact}}, \{\mathsf{M}^0_{\mathsf{a}} \doteq 1, \mathsf{B}^0_{\mathsf{b}} \doteq \mathbf{null}, \mathsf{R}^0_{\mathsf{c}} \doteq \mathsf{R}^0 {*} 3 {*} 1 {*} 1\}, \theta^0_{\mathsf{f}})$. The renamings θ^0_{s} and ρ^0_{s} are defined in Ex. 2 and the remaining ones are defined as $\theta^i_{\mathsf{f}}(\mathsf{mx}) = \mathsf{M}^{\mathsf{i}}_{\_}$, $\theta^i_{\mathsf{f}}(\mathsf{b}) = \mathsf{B}^{\mathsf{i}}_{\_}$, $\theta^i_{\mathsf{f}}(\mathsf{r}) = \mathsf{R}^{\mathsf{i}}_{\_}$, where $1 \leq i \leq 2$ and "_" refers to the corresponding subindex. Note that test case $\mathcal{A}_2$ reveals the bug in the program, which is only observable when an intermediate actor in the chain of involved actors (in this case actor $\mathsf{s}_1^{\mathsf{Fact}}$), executes task rp before task wk, hence sending to its caller a partial result. $\square$

From the constraints in the test cases, it is possible to produce actual values by relying on standard *labeling* mechanisms. It is also straightforward to automatically generate xUnit unit tests [4].

Example 7. The following concrete test case is obtained from $\mathcal{A}_1$:

$$\begin{aligned}
\mathcal{T}_I &= \{\mathsf{args}(\mathsf{ft}(\mathsf{n}^0), \{\mathsf{n}^0 \doteq 3\}), \mathsf{fields}(\mathsf{s}_0^{\mathsf{Fact}}, \{\mathsf{mx} \doteq 1, \mathsf{b} \doteq \mathbf{null}, \mathsf{r} \doteq 1\})\}\\
\mathcal{T}_O &= \{\mathsf{fields}(\mathsf{s}_0^{\mathsf{Fact}}, \{\mathsf{mx} \doteq 1, \mathsf{b} \doteq \mathbf{null}, \mathsf{r} \doteq 3 {*} 2 {*} 1\}),\\
&\qquad \mathsf{fields}(\mathsf{s}_1^{\mathsf{Fact}}, \{\mathsf{mx} \doteq 1, \mathsf{b} \doteq \mathsf{s}_0^{\mathsf{Fact}}, \mathsf{r} \doteq 2\}), \mathsf{fields}(\mathsf{s}_2^{\mathsf{Fact}}, \{\mathsf{mx} \doteq 1, \mathsf{b} \doteq \mathsf{s}_1^{\mathsf{Fact}}, \mathsf{r} \doteq 1\})\}
\end{aligned}$$

In this case, only field r of actor $\mathsf{s}_0^{\mathsf{Fact}}$ has been labeled (with value 1). $\square$

6 Implementation and Experimental Evaluation

We have implemented all the techniques presented in the paper within the tool aPET [4], a test case generator for ABS programs, which is available at http://costa.ls.fi.upm.es/apet. ABS [16] 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 symbolic execution. The experiments have been performed using as benchmarks: (i) a set of classical actor programs borrowed from [18,21,22] and rewritten in ABS

	Ignoring task indep. info				Exploiting task indep. info				Reduction	
Benchm.	**Tests**	**Time**	**States**	**L/T/O**	**Tests**	**Time**	**States**	**L/T/O**	**Tests**	**Time**
QSort(2,_,1)	236	1934	2688	332/0/1052	236	1934	2688	332/0/1052	1.0x	1.0x
QSort(3,_,1)	1728	39084	44895	4719/0/20524	1728	39084	44895	4719/0/20524	1.0x	1.0x
QSort(2,_,2)	1017	21708	19300	3455/0/7928	1017	21825	19300	3455/0/7928	1.0x	1.0x
PSort(1,_,1)	478	696	1637	2/0/172	239	347	821	2/0/86	2.0x	2.0x
PSort(2,_,1)	3423	>200s	470087	0/0/182550	3425	>200s	470451	1/0/182649	1.0x	1.0x
PSort(1,_,2)	13678	19072	43341	2/0/4148	6839	9508	21673	2/0/2074	2.0x	2.0x
RSim(1,_,1)	9	8	25	1/0/2	4	5	14	1/0/2	2.2x	1.6x
RSim(2,_,1)	441	333	1350	1/0/12	14	20	80	1/0/8	31.5x	16.6x
RSim(2,_,2)	4111	3101	11841	1/0/12	59	82	340	1/0/8	69.7x	37.8x
DHT(1,4,1)	35	665	3179	733/1730/8	21	124	555	125/98/8	1.7x	5.4x
DHT(2,4,1)	97	8171	19018	2977/12639/24	55	2864	2332	349/651/24	1.8x	2.9x
DHT(1,5,1)	35	6425	30231	7065/17090/10	21	343	1623	369/226/10	1.7x	18.7x
DHT(1,5,2)	53	21092	98117	23119/57504/0	39	2615	12613	2879/3632/0	1.4x	8.1x
Mail(2,4,2)	161	1033	4540	654/5184/6	58	236	944	157/648/6	2.8x	4.4x
Mail(3,4,2)	400	12321	46760	2100/72090/24	232	1029	4310	291/3994/24	1.7x	12.0x
Mail(2,5,2)	161	4226	13756	654/9216/582	58	641	2096	157/1152/78	2.8x	6.6x
Mail(2,5,3)	161	4495	14908	660/10368/0	58	665	2240	163/1296/0	2.8x	6.8x
Cons(2,_,_)	15	10	30	1/0/0	9	7	19	1/0/0	1.7x	1.4x
Cons(3,_,_)	159	118	334	1/0/0	33	26	75	1/0/0	4.8x	4.5x
Cons(4,_,_)	3039	2562	6639	1/0/0	153	138	351	1/0/0	19.9x	18.6x
Prod(2,_,_)	29	30	52	10/0/0	17	16	31	6/0/0	1.7x	1.9x
Prod(3,_,_)	398	745	819	100/0/0	82	140	169	21/0/0	4.9x	5.3x
Prod(4,_,_)	9155	30268	20679	1636/0/0	465	1393	1041	85/0/0	19.7x	21.7x
Fact(2,4,2)	720	944	2430	59/0/278	270	451	1128	41/0/128	2.7x	2.1x
Fact(3,4,2)	1104	1425	3576	52/0/395	432	665	1664	38/0/171	2.6x	2.1x
Fact(2,3,2)	72	286	720	59/204/98	54	222	564	41/120/80	1.3x	1.3x
Fact(3,4,3)	3416	4704	11938	63/0/896	960	1668	4094	49/0/282	3.6x	2.8x

Table 1. Experimental evaluation (times in ms on an Intel Core i5 at 3.2GHz, 4GB)

from ActorFoundry, and, (ii) some ABS models of typical concurrent systems. Specifically, *QSort* is a distributed version of the Quicksort algorithm, *PSort* is a modified version of the sorting algorithm used in the dCUTE study [21], *RSim* is a server registration simulation, *DHT* is a distributed hash table, *Mail* is an email client-server simulation, *Cons* resp. *Prod* is the *consume* resp. *produce* method in the classical producer-consumer protocol, and, *Fact* is the distributed factorial in Fig. 2. All sources are available at the above website.

Table 1 shows the results of our experimental evaluation. For each benchmark, we perform the symbolic execution and TCG of its most relevant method(s) with different values for k of the criteria in Sec. 5.1 (resp. loop-k, task-level and actor-number), shown in parenthesis right after the benchmark name. We consider combinations so that we can observe the impact of each criterion in the overall process. E.g., for *QSort*, the impact of look-k is observed comparing executions with parameters $(2, \_, 1)$ and $(3, \_, 1)$; and the impact of actor-number comparing executions with parameters $(2, \_, 1)$ and $(2, \_, 2)$. An underscore indicates that it does not affect the computation, provided it is above a certain minimum (typically 1 or 2). Also, for each benchmark and combination, we perform the TCG both ignoring and exploiting the independency information among tasks. After the name and criteria parameters, the first (resp. second) set of columns show the results ignoring (resp. exploiting) task independency information. For each run, we measure: the number of obtained test cases (column *Tests*); the total time taken and number of states generated by the whole exploration (columns *Time* and *States*); and the number of explorations, which have been cut resp. by criteria loop-k, task-level and actor-number (column $L/T/O$).

A relevant point, which is not shown in the table, is that our sufficient condition for temporal stability is able to determine a stable actor in all states of all benchmarks except for some states in benchmark *PSort*. This demonstrates that our sufficient condition for stability is very effective also in symbolic execution. Another important point to observe is the huge pruning of redundant executions performed when the task independency information is exploited. Last two columns show the reduction in number of tests and TCG time obtained when exploiting task independency information. In general, the more complex the programs and the deeper the exploration, the bigger is the reduction.

7 Related Work and Conclusions

We have presented a novel approach to automate TCG for actor systems, which ensures *completeness* of the test cases w.r.t. several interesting criteria. In order to ensure completeness in a concurrent setting, the symbolic execution tree must consider all possible task interleavings that could happen in an actual execution. The coverage criteria prune the tree in several dimensions: (1) limiting the number of iterations of loops at the level of tasks, (2) limiting the number of task switches allowed in each concurrency unit and (3) limiting the number of concurrency units created. Besides, our TCG framework tries to avoid redundant computations in the exploration of different orderings among tasks. This is done by leveraging and improving existing techniques to further reduce explorations in dynamic testing actor systems to the more general setting of static testing. Most related work is developed in the context of dynamic testing. The stream of papers devoted to further reduce the search space [1,10,18,22] is compatible with our work and the TCG framework can use the same algorithms and techniques, as we showed for the actor's stability of [3]. Dynamic symbolic execution consists in computing in parallel with symbolic execution a concrete test run. In [13] a dynamic symbolic execution framework is presented, however, there is no calculus for symbolic execution. In particular, the difficulties of handling asynchronous calls and the constraints over the field data are not considered.

Acknowledgments. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006.

References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. POPL'14, pp. 373–384*. ACM, 2014.

2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.
3. E. Albert, P. Arenas, and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *Proc. FORTE'14*, LNCS 8461, pp. 49-65. Springer, 2014.
4. E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In *Proc. ESEC/FSE'13*, pp. 595–598. ACM, 2013.
5. G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.
6. L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.
7. F. Degrave, T. Schrijvers, and W. Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *Proc. LOPSTR'09*, LNCS 6037, pp. 128–142. Springer, 2010.
8. C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In *Proc. TAP'07*, LNCS 4454, pp. 169–188. Springer, 2007.
9. J. Esparza. Model Checking Using Net Unfoldings. *SCP*, 23(2-3):151–195, 1994.
10. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proc. POPL'05, pp. 110-121.* ACM, 2005.
11. P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proc. CAV'91*, LNCS 531. pp. 176-185. Springer, 1991.
12. A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Proc. CL'00*, LNAI 1861, pp. 399–413. Springer, 2000.
13. A. Griesmayer, B. K. Aichernig, E. B. Johnsen, and R. Schlatte. Dynamic Symbolic Execution of Distributed Concurrent Objects. In *Proc. FMOODS/FORTE'09*, LNCS 5522, pp. 225-230. Springer, 2009.
14. P. Haller and M. Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
15. W.E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.
16. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMCO'10 (Revised Papers)*, LNCS 6957, pp. 142-164. Springer, 2012.
17. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.
18. S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In *Proc. FASE'10*, LNCS 6013, pp. 308-322. Springer, 2010.
19. C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *STVR*, 11(2):81–96, 2001.
20. R. A. Müller, C. Lembeck, and H. Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *Proc. IASTEDSE'04, pp. 365–371.* ACTA Press, 2004.
21. K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *Proc. FASE'06*, LNCS 3922, pp. 339-356. Springer, 2006.
22. S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Proc. FMOODS/FORTE'12*, LNCS 7273, pp. 219-234. Springer, 2012.
23. N. Tillmann and J. de Halleux. Pex: White Box Test Generation for .NET. In *Proc. TAP'08*, LNCS 4966, pp. 134-153. Springer, 2008.