



Project N°: **FP7-610582**  
Project Acronym: **ENVISAGE**  
Project Title: **Engineering Virtualized Services**  
Instrument: **Collaborative Project**  
Scheme: **Information & Communication Technologies**

## Deliverable D3.5 Test Case generation

Date of document: T30



Start date of the project: **1st October 2013**

Duration: **36 months**

Organisation name of lead contractor for this deliverable: **UCM**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

# Executive Summary:

## Test Case generation

This document summarises deliverable D3.5 of project FP7-610582 (**Envisage**), a Collaborative Project supported by the 7th Framework Programme of the EC. within the Information & Communication Technologies scheme. Full information on this project is available online at <http://www.envisage-project.eu>.

This deliverable of nature “prototype” consists of two main parts:

- The first part describes the *basic concepts* which are necessary for understanding the techniques used for testing the abstract behavioral models developed in **Envisage**, including the concepts of dynamic and static testing, and deadlock-guided testing.
- The second part provides end user documentation for using the two testing tools developed in the project: the **SYCO** tool that is used for dynamic (optionally deadlock-guided) testing and the **aPET** tool that is used for static testing and test case generation by means of symbolic execution.

## List of Authors

Elvira Albert (UCM)  
Puri Arenas (UCM)  
Miguel Gómez-Zamalloa (UCM)  
Miguel Isabel (UCM)

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Basic Concepts</b>	<b>6</b>
2.1	Dynamic Testing . . . . .	6
2.1.1	The SYCO Tool . . . . .	6
2.2	Deadlock-guided Testing . . . . .	7
2.3	Static Testing . . . . .	8
2.3.1	The aPET Tool . . . . .	9
<b>3</b>	<b>End User Documentation</b>	<b>10</b>
3.1	Use of SYCO . . . . .	10
3.1.1	Parameters of SYCO . . . . .	17
3.1.2	Deadlock-guided Testing with SYCO . . . . .	17
3.2	Use of aPET . . . . .	19
3.2.1	Parameters of aPET . . . . .	21
<b>4</b>	<b>Conclusions</b>	<b>24</b>
	<b>Bibliography</b>	<b>24</b>
	<b>Glossary</b>	<b>27</b>
<b>A</b>	<b>Book Chapter “Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency”, [5]</b>	<b>28</b>
<b>B</b>	<b>Article “Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing”, [3]</b>	<b>76</b>
<b>C</b>	<b>Article “Test Case Generation of Actor Systems”, [4]</b>	<b>94</b>
<b>D</b>	<b>Article “Testing Abstract Behavioral Specifications”, [24]</b>	<b>111</b>
<b>E</b>	<b>Tool Demo “SYCO: A Systematic Testing Tool for Concurrent Objects”, [7]</b>	<b>126</b>
<b>F</b>	<b>Article “Combining Static Analysis and Testing for Deadlock Detection”, [2]</b>	<b>132</b>

# Chapter 1

## Introduction

Writing correct concurrent programs is harder than writing sequential ones, because with concurrency presents 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., [23] and its references). Furthermore, the order in which tasks are selected for execution can be affected by different scheduling policies, and thus the initial state when resuming a task can be different by adopting one policy or another. As a result, computation is often non-deterministic and multiple (possibly different) solutions can be produced depending on the interleaved tasks and the scheduler.

One of the main challenges when testing concurrent programs, that we have faced in **Envisage**, is to reduce as much as possible the exploration of the search space without losing solutions. Partial-order reduction (POR) [14] is a well-known theory to tackle this problem, focusing on exploring the subset of all possible interleavings which lead to different solutions. In Section 2.1, we overview the techniques that we have proposed to reduce state exploration of **ABS** programs. These techniques are implemented within the **SYCO** tool that is described in Section 3.1. Besides, **SYCO** can be used to find deadlock traces (or discard the absence of them). This approach is described in Section 2.2. Finally, in addition to improving POR methods in the context of dynamic testing, we have also applied them in static testing using symbolic execution techniques. This generalization is described in Section 2.3. Since the symbolic execution tree is in general infinite, a termination criterion must be imposed to ensure its finiteness. In this section, a termination and coverage criterion is defined for **ABS** programs. The **aPET** tool, described in Section 3.2, implements static testing and incorporates the aforementioned coverage criterion.

The technical details which describe the dynamic and static methods used for testing **ABS** models can be found in the following papers, which have been published along the duration of the **Envisage** project, and that are attached in the Appendix:

- **SFM’14** [5]: This tutorial paper provides a comprehensive description of a symbolic execution mechanism used for static testing, and the main extensions performed in a test case generation tool for sequential programs in order to extend it to testing **ABS** models.
- **FORTE’14** [3]: This work presents new mechanisms and strategies for effectively testing **ABS** models. A relevant aspect of this work is that the new techniques can be used in combination with existing algorithms proposed for systematic testing and model checking.
- **ATVA’15** [4]: We extend the approach for dynamic testing of [3] to the context of static testing. This allows us to achieve effectiveness as in the dynamic context and have an engine for test case generation.
- **STTT’15** [24]: This article shows by means of a case study (developed by Fredhopper) how the test

case generation process is performed on **ABS** models and how it can be combined with other testing and runtime-checking methodologies.

- **CC'16** [7]: This tool demonstration paper overviews the **SYCO** tool for testing systematically **ABS** models.
- **iFM'16** [2]: Our most recent work guides the testing process towards deadlock traces so that we can provide a detailed description of the task scheduling and program state in deadlock executions. For this, we use a static deadlock analyzer which provides potential deadlock cycles that are used by the testing tool to discard deadlock-free paths.

# Chapter 2

## Basic Concepts

### 2.1 Dynamic Testing

Dynamic testing consists in executing the application under test on concrete input values or on a set of test cases. In the context of concurrent and parallel programs, a single dynamic execution with a concrete test case can be of little value, since the execution is usually non-deterministic, possibly producing different outputs depending on the interleavings of the involved processes. A thorough testing process must therefore systematically explore all non-deterministic interleavings that the concurrent execution may have - any of the interleavings may reveal the erroneous behavior. This is known as *systematic testing* [8, 23] in the context of concurrent programs.

In the execution of **ABS** concurrent objects, two sources of non-determinism can be distinguished:

- *Object-selection*, i.e., the selection of which object executes and;
- *Task-selection*, the selection of the task within the selected object.

Thanks to the non-preemptive scheduling and the absence of shared memory among different objects, it suffices to consider the above two levels of non-determinism only at *release* points, in order not to lose any behavior of the program. Compared to other models of concurrency this alleviates the state space explosion. However, a naïve systematic exploration of all possible selections usually does not scale. Two different families of techniques can help in mitigating such a state explosion:

1. *Partial-order reduction* (POR): It is well-known that many different derivations are often redundant and are guaranteed to produce the same results. POR [18, 13] is a general theory that allows characterizing derivations in *equivalence classes*. State-of-the-art POR algorithms are able to detect redundant derivations dynamically during the execution, and, allow generating only one derivation per equivalence class, avoiding a considerable number of redundant explorations.
2. *Guided testing*: A complementary approach to POR is to focus the search towards specific behaviors of the model, avoiding, as much as possible, the exploration of derivations leading to non-interesting behaviors. A particular case of this, which has been subject of our research in this task, is *deadlock-guided testing*, where the execution is driven towards potentially deadlock paths (while other paths are pruned). This is further elaborated in Section 2.2.

#### 2.1.1 The SYCO Tool

The first prototype reported in this deliverable is the **SYCO** tool, a systematic tester for **ABS** concurrent objects. Figure 2.1 shows the main architecture of **SYCO**. Boxes with dash lines are internal components of **SYCO** whereas boxes with regular lines are external components. The user interacts with **SYCO** through its web interface which is integrated within the **ABS** collaboratory and is hence provided by *EasyInterface* [16]. The

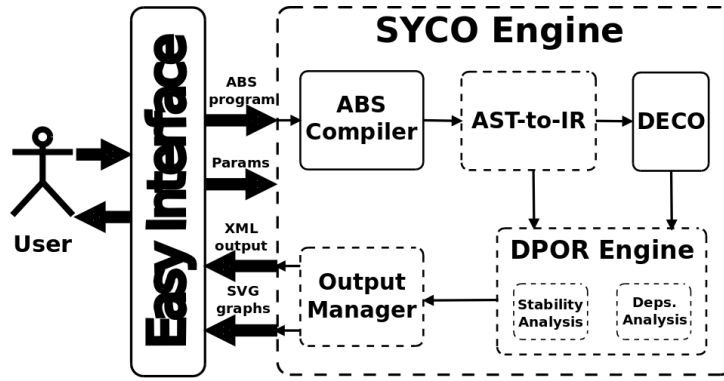


Figure 2.1: SYCO architecture

SYCO engine receives an ABS program and a selection of parameters. The ABS *compiler* compiles the program into an abstract-syntax-tree (AST) which is then transformed into the SYCO intermediate representation (IR). The DPOR *engine* carries out the actual systematic testing process. It comprises the ABS semantics, the DPOR algorithm of [3] and the *stability* and *dependencies* analyses of [3]. The *output manager* then generates the output in the format which is required by EasyInterface, including an XML file containing all the EasyInterface commands and actions and SVG diagrams. In case deadlock-guided testing is applied, the DECO *deadlock analyzer* [15] is invoked, which returns a set of potential deadlock cycles that are then fed to the DPOR engine to guide the testing process (discarding non-deadlock executions) [2] (see Section 2.2). Let us note that other actor-based languages with similar features could be handled by SYCO just by providing a transformation to the SYCO IR.

The web interface of SYCO is available both at the ABS collaboratory site and also at `costa.ls.fi.upm.es/syco`. Section 3.1 details its usage. Essentially, once the input program is ready, either selected from the available library of ABS programs or supplied by the user, the SYCO engine is run (with the selected settings) and the output is obtained. As a result, SYCO outputs a set of executions. For each one, SYCO shows the output state and the sequence of tasks/interleavings and concrete instructions of the execution (highlighting the source code). SYCO also generates sequence diagrams for each execution. Such sequence diagrams provide graphical and more comprehensive representations of execution traces. Essentially, they show the task/object executing at each time of the simulation, the spawned asynchronous calls (with arrows from caller to callee), and, the waiting and blocking dependencies. See Section 3.1 for details.

## 2.2 Deadlock-guided Testing

In concurrent programs, deadlock is one of the most common programming errors and, thus, a main goal of verification and testing tools for concurrent programs is, respectively, proving deadlock freedom and deadlock detection. Since the execution of concurrent programs is non-deterministic, all combinations of process interleavings must be considered as any of them might lead to a deadlock. Static analysis and testing are two different ways of detecting deadlocks that often complement each other and thus it seems quite natural to combine them. As static analysis examines all possible execution paths and values of variables, it can reveal deadlocks that could not manifest until weeks, months or years after releasing the application. This aspect of static analysis is especially important in security assurance, because security attacks try to exercise an application in unpredictable and untested ways. However, when a deadlock is found, state-of-the-art analysis tools [6, 17, 10, 22] often provide insufficient information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and objects), it is essential to know the task interleavings that have occurred and the objects involved in the deadlock, i.e., provide a concrete deadlock trace that allows the programmer to identify and fix the problem. In contrast, testing consists in executing the application for concrete input values. The primary advantage of testing for deadlock detection is that it can provide the deadlock trace with all information that the user needs in order to fix the problem. There

are two shortcomings though:

- Since not all inputs can be tried, there is no guarantee of deadlock freedom.
- Although recent research on POR tries to avoid redundant exploration as much as possible [14, 23, 8, 1, 3], the search space (even with POR) can be huge.

This is a threat to the application of testing in concurrent programming. We have proposed in [2] a seamless combination of static analysis and testing for effective deadlock detection. The basic idea is the following: (1) an existing static deadlock analysis [15] is used to obtain abstract descriptions of potential deadlock cycles. (2) Such potential deadlock cycles are then used to guide the systematic testing process in order to find associated deadlock traces (or discard them). This hybrid static analysis/testing technique has been studied in the context of Task 3.4 where its theoretical foundations will be reported. The technique has been implemented and is available within the **SYCO** tool. Details about its usage are found in Section 3.1.2.

## 2.3 Static Testing

Static testing comprises the set of testing techniques in which the code under test is not really executed. This includes techniques performed by humans, like *code reviews* and *inspections*, and automatic or semi-automatic techniques which make use of other programs or analyzers. One of the standard techniques for automatic static testing is *symbolic execution* [9, 11, 12, 19, 20, 21], in which the program execution is simulated for possibly unknown inputs hence using symbolic expressions for program variables. As a result, it produces a system of constraints over the inputs containing the conditions to execute the different paths and the expressions computed for their outputs. For instance, consider the method:

```
Int absValue(Int x) {
    if (x < 0) return -x;
    else return x;
}
```

The outcome is the set  $\{\langle x < 0, y = -x \rangle, \langle x \geq 0, y = x \rangle\}$ , where  $y$  refers to the return value. E.g., the first element can be read as: if the program is executed with an input  $x < 0$ , then the output is  $y = -x$ .

Symbolic execution has many applications, namely *software verification*, *program comprehension* and automatic *test case generation* (TCG). In this latter context, symbolic execution produces, by construction, a (possibly infinite) set of test cases, which satisfy the *path-coverage* criterion.

In the context of symbolic execution of concurrent programs, the problem of the non-deterministic interleavings of processes mentioned in Section 2.1, is added to the intrinsic non-determinism of symbolic execution due to branching statements involving partially unknown data. It is therefore crucial to apply aggressive POR techniques, and in many cases in practice, even to lose some interleavings and thus possibly sacrifice full path-coverage. To this aim, [4] extends the POR techniques of [3] to the context of symbolic execution and TCG.

On the other hand, in order to ensure finiteness of the process, and, at the same time, obtain a meaningful set of test-cases, [4] proposes coverage criteria for concurrent objects consisting on limiting the number of:

- iterations of loops at the level of tasks;
- task switches allowed in each concurrent object and;
- concurrency units originated during symbolic execution per program point.

The theoretical aspects of our symbolic execution engine and our TCG framework can be found in [5, 4].



### 2.3.1 The aPET Tool

The second prototype reported in this deliverable is the **aPET** tool, a symbolic execution-based test case generator for **ABS** concurrent objects. The architecture of **aPET** is essentially the same as that of **SYCO**. Indeed, both tools share most of their components, namely the **ABS** compiler, the AST-to-IR and part of the DPOR Engine and Output Manager. The main differences are that the internal engine of **aPET** includes support for symbolic execution and its termination criteria, and that the output manager includes support for TCG in different formats. The user interacts with **aPET** through its web interface (available at the **SYCO** website) which is also integrated within the **ABS** collaboratory and is hence provided by EasyInterface.

The usage of **aPET** is essentially as follows: given an input program and a selection of methods, the **aPET** symbolic execution engine computes a set of test cases for the selected methods. Test cases can be given as path constraints or, after a constraint solving procedure, as concrete test cases. Each test case includes the input arguments and input state, and the output argument and output state. Section 3.2 details how to use **aPET** with screenshots and provides information about the different parameters which can be set.

## Chapter 3

# End User Documentation

This chapter presents the user manuals of **SYCO** and **aPET**. Both tools are integrated with the **Envisage** collaboratory. In order to start **SYCO** (resp. **aPET**) we select **Systematic testing (SYCO)** (resp. **Test case generation (aPET)**) from the pull-down menu of available tools as shown in Fig. 3.1. The main window of the **Envisage** collaboratory web interface comprises five components:

- *File manager*: which contains a list of predefined examples and/or files uploaded by the user;
- *Code area*: where the user can edit the selected program or write a new one;
- *Outline view*: which includes an outline (list of classes, methods, etc.) of the selected file and module;
- *Toolbar*: which includes several buttons to execute the main actions; and
- *Console view*: where the results and information of the execution is printed.

Let us go to the file manager, located at the left-hand side and open the folder **syco\_examples**. The subfolder **Exhaustive\_execution** contains the code of the running example shown in Figure 3.2. Method **fact** of class **Fact** computes the factorial of a number **n** in a distributed way so that each involved object computes at most **h** multiplications. Let us suppose object **o** of class **Fact** is asked to compute the factorial of **n** by means of a call **o ! fact(n)**. Object **o** executes the task **work(n, o.maxH)** computing  $n*(n-1)*\dots*(n-o.maxH+1)$ . Afterwards, the call **delegate(n-o.maxH)** *delegates* the rest of the computation to another object. When an object is asked to compute the factorial of some **n**, smaller than its **maxH**, then the call **this ! work(n, n)** computes directly the factorial of **n** and the result is *reported* to its caller by task **report**. The result is then reported back to the initial object in a chain of **report** tasks using field **boss**, which stores the caller object. The computed result of each object is stored in field **r**. The provided **main** block just creates a runner object **r** and calls **r ! fact(5,2)** to compute the factorial of 5, which will be stored in field **r** of the initial **Fact** object. The expected result is hence 120. As we show later, the program has a bug, which is only exploited in a concrete sequence of interleavings when at least three objects are involved.

If we click over **Fact.abs**, the code of the running example appears at the code area. Now, if we press button **Refresh Outline**, the right-hand side with the class and module information is updated. The **Clear** button cleans the console area. Optionally, the parameters of the selected testing tool can be configured by clicking on **Settings** (details are given in Sections 3.1.1 and 3.2.1 for **SYCO** and **aPET** respectively). To execute the selected tool it is enough to click **Apply** in the pulldown menu on the tool bar and the results are presented in the console area.

### 3.1 Use of SYCO

Let us perform a systematic testing of our running example with **SYCO** using default parameters. We just select **SYCO** and press **Apply**. Note that systematic testing always targets the **main** block. Therefore, the selection made in the outline view is ignored. The results are printed in the console area.

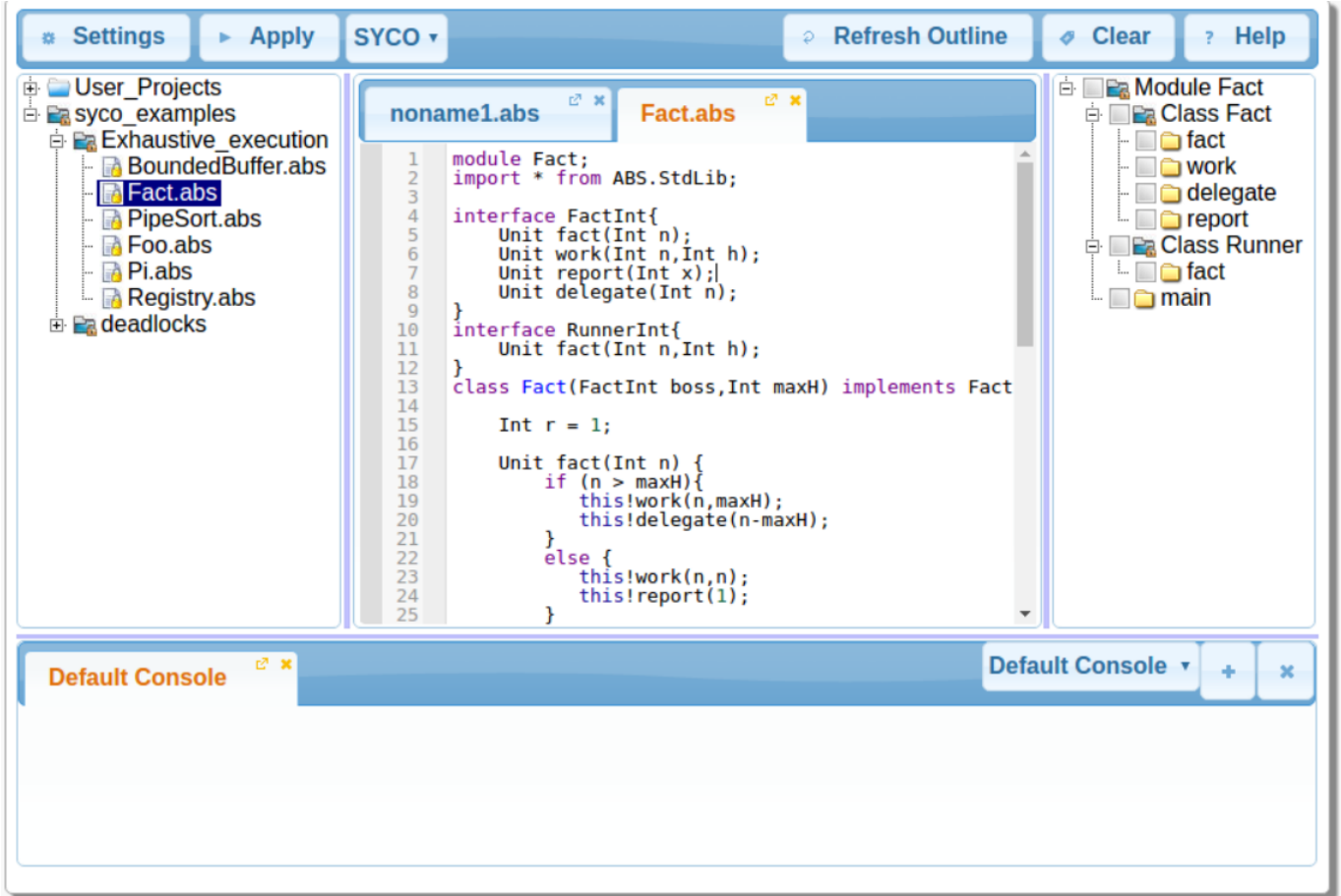


Figure 3.1: Collaboratory web Interface

SYCO first prints the number of complete executions explored (in this case eight executions). Note that, by default, the most aggressive POR is applied. As we will see later, the number of executions without POR is 280. Then, it prints the output state and the execution trace. The output state (in blue color) contains all the objects created during the execution. Each object is represented as a term with three arguments: the object identifier, the object type or class, and the final values of the object fields. For instance:

```
|-----object(2,'Fact',[field(boss,null),field(maxH,2),field(r,20)])
```

means that the final state contains an object identified by 2 of class `Fact`, whose fields `boss`, `maxH` and `r` have `null`, 2 and 20 as values. Since we are computing the factorial of 5, and this object is the initial object, its `r` field should end with value 120, instead of the obtained 20. This execution therefore reveals a bug in the program

The execution trace (in red color) shows, for each time or macro-step of the execution, the object and task executing at this time. If we click one time of the trace, the corresponding line in the source code is highlighted (in yellow color) in the code area. This is shown in Figure 3.3 where the first time (`|-----'Time: 0, Object: main, Task: 0:main'`) of the trace has been clicked.

To see the sequence diagram of a concrete execution we click the text ‘Click here to see the sequence diagram’ (next to the execution number in the console view). Figure 3.4 shows the sequence diagram of the first execution for our running example. At the left-hand side, a timeline is shown with the times of the execution, in this case 13 times (0 – 12). Each vertical cluster corresponds to the activities performed by each object, and each node corresponds to the task executing at the corresponding object in the corresponding time. Objects are of the form `class_id`, where `class` is the object type and `id` is a unique object identifier. Tasks are of the form `id:method` where `id` is a unique task identifier and `method` is the

<pre> <b>interface</b> FactInt {     <b>Unit</b> fact(Int n);     <b>Unit</b> work(Int n, Int h);     <b>Unit</b> report(Int x);     <b>Unit</b> delegate(Int n); } <b>interface</b> RunnerInt {     <b>Unit</b> fact(Int n, Int h); } <b>class</b> Fact(Fact boss, int maxH) <b>implements</b> FactInt {     Int r = 1;     <b>Unit</b> fact(Int n) {         <b>if</b> (n &gt; this.maxH) {             this ! work(n,this.maxH);             this ! delegate(n-this.maxH);         } <b>else</b> {             this ! work(n,n);             this ! report(1);         }     }     <b>Unit</b> delegate(Int n) {         FactInt worker = <b>new</b> Fact(this,this.maxH);         worker ! fact(n);     } } </pre>	<pre> <b>Unit</b> work(Int n,Int h) {     <b>while</b> (h &gt; 0) {         this.r = this.r * n;         n = n - 1;         h = h - 1;     } } <b>Unit</b> report(Int x) {     this.r = this.r * x;     <b>if</b> (this.boss != <b>null</b>) this.boss ! report(this.r); } <b>class</b> Runner <b>implements</b> RunnerInt {     <b>Unit</b> fact(Int n, Int h) {         FactInt f = <b>new</b> Fact(<b>null</b>,h);         f ! fact(n);     } } // main block {     RunnerInt r = <b>new</b> Runner();     r ! fact(5,2); } </pre>
--	---

Figure 3.2: Running Example

name of the method. Nodes also indicate why the execution of the associated task stopped. Nodes in green color labeled with **return** correspond to tasks that have finished their executions; nodes in orange color labeled with **waiting for taskId** are tasks which have been suspended waiting for task **taskId**; and nodes in red color labeled with **blocked for taskId** are tasks which block the object waiting for task **taskId**. Finally, arrows from nodes to clusters indicate asynchronous calls or object creations.

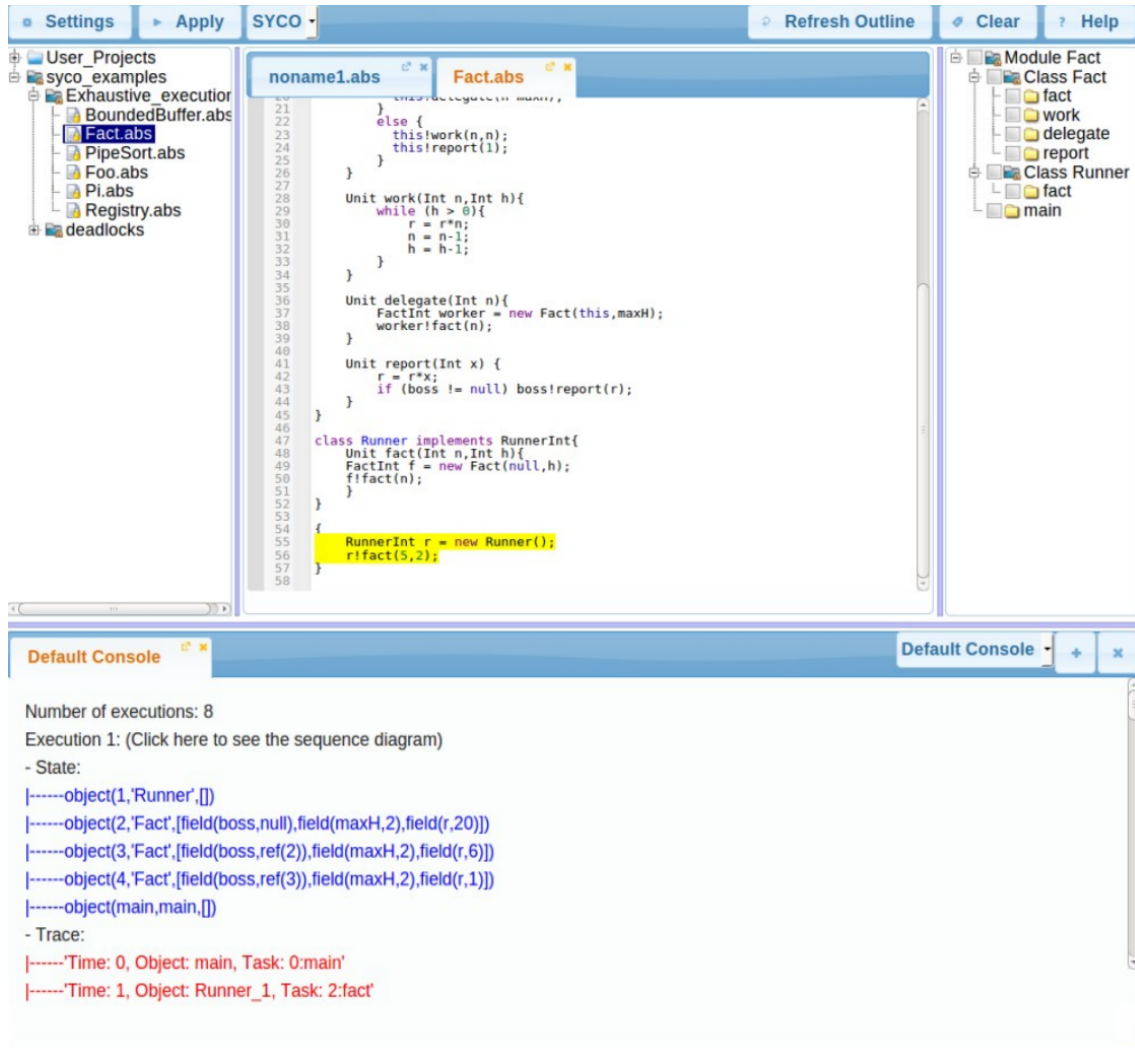


Figure 3.3: Execution of SYCO with default parameters

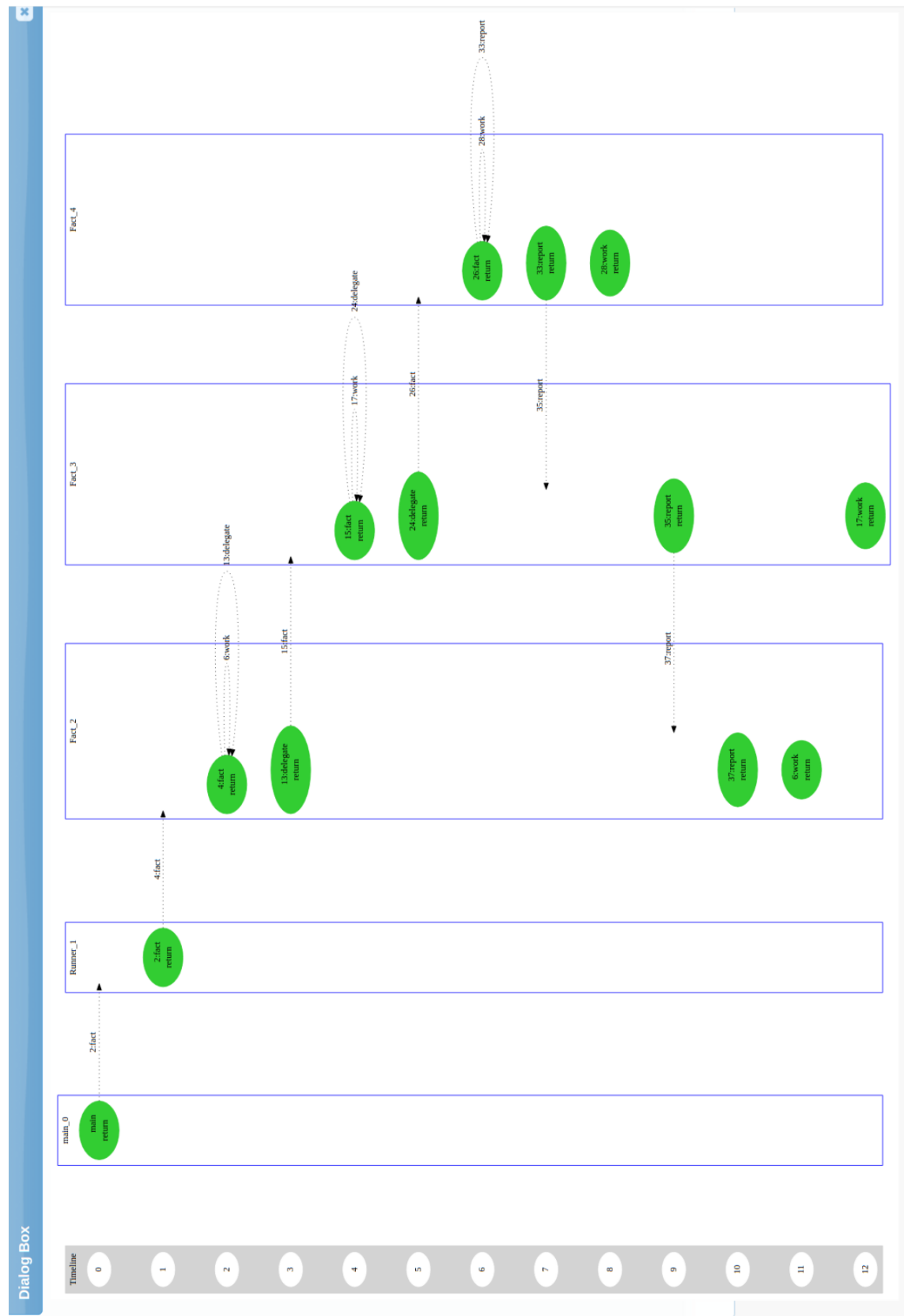


Figure 3.4: A buggy execution trace for the running example

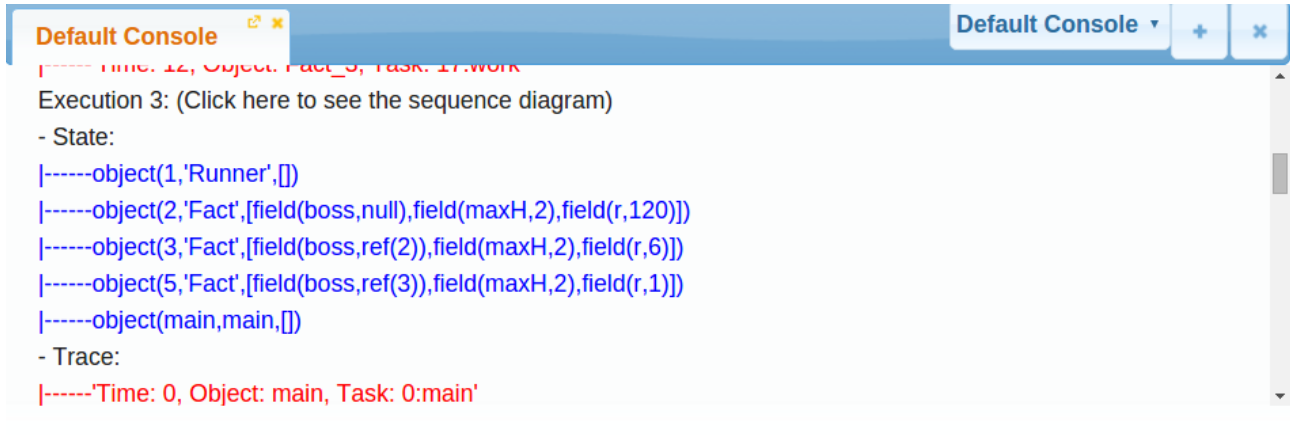


Figure 3.5: A correct execution for the running example

In our running example, the trace corresponding to execution 1 is shown in Figure 3.4. Let us briefly explain the diagram and the relations among the diagram, the code of the program (Figure 3.2) and the final state computed for execution 1 (Figure 3.3 below). Time 0 corresponds to the execution of the `main` block within the object identified as `main_0`. It creates a new object `Runner_1` (`RunnerInt r = new Runner()`) and spawns task 2:fact (`Runner_1 ! fact(5,2)`). In the final state, this adds the objects `object(main,main,[])` and `object(1,'Runner',[])` respectively. Then, the block `main_0` finishes its execution and it is marked with `return`. During time 1, object `Runner_1` executes the task 2:fact which creates the new object `Fact_2` (`FactInt f = new Fact(null,h)`), spawns task 4:fact to compute `Fact_2 ! fact(5)` and finishes. The new created object `object(2,'Fact',[field(boss,null),field(maxH,2),...])` appears in the final state. At time 2, `Fact_2` spawns tasks 6:work (`Fact_2 ! work(5,2)`) and 13:delegate (`Fact_2 ! delegate(3)`) and finishes. At time 3, the execution of task 13:delegate creates a new object `Fact_3` (`FactInt worker = new Fact(Fact_2,Fact_2.maxH)`) and spawns task 15:fact which corresponds to the computation of `Fact_3 ! fact(3)`. The execution of 13:delegate finishes and the corresponding green node is marked with `return`. The new object `object(3,'Fact',[field(boss,ref(2)),field(maxH,2),...])` appears in the final state. Time 4 is similar to time 2, but executing task 15:fact. Time 5 is similar to time 3, but executing 24:delegate, which creates the new object `Fact_4`, also appearing in the final state as `object(4,'Fact',[field(boss,ref(3)),field(maxH,2),...])`. At time 6, the execution of 26:fact, which corresponds to the execution of `Fact_4 ! fact(1)`, spawns tasks 28:work (`Fact_4 ! work(1,1)`) and 33:report (`Fact_4 ! report(1)`). At time 7, the execution of 33:report spawns task 35:report on object `Fact_3`, i.e., `Fact_3 ! report(1)`. At time 8, the execution of task 28:work finishes and the final state for object `Fact_4` is `object(4,'Fact',[field(boss,ref(3)),field(maxH,2),field(r,1)])`. At time 9, task 35:report spawns task 37:report on object `Fact_2` (`Fact_2 ! report(1)`). Time 10 executes 37:report and finishes since the field `boss` of `Fact_2` is `null`. At times 11 and 12, tasks 6:work (`Fact_2 ! work(5,2)`) and 17:work (`Fact_3 ! work(3,2)`) are executed completely, and thus the final states for objects `Fact_2` and `Fact_3` are, respectively, `object(2,'Fact',[field(boss,null),field(maxH,2),field(r,20)])` and `object(3,'Fact',[field(boss,ref(2)),field(maxH,2),field(r,6)])`.

Figures 3.5 and 3.6 show the result and sequence diagram of the third execution, in which we can observe that the expected value 120 is obtained. If we make a comparison between the sequence diagrams of executions 1 and 3, we can figure out that the problem in execution 1 originates on time 9, where the result is reported before executing task 17:work. In the sequence diagram of execution 3 we can observe that object `Fact_3` reports the result after executing task 17 : work (see times 5 and 10).

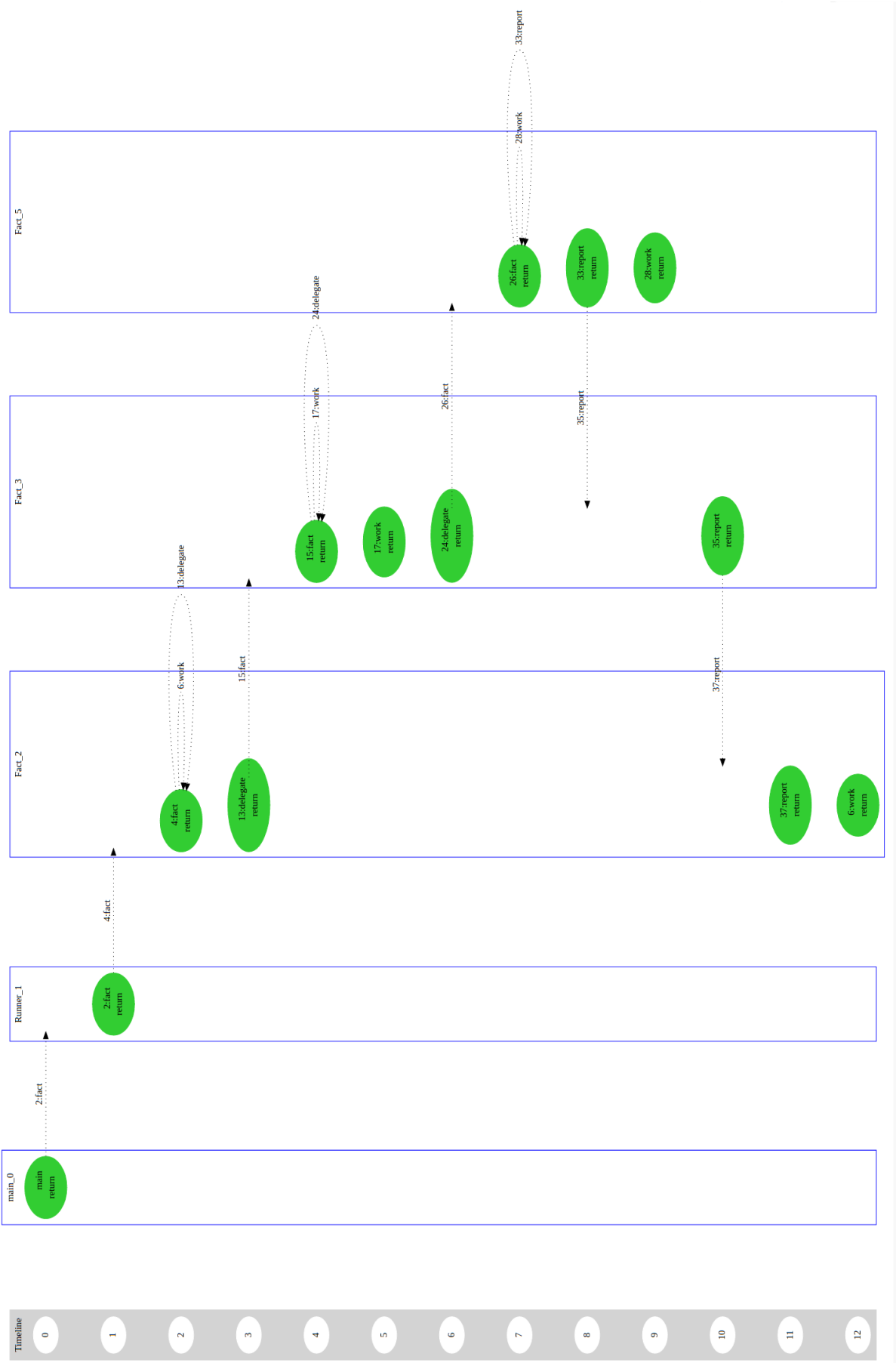


Figure 3.6: Diagram of correct execution for the running example



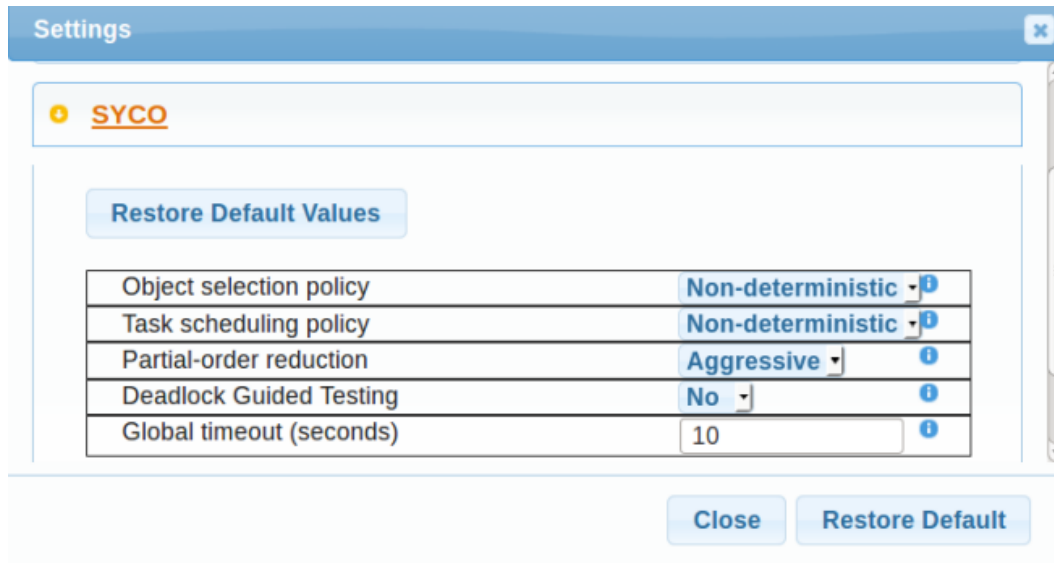


Figure 3.7: The SYCO parameters

### 3.1.1 Parameters of SYCO

Up to now we have executed SYCO with default parameters. Pressing button **Settings** at the toolbar, the SYCO parameters window shows up, which allows to configure the available parameters for each application. Figure 3.7 shows the parameters of SYCO, with default values. The following parameters can be set:

- *Object selection policy.* By default all objects from a state are selected non-deterministically on backtracking (option **Non-deterministic**). In case parameter *Partial-order reduction* below is enabled, only the required objects are selected according to the POR theory (see [3]). The other value **Round-robin** selects an object deterministically using a round-robin strategy.
- *Task scheduling policy.* It allows us to set the scheduling policy of objects. Available values are **FIFO**, **LIFO** and **Non-deterministic**. The default value is **Non-deterministic**. Otherwise, SYCO performs a deterministic simulation with the selected strategies.
- *Partial-order reduction.* It allows one to disable POR, by selecting value **None**, or to enable it with one of the following two levels of precision, values **Simple** and **Aggressive** (by default). Option **Simple** only applies the POR object selection in [3] whereas option **Aggressive** also applies the task selection function of [3]. In the example, 8 executions are obtained with aggressive POR, whereas 18 are obtained with the simple POR and 280 if POR is disabled. This illustrates the effectiveness of the available POR techniques.
- *Deadlock-guided testing.* It allows us to enable/disable deadlock-guided testing. By default it is disabled. If it is enabled, the testing process is guided towards deadlocks, discarding non-deadlock executions, with the corresponding state space reduction. This is useful in the context of deadlock detection and debugging. See Section 3.1.2 above.
- *Global timeout (seconds).* It allows us to set a global time limit.

### 3.1.2 Deadlock-guided Testing with SYCO

As already mentioned, SYCO includes the deadlock-guided testing approach of [2], in which the execution is driven towards potential deadlock paths discarding deadlock-free executions. If we enable **Deadlock Guided**

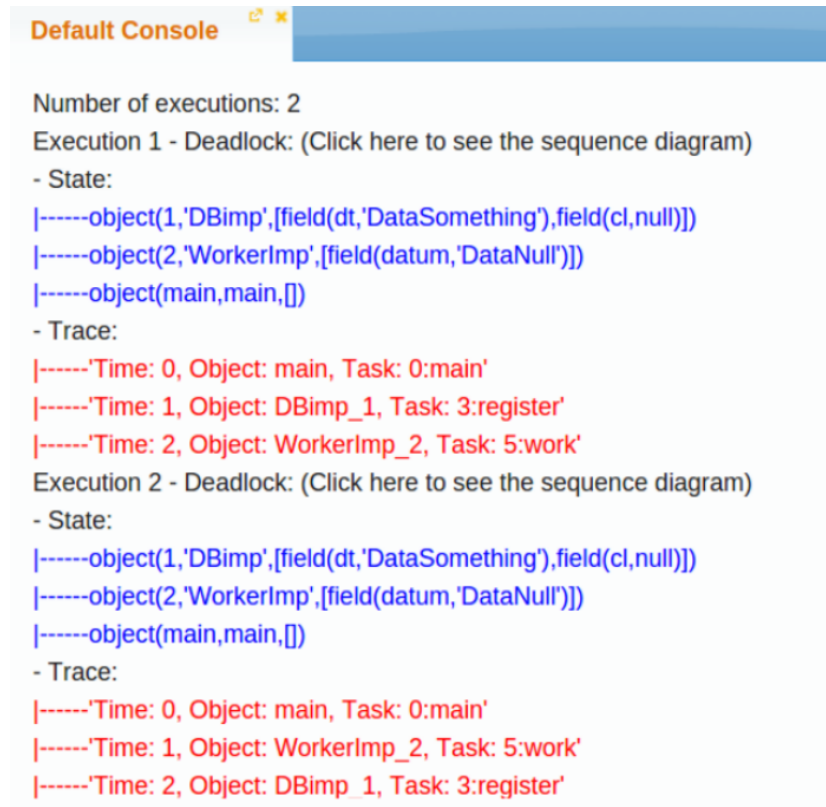
<pre> <b>data</b> Data = DataNull   DataSomething; <b>interface</b> DB {   <b>Unit</b> register(Worker w);   Data getData(Worker w); } <b>class</b> DBimp(Data dt) <b>implements</b> DB {   Worker cl = null;   <b>Unit</b> register(Worker w){     <b>Fut</b> &lt;Int&gt; f = w ! ping(1);     <b>Int</b> r = f.get;     <b>if</b> (r == 1) this.cl = w;   }   Data getData(Worker w){     Data d;     <b>if</b> (cl == w) d = this.dt;     <b>else</b> d = DataNull;     <b>return</b> d;   } } </pre>	<pre> <b>interface</b> Worker {   <b>Unit</b> work(DB db);   <b>Int</b> ping(<b>Int</b> n); } <b>class</b> WorkerImp() <b>implements</b> Worker {   Data datum = DataNull;   <b>Unit</b> work(DB db){     <b>Fut</b>&lt;Data&gt; f = db ! getData(this);     this.datum = f.get;   }   <b>Int</b> ping(<b>Int</b> n){     <b>Int</b> m = n;     <b>return</b> n;   } } { // main block   DB db = <b>new</b> DBimp(DataSomething);   Worker w = <b>new</b> WorkerImp();   db ! register(w);   w ! work(db); } </pre>
--	---

Figure 3.8: An example with deadlock

Testing for our running example, we get printed **Number of executions:** 0 as result in the console, which means that the program is deadlock-free.

Let us consider the program in Figure 3.8 which simulates a simple communication protocol between a database and a worker. The main block creates the two objects and invokes the methods **register** and **work** respectively. The **work** method of the worker simply accesses the database (invoking asynchronously method **getData**) and then blocks until it gets the result, which is assigned to its **datum** field. The **register** method of the database, first checks that the worker is online (invoking asynchronously method **ping**), then blocks until it gets the result, and finally it registers the worker by storing its reference in its **cl** field. Method **getData** of the database returns its **dt** field if the caller worker is registered, otherwise it returns **DataNull**.

Depending on the sequence of interleavings, the execution of this program can finish: (i) as expected, i.e., with **w.datum** having the same value as **db.dt**, (ii) with **w.datum = DataNull**, or, (iii) in a deadlock. Case (i) happens when the worker is registered in the database before **getData** is executed. Case (ii) happens when **getData** is executed before assigning the worker as the database client. A deadlock is produced if both **register** and **work** start executing before **getData** and **ping**. With POR disabled, SYCO produces 6 executions for the example in Figure 3.8, which cover all possible task interleavings that may occur. SYCO reports that two executions are deadlock executions corresponding to sequences **main** → **register** → **work** and **main** → **work** → **register**, which correspond to scenario (iii). Within the remaining four executions, two of them correspond to scenario (i) and the other two to scenario (ii). If we enable **Deadlock-guided testing**, we obtain just the two deadlock executions which are shown in Figure 3.9. Looking at the sequence diagram of the first execution (Figure 3.10 up), we can observe a deadlock situation, since both **DBimp\_1** and **WorkerImp\_2** are blocked and, as we can see, they are squared in red color. During time 1, **DBimp\_1** gets blocked waiting for **WorkerImp\_2** to execute task 4:ping. During the next time, object **WorkerImp\_2**, instead of executing task 4:ping, it executes task 5:work, getting blocked waiting for **DBimp\_1** to execute 6:getData. Therefore, none of the objects can make any progress. Both tasks are highlighted with red solid edges to indicate that these are the ones responsible for the deadlock. The second execution (see Figure 3.10



```

Default Console

Number of executions: 2
Execution 1 - Deadlock: (Click here to see the sequence diagram)
- State:
|-----object(1,'DBimp',[field(dt,'DataSomething'),field(cl,null)])
|-----object(2,'WorkerImp',[field(datum,'DataNull')])
|-----object(main,main,[])
- Trace:
|-----'Time: 0, Object: main, Task: 0:main'
|-----'Time: 1, Object: DBimp_1, Task: 3:register'
|-----'Time: 2, Object: WorkerImp_2, Task: 5:work'
Execution 2 - Deadlock: (Click here to see the sequence diagram)
- State:
|-----object(1,'DBimp',[field(dt,'DataSomething'),field(cl,null)])
|-----object(2,'WorkerImp',[field(datum,'DataNull')])
|-----object(main,main,[])
- Trace:
|-----'Time: 0, Object: main, Task: 0:main'
|-----'Time: 1, Object: WorkerImp_2, Task: 5:work'
|-----'Time: 2, Object: DBimp_1, Task: 3:register'

```

Figure 3.9: Deadlock-guided testing on the Database example

down) is similar but changing the execution order between tasks 3:register and 5:work.

### 3.2 Use of aPET

This section illustrates the usage of aPET using our running example. In this case we select **Test case generation (aPET)** from the pull down menu in the toolbar. In contrast to SYCO, since aPET performs symbolic execution, it can be applied over any method, possibly containing input arguments. Symbolic execution produces as a result the conditions over the input arguments and input state, or directly concrete values satisfying those conditions, to execute the different execution paths. Also, for each considered path, the expressions to compute the corresponding outputs, or concrete outputs satisfying them, are generated. Methods to which we want to apply aPET are selected in the outline view.

Let us select method **fact** of class **Runner**, and generate test cases for it with aPET using default parameters. For this, we just click over the **Apply** button and in the console area we can observe that 5 test cases have been generated. Let us focus on the first test case which is shown in Figure 3.11.

- In the **Input** section, **Args** stands for the value of the input arguments; in this case **ref(A)**, 4 and 1 are the initial values computed for the input parameters **this**, **n** and **h**. **State** shows the input state. It contains only one object (the caller object) of class **Runner** identified by **A**.
- The **Output** section contains the **Return** value, followed by the final state. The return type of method **fact** is **Unit**, and, the final state contains 5 different objects, where the object identified by 0 contains the value 24 in its field **r**, which is the expected result for this input (4).

aPET also generates the traces associated with each test case and the corresponding sequence diagrams to graphically visualize the traces. They are displayed by clicking on “**Click here to see the sequence**

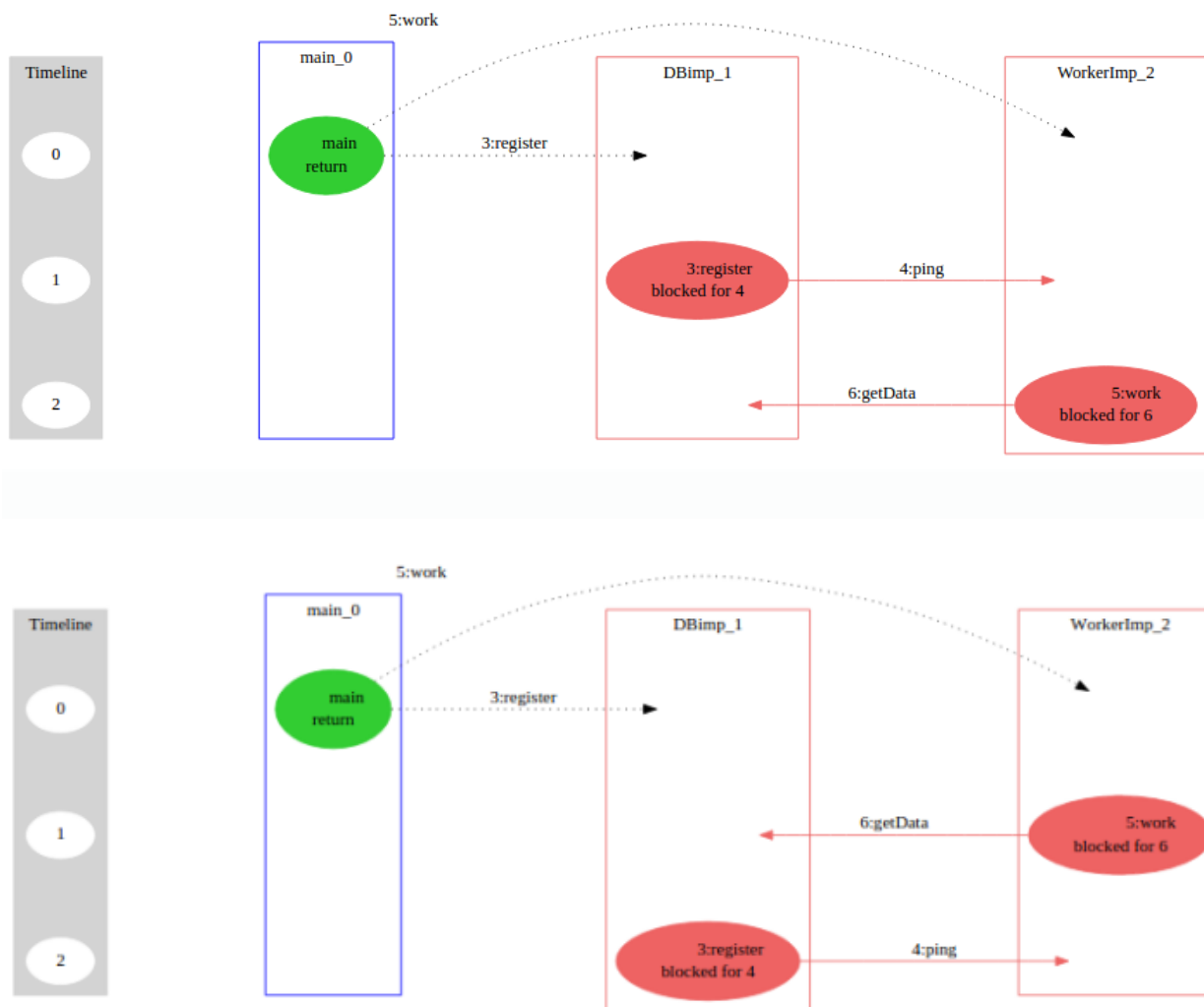


Figure 3.10: Sequence diagrams of deadlock executions

```

Test Case 1: (Click here to see the sequence diagram)
|-- Input:
|--- Args:
|-----[ref(A),4,1]
|--- State:
|-----object(A,'Runner',[])
|-- Output:
|--- Return:
|-----'Unit'
|--- State:
|-----object(1,'Fact',[field(boss,null),field(maxH,1),field(r,24)])
|-----object(6,'Fact',[field(boss,ref(1)),field(maxH,1),field(r,6)])
|-----object(7,'Fact',[field(boss,ref(6)),field(maxH,1),field(r,2)])
|-----object(8,'Fact',[field(boss,ref(7)),field(maxH,1),field(r,1)])
|-----object(A,'Runner',[])

```

Figure 3.11: TCG with aPET for method `fact`

`diagram''` in each test case. As in SYCO, if we click over one time point of the trace, the corresponding line in the source code is highlighted (in yellow color) in the code area.

### 3.2.1 Parameters of aPET

The parameters available for aPET are shown in Figure 3.12, with their corresponding default values. In the following we describe the meaning and available values for the different parameters:

*Concrete test-cases or path-constraints.* The result of each feasible execution path in the symbolic execution can be given in the form of (unresolved) path constraints (value **Path constraints**), or in the form of a concrete test case (value **Concrete tests**), where arbitrary concrete values satisfying the constraints are generated. Value **Hybrid** generates concrete data only for functional data, leaving path constraints involving numeric variables. As an example, let us consider the TCG with aPET of method `report` of class `Fact` with value **Path constraints**. The first computed test case is shown in the screenshot below:

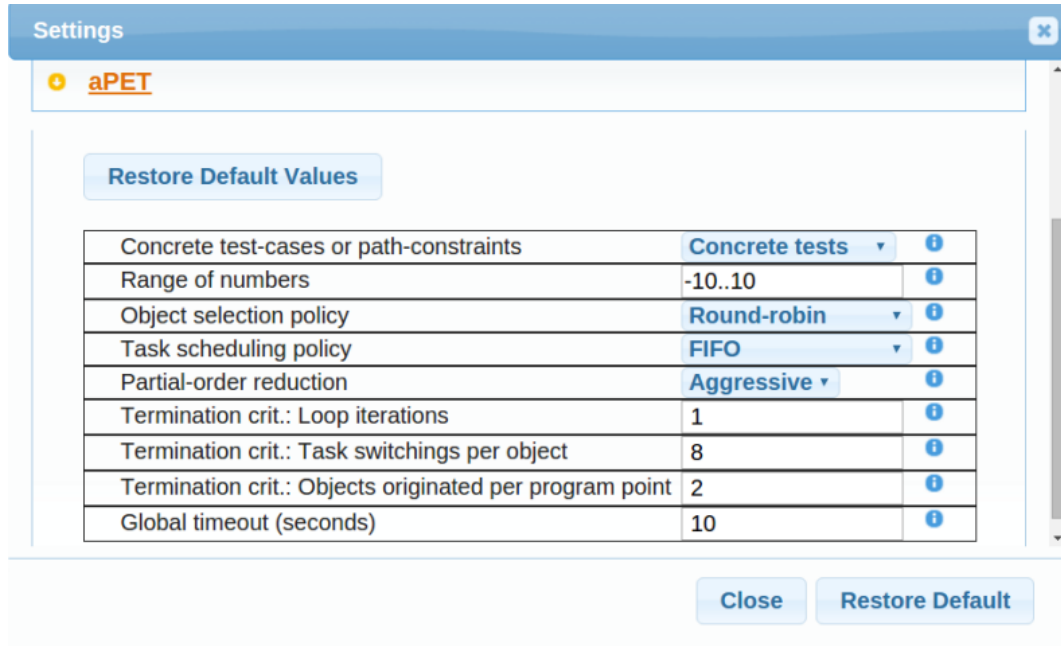
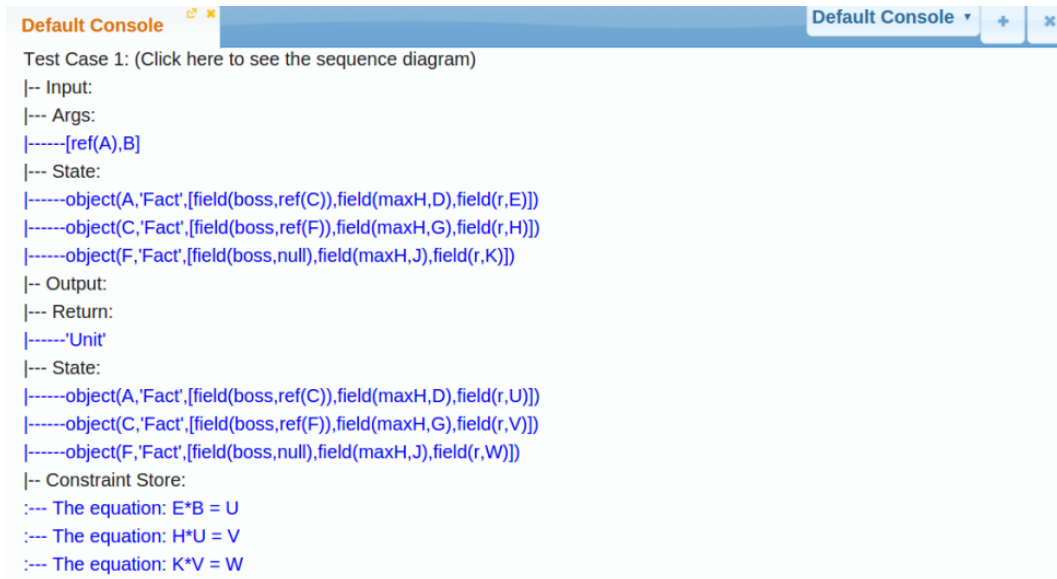


Figure 3.12: The aPET parameters



which can be read as: the initial and final states contain three objects A, C and F such that the **boss** of A is C, the **boss** of C is F and the **boss** of F is null. Field **maxH** of the objects in the initial state remain the same in the final state. If we look at field **r** in the final state, the following associated constraints are obtained:

$$\begin{aligned}
 r_f^A &= r_i^A * B \\
 r_f^C &= r_i^C * r_i^A * B \\
 r_f^F &= r_i^F * r_i^C * r_i^A * B
 \end{aligned}$$

where  $r_f^o$  (resp.  $r_i^o$ ) stands for the final value (initial value) of field **r** of object **o**,  $o \in \{A, C, F\}$ .

*Range of numbers for concrete test cases.* It allows specifying the domain for numeric variables and it is given in the format *Min..Max*. This option is only applicable when that concrete test-cases are

```

Number of Test Cases: 5
Test Case 1: (Click here to see the sequence diagram)
|-- Input:
|--- Args:
|-----[ref(A),4,1]
|--- State:
|-----object(A,'Runner',[])
|-- Output:
|--- Return:
|-----'Unit'
|--- State:
|-----object(1,'Fact',[field(boss,null),field(maxH,1),field(r,4)])
|-----object(2,'Fact',[field(boss,ref(1)),field(maxH,1),field(r,3)])
|-----object(3,'Fact',[field(boss,ref(2)),field(maxH,1),field(r,2)])
|-----object(4,'Fact',[field(boss,ref(3)),field(maxH,1),field(r,1)])
|-----object(A,'Runner',[])

```

Figure 3.13: First test case obtained with LIFO scheduling

generated.

*Termination crit.: Loop iterations.* The specified number (by default 1) is used as a limit on the maximum number of loop iterations or function recursive calls which are allowed in symbolic execution.

*Termination crit.: Task switchings per object.* The specified number (by default 8) is used as a limit on the maximum number of task switchings per object which are allowed in symbolic execution.

*Termination crit.: Objects originated per program point.* The specified number (by default 2) is used as a limit on the maximum number of objects originated per program point which are allowed in symbolic execution.

Parameters *Object selection policy*, *Task scheduling Policy*, *Partial-order reduction* and *Global timeout* have the same meaning as in SYCO (see Section 3.1.1). The first two have however different default values in aPET, namely **Round-robin**, and **FIFO** respectively. This is because, in the context of symbolic execution, it is much more likely to run into state explosion problems with non-deterministic schedulings.

Let us set the task scheduling to **LIFO** and run again aPET for method **fact**. We also get five test cases, but in this case, the first two test cases exploit the error reported in Section 3.1. E.g., in the first test case (see Figure 3.13), method **fact** is called with values 4 and 1 resp., and 4 is obtained as a result. If we have a look at the sequence diagram, it could be observed that the problem is similar to that shown in Figure 3.4.

Finally, if we set both scheduling parameters to **Non-deterministic**, we get 32 test cases, many of which exploit the same error.

## Chapter 4

# Conclusions

There has been good scientific progress in this task. We have defined new strategies for eliminating redundant computations when testing **ABS** models. The strategies have been successfully applied in both static and dynamic scenarios. We have contributed as well with the combination of deadlock analysis and testing in such a way that potential deadlock cycles found by deadlock analysis are used to guide the systematic testing process in order to find associated deadlock traces (discard deadlock-free paths). The technical details of this hybrid technique that combines analysis and testing will be reported on Deliverable D3.4 (hybrid analyses).

Besides the scientific progress, there has been also an important implementation effort: the techniques described in this deliverable have lead to the development of two testing prototypes, the **SYCO** and **aPET** tools, which are fully integrated into the **Envisage** collaboratory.



# Bibliography

- [1] P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. of POPL'14*, pages 373–384. ACM, 2014.
- [2] E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. In *12th International Conference on integrated Formal Methods, iFM 2016*, 2016. To appear.
- [3] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In Erika Ábrahám and Catuscia Palamidessi, editors, *34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems (FORTE 2014)*, volume 8461 of *Lecture Notes in Computer Science*, pages 49–65. Springer-Verlag, 2014.
- [4] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Test Case Generation of Actor Systems. In *13th International Symposium on Automated Technology for Verification and Analysis, ATVA 2015. Proceedings*, volume 9364 of *Lecture Notes in Computer Science*, pages 259–275. Springer-Verlag, 2015.
- [5] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Jose Miguel Rojas. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency. In *Formal Methods for Executable Software Models*, volume 8483 of *Lecture Notes in Computer Science*, pages 263–309. Springer, 2014.
- [6] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. May-Happen-in-Parallel Analysis for Actor-based Concurrency. *ACM Transactions on Computational Logic*, 2015.
- [7] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A systematic testing tool for concurrent objects. In *25th International Conference on Compiler Construction*. ACM, 2016. To appear.
- [8] M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163. IEEE, 2013.
- [9] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.
- [10] F.S. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen, and G. Zavattaro. A petri net based analysis of deadlocks for active objects and futures. In *Proc. of Formal Aspects of Component Software - 9th International Workshop, FACS 2012*, volume 7684 of *Lecture Notes in Computer Science*, pages 110–127. Springer-Verlag, 2012.
- [11] François Degraeve, Tom Schrijvers, and Wim Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *LOPSTR'09*, volume 6037 of *LNCS*, pages 128–142. Springer, 2010.
- [12] Christian Engel and Reiner Hähnle. Generating Unit Tests from Formal Proofs. In *Proc. of TAP'07*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.

- [13] Javier Esparza. Model checking using net unfoldings. *Sci. Comput. Program.*, 23(2-3):151–195, 1994.
- [14] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proc. of POPL’05*, pages 110–121. ACM, 2005.
- [15] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Proc. FORTE/FMOODS 2013*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer-Verlag, 2013.
- [16] S. Genaim and J. Doménech. The EasyInterface Framework, 2015. <http://github.com/abstools/easyinterface>.
- [17] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *CoRR*, abs/1511.04926, 2015.
- [18] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proc. of CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1991.
- [19] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP Framework for Computing Structural Test Data. In *Proc. of Computational Logic’00*, volume 1861 of *LNAI*, pages 399–413. Springer, 2000.
- [20] C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
- [21] Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *Proc. of IASTED Conf. on Software Engineering’04*, pages 365–371. IASTED/ACTA Press, 2004.
- [22] M. Naik, Chang-Seo Park, Koushik Sen, and D. Gay. Effective static deadlock detection. In *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009.*, pages 386–396, 2009.
- [23] 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 *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.
- [24] Peter Y. H. Wong, Richard Bubel, Frank S. de Boer, Miguel Gómez-Zamalloa, Stijn de Gouw, Reiner Hähnle, Karl Meinke, and Muddassar Azam Sindhu. Testing Abstract Behavioral Specifications. *Journal on Software Tools for Technology Transfer*, 17(1):107–119, 2015.

# Glossary

**ABS:** Abstract behavioral specification.

**AST:** Abstract syntax tree.

**aPET:** Symbolic execution-based test case generator for ABS programs.

**Coverage criteria:** It is a measure used to describe the degree to which the source code of a program is tested by a particular test suite.

**DECO:** Deadlock analyzer for concurrent objects.

**DPOR:** Dynamic partial order reduction

**Dynamic testing:** It is a term used in software engineering to describe the testing of the dynamic behavior of code. That is, dynamic testing refers to the examination of the behaviour from the program w.r.t. concrete values for the input variables.

**IR:** Intermediate representation.

**POR:** Partial order reduction techniques to reduce the exploration of the search space without losing solutions when testing concurrent programs.

**SYCO:** Systematic testing tool for Concurrent Objects.

**Software testing:** It is a process conducted to provide stakeholders with information about the quality of the software under test.

**Static testing:** It refers to the examination of the behaviour from the program without any information on the concrete values for the input variables.

**Test Case:** Set of conditions under which a tester will determine whether an application is working as it was originally established for it to do.

**Test case generation (TCG):** It is the process of creating a set of data for testing the adequacy of new or revised software applications.

## Appendix A

Book Chapter “*Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency*”, [5]

# Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-based Instance, and Actor-based Concurrency

Elvira Albert<sup>1</sup>, Puri Arenas<sup>1</sup>, Miguel Gómez-Zamalloa<sup>1</sup>, Jose Miguel Rojas<sup>2</sup>

<sup>1</sup> DSIC, Complutense University of Madrid (UCM), Spain

<sup>2</sup> Department of Computer Science, University of Sheffield, UK

**Abstract.** The focus of this tutorial is white-box test case generation (TCG) based on symbolic execution. Symbolic execution consists in executing a program with the contents of its input arguments being symbolic variables rather than concrete values. A symbolic execution tree characterizes the set of execution paths explored during the symbolic execution of a program. Test cases can be then obtained from the successful branches of the tree. The tutorial is split into three parts: (1) The first part overviews the basic techniques used in TCG to ensure termination, handling heap-manipulating programs, achieving compositionality in the process and guiding TCG towards interesting test cases. (2) In the second part, we focus on a particular implementation of the TCG framework in constraint logic programming (CLP). In essence, the imperative object-oriented program under test is automatically transformed into an equivalent executable CLP-translated program. The main advantage of CLP-based TCG is that the standard mechanism of CLP performs symbolic execution for free. The PET system is an open-source software that implements this approach. (3) Finally, in the last part, we study the extension of TCG to actor-based concurrent programs.

## 1 Introduction

A lot of research has been devoted in the last years to the problem of generating test cases automatically. A recent survey [6] describes some of the most prominent approaches to TCG, namely *model-based TCG*, *combinatorial TCG*, *(adaptive) random TCG*, *search-based TCG* and *structural (white-box) TCG*. This tutorial focuses on *structural (white-box) TCG*, an approach in which the availability of the code of the program under test is assumed and test cases are obtained from the concrete program (e.g., using its control flow graph) in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, since we assume no knowledge about the input data, in contrast to *dynamic* approaches [17, 24] which execute the program under test using concrete input values.

*Symbolic execution* [11, 13, 15, 23, 31, 35, 36, 46] is arguably the most widely used enabling technique for structural white-box TCG. It has received a renewed

interest in recent years, thanks in part to the increased availability of computational power and decision procedures [9]. Structural white-box TCG is among the most studied applications of symbolic execution, with several tools available [10]. Symbolic execution consists in executing a program with the contents of its input arguments being symbolic variables rather than concrete values. A symbolic execution *tree* characterizes the set of execution paths explored during the symbolic execution of a program. Test cases are obtained from the successful branches of the tree. The set of obtained test cases forms a test suite.

The first part of the tutorial is devoted to review the basic concepts of TCG by symbolic execution. We start by explaining the challenges to efficiently handle heap-manipulating programs [38] in symbolic execution. The presence of dynamic memory operations such as object creation and read/write field accesses requires special treatment during symbolic execution. Moreover, in order to ensure reliability, symbolic execution must consider all possible shapes these dynamic data structures can take. We proceed next to see how one can go to symbolic execution to the actual production of test cases. An important issue that is discussed afterwards is the compositionality of the TCG process. Finally, we overview a practical issue to efficiently generate more relevant test cases. In particular, guided TCG is a methodology that aims at steering symbolic execution towards specific program paths in order to generate relevant test cases and filter out less interesting ones.

The second part of the tutorial introduces CLP-based Test Case Generation. CLP-based TCG advocates the use of CLP technology to perform test case generation of imperative object-oriented programs. The process has two phases. In the first phase, the imperative object-oriented program under test is automatically transformed into an equivalent executable *CLP-translated* program. Instructions that manipulate heap-allocated data are represented by means of calls to specific *heap operations*. In the second phase, the CLP-translated program is symbolically executed using the standard CLP execution and constraint solving mechanisms. The above-mentioned heap operations are also implemented in standard CLP, in a suitable way in order to support symbolic execution. We will see the advantages of the CLP-based framework and, in particular, why it is very relevant to implement guided TCG and an efficient heap solver. In this context, we present the PET system, a system that implements the CLP-based TCG framework described in this part and which is available online.

The last part of the tutorial is focused on TCG of actor-based concurrent programs. It is known that 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. 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., [45] and its references). Furthermore, the fact that different scheduling policies can be implemented affects the order in which tasks are selected for execution and, thus,

the initial state when resuming a task can be different by adopting one policy or another. As a result, computation is often non-deterministic and multiple (possibly different) solutions can be produced depending on the interleaved tasks and the scheduler.

The adoption of actor systems has some advantages in the regard. Very briefly, actors [1, 25] constitute 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 or objects, each with its own local state and thread of control, that communicate by exchanging messages asynchronously. An object configuration consists of the local state of the objects and a set of pending messages (or *tasks*). In response to receiving a message, an object can update its local state, send messages, or create new objects. At each step in the computation of an object system, an object from the system is scheduled to process one of its pending messages. The advantage of using actor-systems in testing is that, as objects do not share their states, one can assume [41] 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). This assumption alleviates already a lot the scalability issues mentioned above. We will discuss a basic algorithm and the main challenges in TCG of actor systems.

## 2 Test Case Generation by Symbolic Execution

This section provides a general overview of TCG by symbolic execution and the main challenges that currently the method poses.

### 2.1 Basic Concepts in Symbolic Execution

A symbolic execution *tree* characterizes the set of execution paths explored during the symbolic execution of a program. During the course of symbolic execution, the values of the program’s variables are represented as symbolic expressions over the input symbolic values and a *path condition* is maintained. Such a path condition is updated whenever a branch instruction is executed. For instance, for each conditional statement in the program, symbolic execution explores both the “then” and the “else” branch, refining the path condition accordingly. The satisfiability of each of these branches is checked and symbolic execution stops exploring any path whose path condition becomes unsatisfiable, hence only feasible paths are followed. Test cases are obtained from the successful branches of the tree. The set of obtained test cases forms a test suite.

In this context, the quality of a test suite is usually assessed by using code coverage criteria. A coverage criterion aims at measuring how well the program under test is exercised by a test suite. Some popular coverage criteria are: *statement coverage* which requires that every statement of the code is executed; *branch coverage* which requires all conditional statements in the program to be

evaluated both to true and false; and *path coverage* which requires that every possible trace through a given part of the code is executed. These criteria are however not *finitely applicable* [49]. That is, they can not always be satisfied by a *finite* test suite, due to infinite paths and infeasible statements in the program under test (i.e., dead code). An alternative to path coverage, which is *finitely applicable* is the *loop- $k$*  coverage criterion, which requires traversing all paths in the program except those with more than  $k$  iterations on any loop.

Observe that by construction symbolic execution achieves the *path coverage* criterion above described. However, since the symbolic execution tree is in general infinite, a termination criterion must be imposed to ensure its finiteness. Such a termination criterion can be expressed in different forms. For instance, a computation time budget can be established, or an explicit bound on the depth of the symbolic execution tree can be imposed. We adopt a more code-oriented termination criterion. Concretely, we impose an upper bound  $k$  on the number of times each loop is iterated. By doing so, the finitely applicable (feasible) version of the *path coverage* criterion, i.e., the *loop- $k$*  coverage, is achieved.

---

```

1 int intExp(int a,int n) {
2   if (n < 0)
3     throw new ArithmeticException();
4   else {
5     int out = 1;
6     while (n > 0) {
7       out = out*a;
8       n--;
9     }
10    return out;
11  }
12 }

```

---

Fig. 1: Java source code

*Example 1.* Figure 1 shows the Java source code for method `intExp` which takes two integer input arguments `a` and `n` and computes  $a^n$  by successive multiplications. If the value of the input argument `n` is less than 0, an arithmetic exception is thrown. For simplicity, we assume that the method cannot receive values 0 for both of its arguments (undefined  $0^0$ ). Figure 2 shows the symbolic execution tree of method `intExp` for *loop-1* termination criterion (*loop- $k$*  with  $k=1$ ). That is to say, we require all paths that do not exercise the loop body (zero times) and those that exercise the loop body one time. Nodes in the tree denote symbolic states, and the edges are labeled with the line number of the instruction that is executed. Observe that symbolic execution starts with the empty path condition ( $PC: true$ ). At each branching point,  $PC$  is updated with different condi-



tions over the input arguments. For instance, when the `if` statement is executed, both `then` (*true*) and `else` (*false*) alternatives are feasible, therefore symbolic execution forks and the PC is updated accordingly in each of the resulting paths.

In the tree, solid squares denote intermediate symbolic states, solid double squares denote successful (terminating) symbolic execution paths, and the only dashed square denotes an unfinished path, i.e., a path that is about to enter the loop body a second time and hence is pruned by the *loop-1* criterion.  $\square$

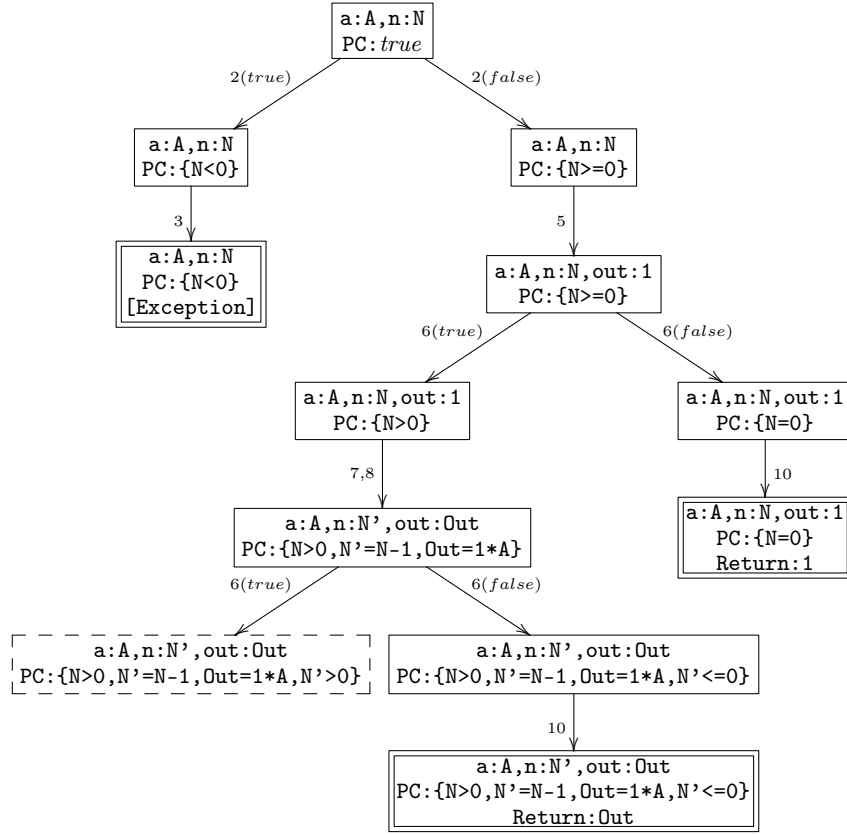


Fig. 2: Symbolic execution tree

## 2.2 Handling Heap-manipulating programs

One of the main challenges in symbolic execution is to efficiently handle heap-manipulating programs [38]. As will be illustrated later through an intuitive

example, these kind of programs often create and use complex dynamically heap-allocated data structures. The presence of dynamic memory operations such as object creation and read/write field accesses requires special treatment during symbolic execution. Moreover, in order to ensure reliability, symbolic execution must consider all possible shapes these dynamic data structures can take. In trying to do so, however, scalability issues arise since high (often exponential) numbers of shapes may be built due to the *aliasing* of references.

In practice, symbolic execution assumes no knowledge about the heap shape (unless explicitly provided in advance via e.g., preconditions), in contrast to standard execution, where a program runs on concrete and fully-known initial heap (as part of the execution context). Let us motivate the importance of special treatment for heap operations and aliasing of references on a simple example.

*Example 2.* Consider the following method `mist`. It receives as input arguments two references `r1` and `r2` to objects of type `C` (contains a field `f` of integer type), checks the value of `r1.f` and writes `r2.f` in the `then` branch or writes `r1.f` in the `else` branch.

```

1 void mist(C r1, C r2) {
2   if (r1.f > 0)
3     r2.f = 1;
4   else
5     r1.f = 0;
6 }
```

Seemingly, the method contains only two feasible paths, each corresponding to one branch of the `if` statement:

1. If `r1.f > 0`, then write `r2.f = 1` (line 3).
2. If `r1.f ≤ 0`, then write `r1.f = 0` (line 5). Nothing is learned about `r2`.

However, these cases fall short to cover all possible executions of method `mist`. There are other unapparent execution paths that must also be explored. Namely:

3. If `r1` points to *null*, then a null pointer exception is thrown at line 2.
4. If `r1.f > 0` and `r2` points to *null*, then a null pointer exception is thrown at line 3.
5. If `r1` and `r2` point to the same object `o` and `o.f > 0`, then write `o.f = 1` (line 3). We say that `r1` and `r2` are *aliased*.

Notice that only by exhaustive exploration of all possible heap configuration can symbolic execution generate these “hidden” paths and hence reveal the presence of potential runtime errors for this rather simple method. Furthermore, let this example also serve to see the relevance of the *loop-k* coverage criterion. Observe that the set of the first two cases above, while not being sufficient to exercise the complete behavior of method `mist`, would still be enough to achieve 100% branch and statement coverage, which may convey an illusory sense of confidence on the correctness of a *possibly* buggy program.  $\square$

*Lazy Initialization.* Lazy initialization [30] is the *de facto* standard technique to enable symbolic execution to systematically handle arbitrary input data structures, and to explore all possible heap shapes that can be generated during the process, including those produced due to aliasing of references. The main idea is that symbolic execution starts with no knowledge about the program’s input arguments and, as the program symbolically executes and accesses object fields, the components of the program’s inputs are initialized on an “as-needed” basis. The intuition is as follows. To symbolically execute method `m` of class `C`, a new object `o` of class `C` with all its fields uninitialized is created (the `this` object in Java). When an unknown field of primitive type is read, a fresh unconstrained variable is created for that field. When an unknown reference field is accessed, all possibilities are explored non-deterministically choosing among the following values: (a) `null`; (b) any existing symbolic object whose type is compatible with the field’s type and might *alias* with it; and (c) a fresh symbolic object. Such non-deterministic choices are materialized into branches in the symbolic execution tree. As a result, the heap associated with any particular execution path is built using only the constraints induced by the visited code.

The practicality and effectiveness of lazy initialization has been proved with its use by existing symbolic execution engines such as PET and SPF. However, the very nature of the technique, i.e., producing branching due to *aliasing* choices at *every* heap operation point, hampers the overall efficiency of symbolic execution and its applicability to real-world programs.

*A Heap Solver.* The observation that branching due to *aliasing* choices can be made “more lazily” than in lazy initialization by delaying such choices as much as possible lead to the development of a *heap solver* [4] which enables a more efficient symbolic execution of heap-manipulating programs. The key features of the heap solver are the treatment of reference aliasing by means of *disjunctive reasoning*, and the use of advanced *back-propagation* of heap related constraints. In addition, the heap solver supports the use of *heap assumptions* to avoid aliasing of data that, though legal, should not be provided as input.

Let us further illustrate the benefits of the heap solver over lazy initialization by symbolically executing method `m` from Figure 3 using both approaches. For simplicity, let us assume that the executions of methods `a` and `b` do not modify the heap. The symbolic execution tree computed using lazy initialization (as in, e.g., PET and SPF) is shown in Figure 4a. Note that before a field is accessed, the execution branches if it can alias with previously accessed fields. For example, the second field access `z.f` branches in order to consider the possible aliasing with the previously accessed `x.f`. Similarly, the write access to `y.f` must consider all possible aliasing choices with the two previous accessed fields `x.f` and `z.f`. This ensures that the effect of the field access is known within each branch. For example, in the leftmost branch the statement `y.f=x.f+1` assigns -4 to `x.f`, `y.f` and `z.f`, since in that branch all these objects are aliased. The advantage of this approach is that by the time we reach the `if` statement we know the result of the test, since each variable is fixed. However, such early branching creates

---

```

1 void m(Ref x, Ref y, Ref z) {
2   x.f=1;
3   z.f=-5;
4   a();
5   y.f=x.f+1;
6   b();
7   if (x==z)
8     c(y.f);
9   else
10    d(y.f);
11 }

```

---

Fig. 3: Heap Solver: Motivating example

a combinatorial explosion problem since, for example, method **a** is symbolically executed in two branches and method **b** in five.

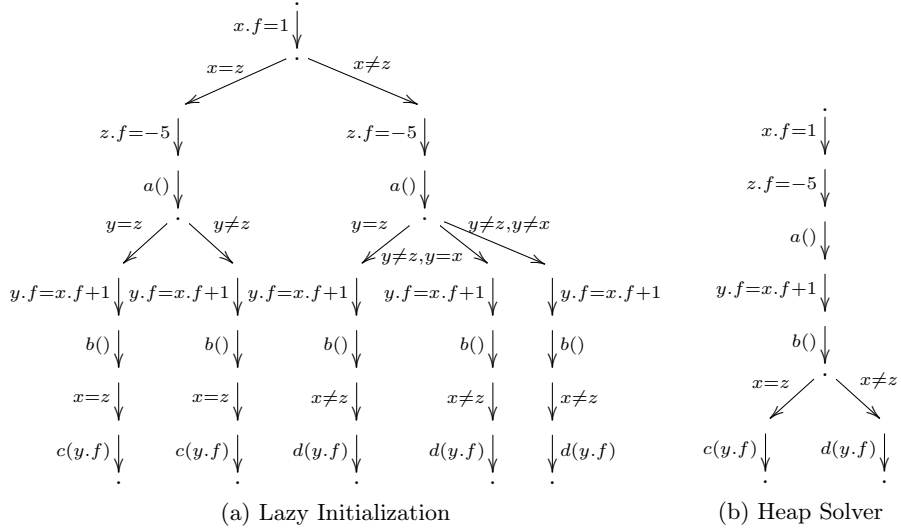


Fig. 4: Symbolic Execution Trees: Lazy Initialization and Heap Solver

On the other hand, the heap solver enables symbolic execution to perform as shown in Figure 4b, where branching only occurs due to explicit branching in the program, rather than to aliasing. For this purpose, the heap solver handles non-determinism due to aliasing of references by means of *disjunctions*. In particular, at instruction 5 the solver will carry the following conditional information for

$x.f'$  (the current value of field  $f$  of  $x$ ):  $x = z \rightarrow x.f' = z.f \wedge x \neq z \rightarrow x.f' = x.f$  indicating that if  $x$  and  $z$  are aliased, then  $x.f'$  will take its value from  $z.f$  and, otherwise, from  $x.f$ . Once the conditional statement at line 7 is executed and we learn that  $x$  and  $z$  are aliased (in the **then** branch), we need to look up *backwards* in the heap and propagate this unification so that instruction 5 can be fully executed. This allows the symbolic execution of  $d(y.f)$  with a known value for  $y.f$ . The heap solver works on a novel internal representation of the heap that encodes the disjunctive information and easily allows looking up backwards in the heap. In addition, it is possible to provide *heap assumptions* on non-aliasing, non-sharing and acyclicity of heap-allocated data in the initial state. The heap solver can take these assumptions into account to discard aliasing that is known not to occur for some input data. Importantly, the heap solver can be used by any symbolic execution tool for imperative languages through its interface heap operations.

**Backwards Propagation, Arrays, and Heap Assumptions.** As described in the previous section, the heap solver uses information about equality and disequality of references to determine equality among the heap cells. This is done by propagating such information forwards in the rules of attributes. A straightforward extension to the solver allows propagating information backwards as well. In doing so, the heap solver is capable of further refining disjunctive information and variables' domains, which in turn can lead to promptly pruning unfeasible symbolic execution branches.

*Example 3.* Consider the method `m` but with the condition of the `if` (in instruction 7) changed to "`if (x.f == 1)`". Thanks to backwards propagation, the solver can infer that in the `if` branch, variables `x`, `y` and `z` do not alias, and therefore the call `call_c` is performed with a 2 value.  $\square$

Another straightforward extension to the heap solver allows to handle arrays in a similar fashion to how object fields are handled, with the difference being that array indices play the role of object references that point to the heap-allocated data.

The last important feature of the heap solver is the support for heap assumptions. As we have seen so far, symbolic execution assumes feasible all possible kinds of aliasing among heap-allocated (reference) input data of the same type. However, it may be the case that while some of these aliasings might indeed occur, others might not (consider, for instance, aliased data structures that cannot be constructed using the public methods in the Java class). In order to avoid generating such inputs, the heap solver provides support for *heap assumptions*, that is, assertions describing reachability, aliasing, separation and sharing conditions in the heap. Concretely, the following heap assumptions are supported:

- *non-aliasing*( $a, b$ ): specifies that memory locations  $a$  and  $b$  are not the same.
- *non-sharing*( $a, b$ ): specifies disjointness, i.e., that references  $a$  and  $b$  do not share any common region in the heap.
- *acyclic*( $a$ ): specifies that  $a$  is an acyclic data structure.

### 2.3 From Symbolic Execution to TCG

The outcome of symbolic execution is a set of *path conditions*, one for each symbolic execution path. Each *path condition* represents the conditions over the input variables that characterize the set of feasible concrete executions of the program that take the same path. In a next step, off-the-shelf constraint solvers can be used to solve such path conditions and generate concrete instantiations for each of them. This last step provides actual test inputs for the program, amenable to further validation by testing frameworks such as JUnit, which execute such test inputs and check that the output is as expected.

*Example 4.* Let us look at the symbolic execution tree of Figure 2 again. Intuitively, the union of the three successful paths denoted with solid double squares make up the symbolic test suite for method `intExp` that optimally satisfies the *loop-1* criterion:

#	Input	Output	Path condition
1	A, N	[exception]	{N<0}
2	A, N	1	{N=0}
3	A, N	Out	{N>0, N'=N-1, Out=1*A, N'<=0}

The following are *concrete* test cases that can be derived from the above symbolic ones.

#	Input	Output
1	-10, -10	[Exception]
2	-10, 0	1
3	-10, 1	10

And from these concrete test cases, the JUnit tests shown in Figure 5 can be obtained.

It is important to note that imposing a larger  $k$  would allow to continue the exploration through the unfinished, pruned path (dashed square) thus generating test cases corresponding to further loop unrollings.  $\square$

### 2.4 Compositionality

Compositional reasoning is a general purpose methodology that has been successfully applied in the past to scale up static analysis and software verification techniques and that has also proved effective for scaling up symbolic execution and TCG [5, 7, 19, 40]. The overall goal of compositionality is to alleviate the inter-procedural path explosion problem. That is, in the context of symbolic execution and TCG, the path explosion caused by repeatedly conjoining the symbolic execution trees of methods when their invocations occur. The main idea is that symbolic execution and TCG of large programs can be done more effectively, and more efficiently, by first performing symbolic execution and TCG

---

```

public void test_1(){
    int input0 = -10;
    int input1 = -10;
    try{
        int output = Test.intExp(input0,input1);
    }
    catch(Exception ex){
        assertEquals("exception","java.lang.ArithmeticException",
            ex.getClass().getName());
        return;
    }
    fail("Fail");
}

public void test_2(){
    int input0 = -10;
    int input1 = 0;
    int output = Test.intExp(input0,input1);
    int expected = 1;
    assertEquals("OK",expected,output);
}

public void test_3(){
    int input0 = -10;
    int input1 = 1;
    int output = Test.intExp(input0,input1);
    int expected = -10;
    assertEquals("OK",expected,output);
}

```

---

Fig. 5: JUnit tests generated for introductory example

of their individual components separately. In the context of object-oriented programming, a method is the basic code component.

In symbolic execution for TCG, compositionality means that when a method  $m$  invokes another method  $p$ , for which TCG has already been performed, the execution can *compose* the *test cases* available for  $p$  (also known as *method summary* for  $p$ ) with the current execution state and continue the process, instead of having to symbolically execute  $p$  again. By test cases (or method summary), we refer to the set of path conditions obtained by symbolically executing  $p$ . As a result of this composition step, a method summary for  $m$  is created. Then, larger portions of the system under test (components, modules, libraries, etc.) are incrementally executed, following a bottom-up traversal of its call graph, composing previously computed components results (*summaries*) until finally whole-program results can be computed. Let us recall that since the symbolic execution tree is in general infinite, a *termination criterion* is essential to ensure finiteness of the process. Then, a method summary is a *finite* set of *summary cases*, one for each terminating path through the symbolic execution tree of the method. Intuitively, a summary can be regarded as a complete specification of

the method for a certain termination criterion, but it is still a partial specification of the method in general.

Intuitively, compositional TCG has several advantages over traditional non-compositional TCG. First, it avoids repeatedly performing TCG of the same method. Second, components can be tested with higher precision when they are chosen small enough. Third, since separate TCG is done on parts and not on the whole program, total memory consumption may be reduced. Fourth, separate TCG can be performed in parallel on independent computers and the global TCG time can be reduced as well. Furthermore, having a compositional TCG approach in turn provides a practical solution to handle *native code*, i.e., code which is implemented in a different programming language and may be unavailable. This is achieved by modeling the behavior of native code as a method summary which can be composed with the current state during symbolic execution in the same way as the test cases inferred automatically by the testing tool are. By treating native code, we overcome one of the inherent limitations of symbolic execution (see [38]).

**Approaches to Compositional TCG.** In order to perform compositional TCG, two main approaches can be considered:

*Context-sensitive.* Starting from an entry method  $m$  (and possibly a set of pre-conditions), TCG performs a top-down symbolic execution such that, when a method call  $p$  is found, its code is executed from the actual state  $\phi$ . In a context-sensitive approach, once a method is executed, we store the summary computed for  $p$  in the context  $\phi$ . If we later reach another call to  $p$  within a (possibly different) context  $\phi'$ , we first check if the stored context is sufficiently general. In such case, we can adapt the existing summary for  $p$  to the current context  $\phi'$ . At the end of each execution, it can be decided which of the computed (context-sensitive) summaries are stored for future use.

*Context-insensitive.* Another possibility is to perform the TCG process in a context-insensitive way. This strategy comprises the following steps. First, the call graph for the entry method  $m_P$  of the program under test is computed, which gives us the set of methods that must be tested. Then, the strongly connected components (SCCs for short) for such graph are computed. SCCs are traversed in reverse topological order starting from those which do not depend on any other. The idea is that each SCC is symbolically executed from its entry  $m_{scc}$  w.r.t. the most general context (i.e., *true*). If there are several entries to the same SCC, the process is repeated for each of them. Hence, it is guaranteed that the obtained summaries can always be adapted to more specific contexts.

In general terms, the advantages of the context-insensitive approach are that composition can always be performed and that only one summary needs to be stored per method. However, since no context information is assumed, summaries can contain more test cases than necessary and can be thus more expensive to obtain. In contrast, the context-sensitive approach ensures that only the required



information is computed, but it can happen that there are several invocations to the same method that cannot reuse previous summaries (because the associated contexts are not sufficiently general). In such case, it is more efficient to obtain the summary without assuming any context. A context-insensitive approach is used in what follows.

**Method Summaries.** A *method summary* for  $m$  is a finite set of *summary cases*, each of which mainly consists of the path condition for a particular symbolic execution path of  $m$ . Each element in a summary is said to be a *summary case* of the summary. Intuitively, a method summary can be seen as a *complete specification* of the method for the considered coverage criterion, so that each summary case corresponds to the *path constraints* associated to each finished path in the corresponding (finite) execution tree. Note that, though the specification is complete for the criterion considered, it will be, in general, a *partial specification* for the method, since the finite tree may contain incomplete branches which, if further expanded, may result in (infinitely) many execution paths.

When the method does not include any heap-related operation, the path condition alone sufficiently characterizes the symbolic execution path (as in [7, 19]). However, in the presence of heap-manipulating methods, special mechanisms must be employed. We adopt an intuitive alternative which consists in explicitly encoding the input and output heaps and store them along with the path condition. Doing so, requires the implementation of two operations, a heap compatibility check and a heap composition operation.

**Compatibility and Composition of Summaries.** Let us assume that during the symbolic execution of a method  $m$ , there is a method invocation to another method  $p$  within a current state  $\phi$ . The challenge is to define a composition operation so that, instead of symbolically executing  $p$ , its previously computed summary  $\mathcal{S}_p$  can be reused. As a result, TCG for  $m$  should produce the same results regardless of whether we use a summary for  $p$  or we inline symbolical execution of  $p$  within TCG for  $m$ , in a non-compositional way. Roughly speaking, the state  $\phi_c$  stored in a summary case is *compatible* with the current state  $\phi$  if: 1) the path condition stored in the summary case can be conjoined to the current path condition, and 2) the structure of the input heap in the summary case match with the structure of the current heap. Note that compatibility of a summary case is checked on the fly, so that if  $\phi$  is not compatible with  $\phi_c$ , the composition will fail, the summary case will be discarded, and symbolic execution will proceed to attempt to compose the next summary case in  $\mathcal{S}_p$ .

*Example 5.* Table 1 shows the summary obtained by symbolically executing method `simplify` using the *loop-1* coverage criterion: The summary contains 5 cases, which correspond to the different execution paths induced by the calls to methods `gcd` and `abs`. For the sake of clarity, we adopt a graphical representation for the input and output heaps. Heap locations are shown as arrows labeled

---

```

class Arithmetics {
    static int abs(int a) {
        if (a >= 0) return a;
        else return -a;
    }
    static int gcd(int a,int b) {
        int res;
        while (b != 0) {
            res = a%b; a = b; b = res;
        }
        return abs(a);
    }
}
class Rational {
    int n; int d;
    void simplify() {
        int gcd = Arithmetics.gcd(n,d);
        n = n/gcd; d = d/gcd;
    }
    Rational[] simp(Rational[] rs) {
        int length = rs.length;
        Rational[] oldRs = new Rational[length];
        arraycopy(rs,oldRs,length);
        for (int i = 0; i < length; i++)
            rs[i].simplify();
        return oldRs;
    }
}

```

---

Fig. 6: Example for Compositional TCG.

Table 1: Summary of method `simplify`


$A_{in}$	$A_{out}$	$Heap_{in}$	$Heap_{out}$	$EF$	$Constraints$
$r(A)$	$A \rightarrow \frac{F}{0}$	$A \rightarrow \frac{M}{0}$		ok	$F < 0, N = -F, M = F/N$
$r(A)$	$A \rightarrow \frac{F}{0}$	$A \rightarrow \frac{1}{0}$		ok	$F > 0$
$r(A)$	$A \rightarrow \frac{0}{0}$	$A \rightarrow \frac{0}{0}$	$B \rightarrow \frac{0}{0}$ 	exc(B)	
$r(A)$	$A \rightarrow \frac{F}{G}$	$A \rightarrow \frac{M}{N}$		ok	$G < 0, F \bmod G = 0, K = -G, M = F/K, N = G/K$
$r(A)$	$A \rightarrow \frac{F}{G}$	$A \rightarrow \frac{M}{1}$		ok	$G > 0, F \bmod G = 0, M = F/G$

Table 2: Summary of method `arraycopy`

$A_{in}$	$A_{out}$	$Heap_{in}$	$Heap_{out}$	$EF$	$Constraints$
$[X, Y, 0]$	H	H	H	ok	$\emptyset$
$[r(A), \text{null}, Z]$	$A \rightarrow \boxed{L[V]_{-}}$	$A \rightarrow \boxed{L[V]_{-}} \rightarrow \text{NPE}$	$B \rightarrow \text{NPE}$	$\text{exc}(B)$	$Z > 0, L > 0$
$[\text{null}, Y, Z]$	H	$A \rightarrow \text{NPE}$	$A \rightarrow \text{NPE}$	$\text{exc}(A)$	$Z > 0$
$[X, Y, Z]$	H	$A \rightarrow \text{AE}$	$A \rightarrow \text{AE}$	$\text{exc}(A)$	$Z < 0$
$[r(A), r(B), 1]$	$A \rightarrow \boxed{L1[V]_{-}} \rightarrow \boxed{L2[V2]_{-}}$	$A \rightarrow \boxed{L1[V]_{-}} \rightarrow \boxed{L2[V]_{-}}$	$B \rightarrow \boxed{L2[V]_{-}}$	ok	$L1 > 1, L2 > 0$

with their reference variable names. Split-circles represent objects of type `R` and fields `n` and `d` are shown in the upper and lower part, respectively. Exceptions are shown as starbursts, like in the special case of the fraction “0/0”, for which an arithmetic exception (**AE**) is thrown due to a division by zero. In the method summary examples of Tables 2 and 3, split-rectangles represent arrays, with the length of the array in the upper part and its list of values in the lower one. Assume that method `arraycopy` is native. This means that its code is not available and we cannot symbolically execute it. A method summary for `arraycopy` can be provided, as shown in Table 2, where we have (manually) specified five cases: the first one for arrays of length zero, the second and third ones for null array references, the fourth one for a negative length, and finally a normal execution on non-null arrays. Now, by using our compositional reasoning, we can continue symbolic execution for `simp` by composing the specified summary of `arraycopy` and the one computed for `simplify`. The result of compositional symbolic execution is presented in Table 3, that is, the entire summary of method `simp` for a *loop-1* coverage criterion.  $\square$

## 2.5 Guided TCG

A common limitation of symbolic execution in the context of TCG is that it tends to produce an unnecessarily large number of test cases for all but tiny programs. This limitation not only hinders scalability but also complicates human reasoning on the generated test cases. Guided TCG is a methodology that aims at steering symbolic execution towards specific program paths in order to efficiently generate more relevant test cases and filter out less interesting ones with respect to a given structural *selection criterion*. The goal is thus to improve on scalability and efficiency by achieving a high degree of control over the coverage criterion and hence avoiding the exploration of unfeasible paths. This has potential applicability for industrial software testing practices such as unit testing, where units of code (e.g. methods) must be thoroughly tested in isolation, or *selective testing*, in which only specific paths of a program must be tested.

Table 3: Summary of method `simp`

$A_{in}$	$A_{out}$	$Heap_{in}$	$Heap_{out}$	$EF$	$Constraints$
$r(A)$	$r(B)$	$A \rightarrow [0]$	$A \rightarrow [0] \ B \rightarrow [0]$	ok	$\emptyset$
null	X	H	$A \rightarrow \text{NPE}$	exc(A)	$\emptyset$
$r(A)$	$r(C)$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{F}{0}$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{M}{0} \ C \rightarrow [1[r(B)]]$	ok	$F < 0, K = -F, M = F/K$
$r(A)$	$r(C)$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{F}{0}$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{1}{0} \ C \rightarrow [1[r(B)]]$	ok	$F > 0$
$r(A)$	X	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{0}{0}$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{0}{0} \ C \rightarrow [1[r(B)]] \ D \rightarrow \text{AE}$	exc(D)	$\emptyset$
$r(A)$	$r(C)$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{F}{G}$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{M}{N} \ C \rightarrow [1[r(B)]]$	ok	$G < 0, F \bmod G = 0, K = -G, M = F/K, N = G/K$
$r(A)$	$r(C)$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{F}{G}$	$A \rightarrow [1[r(B)]] \ B \rightarrow \frac{M}{1} \ C \rightarrow [1[r(B)]]$	ok	$G > 0, F \bmod G = 0, M = F/G$
$r(A)$	X	$A \rightarrow [1[\text{null}]]$	$A \rightarrow [1[\text{null}]] \ C \rightarrow [1[\text{null}]] \ B \rightarrow \text{NPE}$	exc(B)	$\emptyset$

*Example 6.* Let us consider the unit-testing for method `simplify` (see Figure 6). A proper set of unit-tests should include one test to exercise the exceptional behavior arising from the division by zero, and another test to exercise the normal behavior. Ideally, no more tests should be provided since there is anything else to be tested in method `simplify`. This methodology works well under the assumption that called methods are tested on their own, in this case method `gcd`. Standard TCG by symbolic execution would consider all possible paths including those arising from the different executions of method `gcd`, in this case 5 paths. The challenge in Guided TCG is to generate only the two test-cases above, avoiding as much as possible traversing the rest of the paths (which for this criterion can be considered redundant). As another example, let us consider selective testing for method `simplify`. E.g., one could be interested in generating a test-case (if any) that makes method `simplify` produce an exception due to a division by zero. The challenge in Guided TCG is again to generate such a test avoiding traversing as much as possible the rest of the paths.  $\square$

The intuition of Guided TCG is as follows: (1) A heuristics-based *trace-generator* generates possibly partial traces, i.e., partial descriptions of paths, according to a given selection criterion. This can be done by relying on the control-flow graph of the program. (2) Bounded symbolic execution is guided by the obtained traces. The process is repeated until the selection criterion is satisfied or until no more traces are generated. Section 3.6 presents a concrete CLP-based methodology for guided TCG and formalizes a concrete guided TCG scheme to support the criteria for unit testing considered in the above example.

### 3 CLP-based TCG

We present a particular instance of TCG based on symbolic execution, and an implementation, in which CLP is used as enabling technology.

#### 3.1 Constraint Logic Programming

We assume certain familiarity with Logic Programming (LP) [33] and Constraint Logic Programming (CLP) [27, 34]. Hence we only briefly overview both paradigms.

**Logic Programming.** Logic Programming is a programming paradigm based on the use of formal logic as a programming language. A logic program is a finite set of predicates defining relationships between logical terms. An atom (or call)  $A$  is a syntactic construct of the form  $p(t_1, \dots, t_n)$ , with  $n \geq 0$ , where  $p/n$  is a predicate signature and  $t_1, \dots, t_n$  are terms. A clause is of the form  $H : -B_1, \dots, B_m$ , with  $m \geq 0$ , where its head  $H$  is an atom and its body  $B_1, \dots, B_m$  is a conjunction of  $m$  atoms (commas denote conjunctions). When  $m = 0$  the clause is called a fact and is written “ $H$ .”. The standard syntactic convention is that names of predicates and atoms begin with a lowercase letter. A goal is a conjunction of atoms. We denote by  $\{X_1 \rightarrow t_1, \dots, X_n \rightarrow t_n\}$  the substitution  $\sigma$  with  $\sigma(X_i) = t_i$  for  $i = 1, \dots, n$  (with  $X_i \neq X_j$  if  $i \neq j$ ), and  $\sigma(X) = X$  for all other variables  $X$ . Given an atom  $A$ ,  $\theta(A)$  denotes the application of substitution  $\theta$  to  $A$ . Given two substitutions  $\theta_1$  and  $\theta_2$ , we denote by  $\theta_1\theta_2$  their composition. An atom  $A'$  is an instance of  $A$  if there is a substitution  $\sigma$  with  $A' = \sigma(A)$ .

SLD (Selective Linear Definite clause)-resolution is the standard operational semantics of logic programs. It is based on the notion of derivations. A derivation step is defined as follows. Let  $G$  be  $A_1, \dots, A_R, \dots, A_k$  and  $C = H : -B_1, \dots, B_m$  be a renamed apart clause in  $P$  (i.e., it has no common variables with  $G$ ). Let  $A_R$  be the selected atom for its evaluation. As in Prolog, we assume the simple leftmost selection rule. Then,  $G'$  is derived from  $G$  if  $\theta$  is a *most general unifier* between  $A_R$  and  $H$ , and  $G'$  is the goal  $\theta(A_1, \dots, A_{R-1}, B_1, \dots, B_m, A_{R+1}, \dots, A_k)$ .

As customary, given a program  $P$  and a goal  $G$ , an SLD derivation for  $P \cup \{G\}$  consists of a possibly infinite sequence  $G = G_0, G_1, G_2, \dots$  of goals, a sequence  $C_1, C_2, \dots$  of properly renamed apart clauses of  $P$  (i.e.  $C_i$  has no common variables with any  $G_j$  nor  $C_j$  with  $j < i$ ), and a sequence of computed answer substitutions  $\theta_1, \theta_2, \dots$  (or most-general unifiers, *mgus* for short) such that each  $G_{i+1}$  is derived from  $G_i$  and  $C_{i+1}$  using  $\theta_{i+1}$ . Finally, we say that the SLD derivation is composed of the subsequent goals  $G_0, G_1, G_2, \dots$ .

A derivation step can be non-deterministic when  $A_R$  unifies with several clauses in  $P$ , giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in SLD trees. A finite derivation  $G = G_0, G_1, G_2, \dots, G_n$  is called successful if  $G_n$  is the empty goal, denoted  $\epsilon$ . In that

case  $\theta = \theta_1\theta_2 \dots \theta_n$  is called the computed answer for goal  $G$ . Such a derivation is called failing if it is not possible to perform a derivation step with  $G_n$ .

Executing a logic program  $P$  for a goal  $G$  consists in building an SLD tree for  $P \cup \{G\}$  and then extracting the computed answer substitutions from every non-failing branch of the tree.

**Constraint Logic Programming.** Constraint Logic Programming is a programming paradigm that extends Logic programming with *Constraint solving*. It augments the LP expressive power and application domain while maintaining its semantic properties (e.g., existence of a fixpoint semantics).

In CLP, the bodies of clauses may contain constraints in addition to ordinary literals. CLP integrates the use of a constraint solver to the operational semantics of logic programs. As a consequence of this extension, whereas in LP a computation state consists of a goal and a substitution, in CLP a computation state also contains a *constraint store*. The special *constraint* literals are stored in the *constraint store* instead of being solved according to SLD-resolution. The satisfiability of the constraint store is checked by a constraint solver. Then, we say that a CLP computation is successful if there is a derivation leading from the initial state  $S_0 = \langle G_0 \mid \text{true} \rangle$  (initially the constraint store is empty, i.e., *true*) to the final state  $S_n = \langle \epsilon \mid S \rangle$  such that  $\epsilon$  is the empty goal and  $S$  is satisfiable.

The CLP paradigm can be instantiated with many constraint domains. A constraint domain defines the class of constraints that can be used in a CLP program. Several constraint domains have been developed (e.g., for terms, strings, booleans, reals). A particularly useful constraint domain is CLP(FD) (Constraint Logic Programming over Finite Domains) [47]. CLP(FD) constraints are usually intended to be arithmetic constraints over finite integer domain variables. It has been applied to constraint satisfaction problems such as planning and scheduling [14,34]. Some features of CLP(FD) that make it suitable for TCG of programs working with integers are:

- It provides a mechanism to define the initial finite domain of variables as an interval over the integers and operations to further refine this initial domain.
- It provides a built-in *labeling* mechanism, which can be applied on a list of variables to find values for them such that the current constraint store is satisfied.

As we will see in the next section, our CLP-based TCG framework will rely on CLP(FD) to translate conditional statements over integer variables into CLP constraints. Moreover, the labeling mechanism is essential to concretize the obtained test cases in order to obtain concrete input data amenable to be used and validated by testing tools.

### 3.2 CLP-based Test Case Generation

CLP-based Test Case Generation advocates the use of CLP technology to perform test case generation of imperative object-oriented programs. The process has two phases. In the first phase, the imperative object-oriented program

under test is automatically transformed into an equivalent executable *CLP-translated* program. Instructions that manipulate heap-allocated data are represented by means of calls to specific *heap operations*. In the second phase, the CLP-translated program is symbolically executed using the standard CLP execution and constraint solving mechanism. The above-mentioned heap operations are also implemented in standard CLP, in a suitable way in order to support symbolic execution. The next two sections overview these two phases, which are also shown graphically in Figure 7.

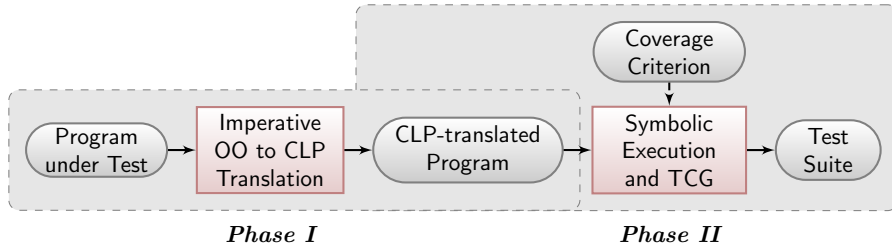


Fig. 7: CLP-based Test Case Generation Framework

*The Imperative Object-Oriented Language.* Although our approach is not tied to any particular imperative object-oriented language, we consider as the source language a subset of Java. For simplicity, we leave out of such subset features like concurrency, bitwise operations, static fields, access control (i.e., the use of `public`, `protected` and `private` modifiers) and primitive types besides integers and booleans. Nevertheless, these features can be relatively easy to handle in practice by our framework, except for concurrency, which is well-known to pose further challenges to symbolic execution and its scalability.

**CLP-translated Programs.** The translation of imperative object-oriented programs into equivalent CLP-translated programs has been subject of previous work (see, e.g., [2, 21]). Therefore, we will recap the features of the translated programs without going into deep details of how the translation is done. The translation is formally defined as follows:

**Definition 1 (CLP-translated program).** *The CLP-translated program for a given method  $m$  from the original imperative object-oriented program consists of a finite, non-empty set of predicates  $m, m_1, \dots, m_n$ . A predicate  $m_i$  is defined by a finite, non-empty set of mutually exclusive rules, each of the form  $m_i^k(In, Out, H_{in}, H_{out}, E) : -[g,]b_1, \dots, b_j.$ , where:*

1. *In and Out are, resp., the (possibly empty) list of input and output arguments.*
2.  *$H_{in}$  and  $H_{out}$  are, resp., the input and (possibly modified) output heaps.*

3.  $E$  is an exception flag that indicates whether the execution of  $m_i^k$  ends normally or with an uncaught exception.
4. If  $m_i$  is defined by more than one rule, then  $g$  is the constraint that guards the execution of  $m_i^k$ , i.e., it must hold for the execution of  $m_i^k$  to proceed.
5.  $b_1, \dots, b_j$  is a sequence of instructions including arithmetic operations, calls to other predicates and built-ins to operate on the heap, etc., as defined in Figure 8. As usual, an SSA transformation is performed [12].

---

$Clause ::= Pred(Args_{in}, Args_{out}, H_{in}, H_{out}, ExFlag) :- [G, B_1, B_2, \dots, B_n].$   
 $G ::= Num^* ROp Num^* \mid Ref_1^* \setminus == Ref_2^* \mid type(H, Ref^*, T)$   
 $B ::= Var \# = Num^* AOp Num^*$   
 $\quad \mid Pred(Args_{in}, Args_{out}, H_{in}, H_{out}, ExFlag)$   
 $\quad \mid new\_object(H_{in}, C^*, Ref^*, H_{out})$   
 $\quad \mid new\_array(H_{in}, T, Num^*, Ref^*, H_{out}) \mid length(H_{in}, Ref^*, Var)$   
 $\quad \mid get\_field(H_{in}, Ref^*, FSig, Var) \mid set\_field(H_{in}, Ref^*, FSig, Data^*, H_{out})$   
 $\quad \mid get\_array(H_{in}, Ref^*, Num^*, Var)$   
 $\quad \mid set\_array(H_{in}, Ref^*, Num^*, Data^*, H_{out})$   
 $Pred ::= Block \mid MSig \quad ROp ::= \#> \mid \#< \mid \#>= \mid \#<= \mid \# = \mid \# \setminus =$   
 $Args ::= [] \mid [Data^* | Args] \quad AOp ::= + \mid - \mid * \mid / \mid mod$   
 $Data ::= Num \mid Ref \mid ExFlag \quad T ::= bool \mid int \mid C \mid array(T)$   
 $Ref ::= null \mid r(Var) \quad FSig ::= C:FN$   
 $ExFlag ::= ok \mid exc(Var) \quad H ::= Var$

---

Fig. 8: Syntax of CLP-translated programs

Specifically, CLP-translated programs adhere to the grammar in Figure 8. As customary, terminals start with lowercase (or special symbols) and non-terminals start with uppercase; subscripts are provided just for clarity. Non-terminals  $Block$ ,  $Num$ ,  $Var$ ,  $FN$ ,  $MSig$ ,  $FSig$  and  $C$  denote, resp., the set of predicate names, numbers, variables, field names, method signatures, field signatures and class names. A clause indistinguishably defines either a method which appear in the original source program ( $MSig$ ), or an additional predicate which correspond to an intermediate block in the control flow graph of original program ( $Block$ ). A field signature  $FSig$  contains the class where the field is defined and the field name  $FN$ . An asterisk on a non-terminal denotes that it can be either as defined by the grammar or a (possibly constrained) variable (e.g.,  $Num^*$ , denotes that the term can be a number or a variable). Heap references are written as terms of the form  $r(Ref)$  or  $null$ . The operations that handle data in the heap are translated into built-in heap-related predicates.

Let us observe the following:



- There exists a one-to-one correspondence between blocks in the control flow graph of the original program and rules in the CLP-translated one.
- Mutual exclusion between the rules of a predicate is ensured either by means of mutually exclusive *guards*, or by information made explicit on the heads of rules, as usual in CLP. This makes the CLP-translated program deterministic, as the original imperative one is (point 4 in Definition 1).
- The global memory (or heap) is explicitly represented by means of logic variables. When a rule is invoked, the input heap  $H_{in}$  is received and, after executing the body of the rule, the heap might be modified, resulting in  $H_{out}$ . The operations that modify the heap will be shown later.
- Virtual method invocations are resolved at compile-time in the original imperative object-oriented language by looking up all possible runtime instances of the method. In the CLP-translated program, such invocations are translated into a choice of **type** instructions which check the actual object type, followed by the corresponding method invocation for each runtime instance.
- Exceptional behavior is handled explicitly in the CLP-translated program.

These observations will become more noticeable later on Example 7.

Note that the above definition proposes a translation to CLP as opposed to a translation to pure logic (e.g. to predicate logic or even to propositional logic, i.e., a logic that is not meant for “programming”). This is because we then want to execute the resulting translated programs to perform TCG and this requires, among other things, handling a constraint store and then generating actual data from such constraints. CLP is a natural paradigm to perform this task.

**Heap Operations.** Figure 9 summarizes the CLP implementation of the operations to create heap-allocated data structures (`new_object` and `new_array`) and to read and modify them (`getfield`, `set_array`, etc.) [22]. These operations rely on some auxiliary predicates (like deterministic versions of member `member_det`, of replace `replace_det`, and `nth0` and `replace_nth0` for arrays) which are quite standard and hence their implementation is not shown. For instance, a new object is created through a call to predicate `new_object( $H_{in}$ , Class, Ref,  $H_{out}$ )`, where  $H_{in}$  is the current heap, *Class* is the new object’s type, *Ref* is a unique reference in the heap for accessing the new object and  $H_{out}$  is the new heap after allocating the object. Read-only operations do not produce any output heap. For example, `get_field( $H_{in}$ , Ref, FSig, Var)` retrieves from  $H_{in}$  the value of the field identified by *FSig* from the object referenced by *Ref*, and returns its value in *Var* leaving the heap unchanged. Instruction `set_field( $H_{in}$ , Ref, FSig, Data,  $H_{out}$ )` sets the field identified by *FSig* from the object referenced by *Ref* to the value *Data*, and returns the modified heap  $H_{out}$ . The remaining operations are implemented likewise.

*The Heap term.* Our CLP-translated programs manipulate the heap as a black-box through its associated operations. The heaps generated and manipulated by

---

```

new_object(H,C,Ref,H') :- build_object(C,Ob), new_ref(Ref),
                           H' = [(Ref,Ob)|H].
new_array(H,T,L,Ref,H') :- build_array(T,L,Arr), new_ref(Ref),
                           H' = [(Ref,Arr)|H].

type(H,Ref,T) :- get_cell(H,Ref,Cell), Cell = object(T,_).
length(H,Ref,L) :- get_cell(H,Ref,Cell), Cell = array(_,L,_).
get_field(H,Ref,FSig,V) :- get_cell(H,Ref,Ob), FSig = C:FN,
                           Ob = object(T,Fields), subclass(T,C),
                           member_det(field(FN,V),Fields).
get_array(H,Ref,I,V) :- get_cell(H,Ref,Arr), Arr = array(_,_,Xs),
                       nth0(I,Xs,V).
set_field(H,Ref,FSig,V,H') :- get_cell(H,Ref,Ob), FSig = C:FN,
                              Ob = object(T,Fields), subclass(T,C),
                              replace_det(Fields,field(FN,_),field(FN,V),
                                           Fields'),
                              set_cell(H,Ref,object(T,Fields'),H').
set_array(H,Ref,I,V,H') :- get_cell(H,Ref,Arr), Arr = array(T,L,Xs),
                           replace_nth0(Xs,I,V,Xs'),
                           set_cell(H,Ref,array(T,L,Xs'),H').

```

---

```

get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).
set_cell([(Ref',_)|H],Ref,Cell,H') :- Ref == Ref', !,
                                       H' = [(Ref,Cell)|H].
set_cell([(Ref',Cell')|H'],Ref,Cell,H) :- H = [(Ref',Cell')|H'],
                                             set_cell(H',Ref,Cell,H').

```

---

Fig. 9: Heap operations for ground execution [22]

using these operations adhere to this grammar:

$$\begin{aligned}
\text{Heap} &::= [] \mid [\text{Loc} \mid \text{Heap}] \\
\text{Cell} &::= \text{object}(C^*, \text{Fields}^*) \mid \text{array}(T^*, \text{Num}^*, \text{Args}^*) \\
\text{Loc} &::= (\text{Num}^*, \text{Cell}) \\
\text{Fields} &::= [] \mid [\text{field}(\text{FN}, \text{Data}^*) \mid \text{Fields}^*]
\end{aligned}$$

The heap is represented as a list of locations which are pairs formed by a unique reference and a cell. Each cell can be an object or an array. An object contains its type and its list of fields, each of which is made of its signature and data content. An array contains its type, its length and its list of elements.

*Example 7.* Figure 10a shows the Java source code of class `List`, which implements a singly-linked list. The class contains one field `first` of type `Node`. As customary, `Node` is a recursive class with two fields: `data` of type `int` and `next` of type `Node`. Method `remAll` takes as argument an object `l` of type `List`, traverses it (outer while loop) and for each of its elements, traverses the `this` object and removes all their occurrences (inner loop). Figure 10b shows the equivalent

---

```

1 class Node {
2   int data;
3   Node next;
4 }
5 class List {
6   Node first;
7   void remAll(List l) {
8     // block1
9     Node lf = l.first;
10    // loop1
11    while (lf != null) {
12      // block2
13      Node prev = null;
14      Node p = null;
15      Node next = first;
16      // loop2
17      while (next != null) {
18        // block3
19        prev = p;
20        p = next;
21        next = next.next;
22        // if1
23        if (p.data == lf.data)
24          // if2
25          if (prev == null) {
26            first = next;
27            p = null;
28          } else {
29            prev.next = next;
30            p = prev;
31          }
32      }
33      // block4
34      lf = lf.next;
35    }
36  }
37 }

```

---

(a) Java source code

---

```

remAll([r(Th),L],[ ],[ ],Hi,Ho,E) :-
  block1([Th,L],Hi,Ho,E).
block1([Th,r(L)],Hi,Ho,E) :-
  get_field(Hi,L,first,LfR),
  loop1([Th,L,LfR],Hi,Ho,E).
block1([Th,null],Hi,Ho,E) :-
  create_object(Hi,'NPE',E,Ho).
loop1([Th,L,null],H,H,ok).
loop1([Th,L,r(Lf)],Hi,Ho,E) :-
  block2([Th,L,Lf],Hi,Ho,E).
block2([Th,L,Lf],Hi,Ho,E) :-
  get_field(Hi,Th,first,FR),
  loop2([Th,L,Lf,null,null,FR],Hi,Ho,E).
loop2([Th,L,Lf,Prev,P,null],Hi,Ho,E) :-
  block4([Th,L,Lf],Hi,Ho,E).
loop2([Th,L,Lf,Prev,P,r(F)],Hi,Ho,E) :-
  block3([Th,L,Lf,P,F],Hi,Ho,E).
block3([Th,L,Lf,P,F],Hi,Ho,E) :-
  get_field(Hi,F,next,FRN),
  get_field(Hi,F,data,A),
  get_field(Hi,Lf,data,B),
  if1([A,B,Th,L,Lf,P,F,FRN],Hi,Ho,E).
if1([A,B,Th,L,Lf,Prev,P,FRN],Hi,Ho,E) :-
  #\=(A,B),
  loop2([Th,L,Lf,Prev,P,FRN],Hi,Ho,E).
if1([A,A,Th,L,Lf,Prev,P,FRN],Hi,Ho,E) :-
  if2([Th,L,Lf,Prev,P,FRN],Hi,Ho,E).
if2([Th,L,Lf,r(F),P,N],Hi,Ho,E) :-
  set_field(Hi,F,next,N,H2),
  loop2([Th,L,Lf,F,F,N],H2,Ho,E).
if2([Th,L,Lf,null,P,N],Hi,Ho,E) :-
  set_field(Hi,Th,first,N,H2),
  loop2([Th,L,Lf,null,null,N],H2,Ho,E).
block4([Th,L,Lf],Hi,Ho,E) :-
  get_field(Hi,Lf,next,LfRN),
  loop1([Th,L,LfRN],Hi,Ho,E).

```

---

(b) CLP-translation

Fig. 10: CLP-based TCG example

(simplified and pretty-printed) CLP-translated code for method `remAll`. Let us observe some of the main features of the CLP-translated program. The `if` statement in line 23 is translated into two mutually exclusive rules (predicate `if1`) guarded by an arithmetic condition. Similarly, the `if` statement in line 25 is translated into predicate `if2`, implemented by two rules whose mutual exclusion

is guaranteed by terms `null` and `r(_)` appearing in each rule head. Observe that iteration in the original program (`while` constructions) is translated into recursive predicates. For instance, the head of the inner `while` loop is translated into predicate `loop2`, its condition is guarded by the rules of predicate `cond2` (`null` or `r(_)`), and recursive calls are made from predicates `if1` (first rule) and `if2` (both rules). Finally, exception handling is made explicit in the CLP-translated program; the second rule of predicate `block1` encodes the runtime null pointer exception (`'NPE'`) that raises if the input argument `1` is `null`.  $\square$

### 3.3 Semantics of CLP-translated Programs

The standard CLP execution mechanism suffices to execute the CLP-translated programs. Let us focus on the concrete execution of CLP-translated programs by assuming that all input parameters of the predicate to be executed (i.e.,  $In$  and  $H_{in}$ ) are fully instantiated in the initial input state.

Let  $M$  be a method in the original imperative program,  $m$  be its corresponding predicate in the CLP-translated program  $P$ , and  $P'$  be the union of  $P$  and the predicates in Figure 9. As explained in the previous section, the operational semantics of the CLP program  $P'$  can be defined in terms of *derivations*. A derivation is a sequence of reductions between states  $S_0 \rightarrow_P S_1 \rightarrow_P \dots \rightarrow_P S_n$ , also denoted  $S_0 \rightarrow_{P'} S_n$ , where a *state*  $\langle G \mid \theta \rangle$  consists of a goal  $G$  and a constraint store  $\theta$ . The concrete execution of  $m$  with input  $\theta$  is the derivation  $S_0 \rightarrow S_n$ , where  $S_0 = \langle m(In, Out, H_{in}, H_{out}, ExFlag) \mid \theta \rangle$  and  $\theta$  initializes  $In$  and  $H_{in}$  to be fully ground. If the derivation successfully terminates, then  $S_n = \langle \epsilon \mid \theta' \rangle$  and  $\theta'$  is the *output constraint store*.

This definition of concrete execution relies on the correctness of the translation algorithm, which must guarantee that the CLP-translated program captures the same semantics of the original imperative one [2, 21].

*Example 8.* The following is a *correct* input state for predicate `remAll/5`:

$\langle \text{remAll}([r(1), \text{null}], \text{Out},$   
 $[(1, \text{object('List', [field('Node':first, null)]))], \text{Hout}, \text{E}) \mid \text{true} \rangle$

Observe that the list of input arguments and the input heap (both underlined) are fully instantiated. Argument `r(1)` corresponds to the implicit reference to the *this* object, which appears in the input heap term with its field `first` being instantiated to `null`. Concrete execution on this input state yields a final state in which:

$\text{Out} = [] \wedge$   
 $\text{Hout} = [(1, \text{object('List', [field('Node':first, null)]))},$   
 $\quad (2, \text{object('NPE', [ ])})] \wedge$   
 $\text{E} = \text{exc}(2)$

Notice that in this final state, a new object of type `NPE` (Null Pointer Exception) is created in the heap. The fact that the execution ends with an uncaught exception is indicated in flag `E`.  $\square$

### 3.4 Symbolic Execution

When the source imperative language does not support dynamic memory, symbolic execution of the CLP-translated programs is attained by simply using the standard CLP execution mechanism to run the main goal (i.e., the predicate name after the method under test) *with all arguments being free variables*. The inherent constraint solving and backtracking mechanisms of CLP allow to keep track of path conditions (or constraint stores), failing and backtracking when unsatisfiable constraints are hit, hence discarding such execution paths; and succeeding when satisfiable constraints lead to a terminating state in the program, which in the context of TCG implies that a new test case is generated.

However, in the case of heap-manipulating programs, the heap-related operations presented in Figure 9 fall short to generate arbitrary heap-allocated data structures and all possible heap shapes when accessing symbolic references. This is a well-known problem in TCG by symbolic execution. A naive solution to this problem could be to fully initialize all the reference parameters prior to symbolic execution. However, this would require imposing bounds on the size of input data structures, which is highly undesirable. Doing so would circumscribe the symbolic search space, hence jeopardizing the overall effectiveness of the technique.

*Lazy Initialization.* A straightforward generalization of predicate `get_cell` in Figure 9 provides a simple and flexible solution to the problem of handling arbitrary input data structures during symbolic execution, and constitutes a quite natural implementation of the *lazy initialization* technique in our CLP-based framework. Figure 11 shows the new implementation of the `get_cell` operations; observe that we have added just two new rules to the implementation shown in Figure 9.

---

```

get_cell(H,Ref,Cell) :- var(H), !, H = [(Ref,Cell)|_].
get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
get_cell([(Ref',Cell')|_],Ref,Cell) :- var(Ref), var(Ref'), Ref = Ref',
                                     Cell = Cell'.
get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).

```

---

Fig. 11: Redefining `get_cell` operations for symbolic execution [22]

The intuitive idea is that the heap during symbolic execution contains two parts: the *known part*, with the cells that have been explicitly created during symbolic execution appearing at the beginning of the list, and the *unknown part*,

which is a logic variable (tail of the list) in which new data can be added. Importantly, the definition of `get_cell/3` distinguishes two situations when searching for a reference: (i) It finds it in the known part (second clause), meaning that the reference has already been accessed earlier (note the use of syntactic equality rather than unification, since references at execution time can be variables); or (ii) It reaches the unknown part of the heap (a logic variable), and it allocates the reference (in this case a variable) there (first clause). The third clause of `get_cell/3` allows to consider all possible aliasing configurations among references. In essence, `get_cell/3` is therefore a CLP implementation of *lazy initialization*.

Let us illustrate the use of lazy initialization in symbolic execution with an example.

*Example 9.* Figure 12 shows the CLP-translated program for method `mist` from Example 2. Let `mist(In,Out,Hin,Hout,E)` be the initial goal for symbolic execution. Observe that the input heap `Hin` is a free variable (i.e., fully unknown). Let us choose rule `mist`<sup>1</sup>. By doing so, the list of input arguments `In` gets instantiated to `[r(A),R2]`, which indicates that the first argument is a reference to an existing object in the heap, as opposed to the `null` reference in rule `mist`<sup>2</sup>. The execution of the `get_field` instruction imposes new constraints on the shape of the input heap. Namely, `Hin` is partially instantiated to `[(A,object('C',[field(f,F)|M]))|N]`. Observe that there is still an unknown part in the heap (variable `N`). Also, observe that the list of fields for object `A` is also represented by an open list, meaning that there might be other fields in this object, but nothing has been learned about them yet.

Now, let us assume that the execution proceeds with rules `if`<sup>1</sup> and `then`<sup>1</sup>. At this point, the second argument is also set to be a valid reference `r(B)`. The execution of the `set_field` will internally reach predicate `get_cell` (Figure 11), leading to consider two possibilities:

- References `R1=r(A)` and `R2=r(B)` point to two different objects in the heap. In this case, the resulting output heap is

$$\text{Hout} = [(A, \text{object}('C', [\text{field}(f, D1) | M])), \\ (B, \text{object}('C', [\text{field}(f, 1) | P])) | N],$$

and the constraint store is  $\theta = \{D1 > 0\}$ .

- References `R1=r(A)` and `R2=r(A)` point to the same object in the heap, i.e., they are aliased. Here, the resulting output heap is `Hout=[(A,object('C',[field(f,D1)|M]))|N]`, with  $\theta = \{D1 > 0\}$ .  $\square$

To conclude this section, let us now provide a definition for symbolic execution in terms of the CLP derivation tree of the CLP-translated program extended with built-in operations to handle dynamic memory:

**Definition 2 (Symbolic Execution).** *Let  $M$  be a method,  $m$  be its corresponding predicate from its associated CLP-translated program  $P$ , and  $P'$  be the union of  $P$  and the set of predicates in Figure 9. The symbolic execution of  $m$  is*

---

```

mist1([r(A),R2],[ ],Hin,Hout,E) :-
    get_field(Hin,A,f,D1),
    if([D1,A,R2],Hin,Hout,E).
mist2([null,R2],[ ],Hin,Hout,exc(Exc)) :-
    create_object(Hin,'NPE',Exc,Hout).
if1([D1,A,R2],Hin,Hout,E) :-
    #>(D1,0),
    then([R2],Hin,Hout,E).
if2([D1,A,R2],Hin,Hout,ok) :-
    #<=(D1,0),
    set_field(Hin,A,f,0,Hout),
then1([r(B)],Hin,Hout,ok) :-
    set_field(Hin,B,f,1,Hout).
then2([null],Hin,Hout,exc(Exc)) :-
    create_object(Hin,'NPE',Exc,Hout).

```

---

Fig. 12: CLP-translated program for method `mist` (Example 2)

the CLP derivation tree, denoted as  $\mathcal{T}_m$ , with root  $m(In, Out, H_{in}, H_{out}, E)$  and initial constraint store  $\theta = \{\}$  obtained using  $P'$ .

### 3.5 Test Case Generation

When handling realistic programs, it is well-known that the symbolic execution tree to be explored is in general infinite. This is because iterative constructs such as loops and recursion, whose number of iterations depend on input arguments, usually induce an infinite number of execution paths when executed with symbolic input values. It is therefore essential to establish a *termination criterion*. Such a termination criterion can be expressed in different forms. For instance, a computation time budget can be established, or an explicit bound on the depth of the symbolic execution tree can be imposed (called *depth- $k$*  criterion). In this thesis, we adopt a more code-oriented termination criterion. Specifically, we impose an upper bound  $k$  on the number of times each loop is iterated. As a byproduct of imposing such a bound, the *loop- $k$*  structural coverage criterion below is satisfied.

**Finite symbolic execution tree, test case, and TCG.** Let us now establish definitions for key concepts of our approach:

**Definition 3 (Finite symbolic execution tree, test case, and TCG).** Let  $m$  be the corresponding predicate for a method  $M$  in a CLP-translated program  $P$ , and let  $\mathcal{C}$  be a termination criterion.

- $\mathcal{T}_m^{\mathcal{C}}$  is the finite and possibly incomplete symbolic execution tree of  $m$  with root  $m(In, Out, H_{in}, H_{out}, EF)$  w.r.t.  $\mathcal{C}$ .

Table 4: Test cases for method `remAll`

N	Input Heap	Output Heap	Constraint Store	EF
1	this l.first = null	this l.first = null	$\emptyset$	ok
2	this.first = null l.first $\rightarrow$ (A) $\rightarrow$ null	this.first = null l.first $\rightarrow$ (A) $\rightarrow$ null	$\emptyset$	ok
3	this.first $\rightarrow$ (A) $\rightarrow$ null l.first $\rightarrow$ (B) $\rightarrow$ null	this.first $\rightarrow$ (A) $\rightarrow$ null l.first $\rightarrow$ (B) $\rightarrow$ null	$\{A \neq B\}$	ok
4	this.first $\rightarrow$ (A) $\rightarrow$ null l.first $\rightarrow$ (A) $\rightarrow$ null	this.first = null l.first $\rightarrow$ (A) $\rightarrow$ null	$\emptyset$	ok
5	this l $\rightarrow$ null	-	$\emptyset$	exc
6	this.first $\rightarrow$ (A) $\rightarrow$ null l = this	this.first = null l = this	$\emptyset$	ok
7	this.first $\rightarrow$ (A) $\rightarrow$ null l.first $\rightarrow$ (A) $\rightarrow$ null	this.first = null l.first $\rightarrow$ (A) $\rightarrow$ null	$\emptyset$	ok

- Let  $b$  be a successful (terminating) path in  $\mathcal{T}_m^C$ . A test case for  $m$  w.r.t.  $\mathcal{C}$  is a 6-tuple of the form:  $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta \rangle$ , where  $\sigma$  and  $\theta$  are, resp., the substitution and the constraint store associated to  $b$ .
- TCG is the process of generating the set of test cases obtained for all successful (terminating) paths in  $\mathcal{T}_m^C$ .

In the remainder of this dissertation, we comply with the above abstract (symbolic) definition of *test case*, hence adopting a non-standard use of the term “test case”. Standard test cases are concrete, i.e., actual input values on which the program under test can be run. In contrast, in this thesis a *test case* represents the class of inputs that will follow the same execution path, characterized by a path condition (and symbolic expressions for variables). A *test suite* is hence a set of test cases that characterizes all symbolic execution paths explored by symbolic execution using a particular termination criterion. Nevertheless, it is possible to produce actual values from the obtained *symbolic* test cases. This can be done in a straightforward subsequent stage in our framework. Namely, we can use the *labeling* mechanisms of standard *clpfd* domains to assign concrete values to all variables which satisfy the path condition, thus solving it. As a result of this last step, concrete and executable test cases are obtained.

*Example 10.* The test suite generated for method `remAll` for a *loop-1* coverage criterion is shown in Table 4. The first 5 cases are generated without considering aliasing of references. By doing so, the last two cases are also generated. Let us explain in detail three of the obtained test cases:



- **Case 3.** Corresponds to the path in which both the `this` list and the input list `l` contain just one element. The constraint  $\{A \neq B\}$  indicates that fields `this.first.data` and `l.first.data` must have different values. The output heap is the same as the input heap, which means that the heap remains unchanged at the end of the execution path represented by this test case (although it may have suffered changes in intermediate derivations).
- **Case 4.** The input heap is the almost same as in case 3, but here, the symbolic variables corresponding to `this.first.data` and `l.first.data` are unified (variable  $A$ ), meaning that their values are the same. In the output heap, notice that the first node from the `this` list has been removed.
- **Case 7.** Reference fields `this.first` and `l.first` are aliased. That is, they point to the same `Node` object in the heap. Removing element  $A$  from the `this` list boils down to setting reference `this.first` to `null`, leaving the object in the heap intact.

Finally, as mentioned before, by solving the constraint system and applying labeling on the variables involved, concrete inputs can be obtained. A concrete instantiation for this test case would consist of the following input heap  $\{\text{this.first} \rightarrow 1 \rightarrow \text{null}, \text{l.first} \rightarrow 2 \rightarrow \text{null}\}$  where variables  $A$  and  $B$  have been assigned concrete values 1 and 2, respectively, such that the constraint store  $A \neq B$  is satisfied. As the test case specifies, the heap in the concrete output state remains unchanged.  $\square$

**The PET System.** PET (Partial Evaluation-based Test case generator) is a system that implements the CLP-based TCG framework described in this chapter. It is fully implemented in SWI-Prolog [48] and uses the CLP(FD) library [47] (Constraint Logic Programming over Finite Domains) as constraint solver. Some of the important features of the PET system are:

- It is generic. Provided that appropriate CLP translations are available, PET can work with other imperative object-oriented languages. That is, once the CLP translation is done, the language features are abstracted away. That is to say, the TCG phase of the approach implemented in PET is language independent. In this way, we elude the difficulties of explicitly dealing with features like recursion, procedure calls, dynamic memory allocation, exceptions, etc., whose treatment may differ from one language to another.
- It is flexible. Different termination (coverage) criteria can be easily incorporated to the PET system. These criteria are written in PET as predicates which are permanently checked during TCG. Adding new criteria consists in implementing such a predicate, which requires only basic knowledge of logic programming.
- It is incremental. One of the artifacts that the PET system generates is a test case generator. To the best of our knowledge, this is a unique feature in a TCG tool nowadays. Namely, PET allows to extend test suites by exploring further in the symbolic execution tree in an on-demand fashion. In

other words, PET allows to incrementally relax the imposed termination criterion to explore symbolic execution paths that were initially pruned by the termination criterion.

The PET system is available for download as open-source software and for online use through its web interface at <http://costa.ls.fi.upm.es/pet>. Furthermore, an Eclipse plugin called jPET [3] is available. jPET supports full sequential Java and some of its interesting features are:

- Interactive test case visualization. jPET integrates a test case viewer to allow an intuitive, interactive visualization of the information contained in test cases. This includes objects and arrays involved in the input and output heap terms.
- Trace highlighting. On selection of a particular test case, jPET highlights the sequence of instructions in the original Java source code that the test case exercises. Alternatively, a *trace debugging* feature allows for a step-by-step highlighting of the source code, as in the traditional style of code debugging.
- Parsing of method preconditions written in JML [28]. jPET enables the specification of conditions on the input arguments of methods. These conditions are written in a subset of JML (Java Modeling Language), the standard specification language within software verification of Java. Using preconditions allows steering symbolic execution towards interesting parts of the program under test, ignoring others that are less interesting.
- Generation of JUnit. JUnit is a Unit Testing Framework for Java, which provides a set of classes to support writing, executing and reusing test cases. jPET generates self-contained JUnit test cases, as shown in Example 4. Whereas those unit tests therein are rather simple, the generation of JUnit code for heap-manipulating programs is much more challenging, as it often involves the need to synthesize the input and output heaps and compare the output heap stored in the test case with the resulting heap after the execution of the test.

### 3.6 Guided CLP-based TCG

Whereas standard TCG by symbolic execution aims to cover *all* feasible paths of the program under test w.r.t. a termination criterion, in guided TCG, the termination criterion is combined with a *selection criterion*. To that end, the concept of *coverage criterion* is redefined to be a pair of two components  $\langle TC, SC \rangle$ .  $TC$  is a *termination criterion* that, as discussed earlier, ensures finiteness of symbolic execution. This can be done either based on execution steps or on loop iterations. Again, let us adhere to *loop- $k$* , which limits to a threshold  $k$  the number of allowed loop iterations and/or recursive calls (of each concrete loop or recursive method).  $SC$  is a *selection criterion* that determines which test cases the TCG must produce. In guided TCG this will steer symbolic execution towards the paths that should be explored. In particular, we consider the following two coverage criteria:

- **all-local-paths**: It requires that all *local* execution paths within the method under test are exercised up to a *loop-k* limit. This has a potential interest in the context of unit testing, where each method must be tested in isolation.
- **program-points(P)**: Given a set of program points P, it requires that all of them are exercised by at least one test case up to a *loop-k* limit. This criterion is the most appropriate choice for bug-detection and reachability verification purposes. A particular case of it is *statement coverage* (up to a limit), where all statements in a program or method must be exercised.

This section develops a concrete methodology to incorporate selection criteria into the CLP-based TCG framework. To that end, we could employ a post-processing phase where only the test cases that are sufficient to satisfy the selection criterion are selected by looking at their traces. This is however not an appropriate solution in general due to the exponential explosion of the paths that have to be explored in symbolic execution. Instead, we now aim at using the selection criterion to drive the TCG process towards satisfying paths, stressing to avoid as much as possible the exploration of irrelevant and redundant ones. The key idea that allows us to guide the TCG process is to pass *trace terms* as input arguments to symbolic execution. These trace terms can be complete or partial, which allows guiding completely or partially, the symbolic execution towards specific paths.

First, let us define the notion of *trace term* and update Definition 1 to add a trace term as an additional argument to each rule of the CLP-translated program, which enables us to keep track of the sequence of rules that are symbolically executed. Notice that trace terms are not cardinal components in the translated program, but rather a supplementary argument with a central role in this chapter.

**Definition 4 (CLP-translated program with traces).** *Given the rule of Definition 1, its CLP-translation with trace is:  $m(In, Out, H_{in}, H_{out}, EF, T) : - g, b'_1, \dots, b'_n$  where:*

- $In, Out, H_{in}, H_{out}$  and  $EF$  remain as in Definition 1.
- $T$  is the trace term for  $m$  of the form  $m(k, P, \langle T_{c_i}, \dots, T_{c_m} \rangle)$ , where
  - $P$  is the (possibly empty) list of trace parameters, i.e., the subset of the variables in rule  $m^k$  on which the resource consumption depends.
  - $c_i, \dots, c_m$  is the (possibly empty) subsequence of method calls in  $b_1, \dots, b_n$ .
  - $T_{c_j}$  is a free logic variable representing the trace term associated to the call  $c_j$ .
- Calls in the body of the rule are extended with their corresponding trace terms, i.e., for all  $1 \leq j \leq n$ , if  $b_j \equiv p(I_p, O_p, H_{in_p}, H_{out_p})$ , then  $b'_j \equiv p(I_p, O_p, H_{in_p}, H_{out_p}, T_{c_j})$ ; otherwise  $b'_j \equiv b_j$ .

Now, let us revisit the definition of test case and TCG (Definition 3) to incorporate the notion of *trace* as an input argument for symbolic execution.

**Definition 5 (Test case with trace and TCG).** *Given a method  $m$ , a termination criterion  $\mathcal{C}$  and a successful (terminating) path  $b$  in the symbolic execution tree  $\mathcal{T}_m^{\mathcal{C}}$  with root  $m(In, Out, H_{in}, H_{out}, EF, T)$ , a test case with trace for  $m$*

w.r.t.  $\mathcal{C}$  is a 6-tuple of the form:  $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \sigma(T), \theta \rangle$ , where  $\sigma$  and  $\theta$  are, resp., the set of bindings and the constraint store associated to  $b$ . TCG generates the set of test cases with traces obtained for all successful paths in  $\mathcal{T}_m^C$ .

**Trace-guided TCG.** Given a method  $m$ , a coverage criterion  $\mathcal{C} = \langle TC, SC \rangle$ , and a (possibly partial) trace  $\pi$ , trace-guided TCG generates the set **tgTCG** of test cases obtained for all successful branches of  $m$  using  $\pi$  as a guiding input argument for symbolic execution. Observe that the TCG guided by one trace  $\pi$  generates: (a) exactly one test case if  $\pi$  is complete and corresponds to a feasible path; (b) none if  $\pi$  is unfeasible; or (c) possibly several test cases if  $\pi$  is partial. In the latter case the traces of all test cases are instantiations of the partial trace.

For convenience, let us also define  $\text{firstOf-tgTCG}(m, TC, \pi)$  to be the unary set containing the leftmost successful branch of the symbolic execution tree of  $m$ . Now, by relying on the existence of a trace generator *TraceGen* that generates, on demand and one by one, (possibly partial) traces according to  $\mathcal{C}$ , we define in Algorithm 1 a generic scheme for guided TCG.

---

**Algorithm 1** Generic scheme for guided TCG

---

```

Input:  $M$ , and  $\langle TC, SC \rangle$ 
TestCases = {}
while TraceGen has more traces and TestCases does not satisfy SC
    Ask TraceGen to generate a new trace in Trace
    TestCases = TestCases  $\cup$  firstOf-tgTCG( $M, TC, Trace$ )
Output: TestCases

```

---

The intuition is as follows: the trace generator generates a trace, possibly using for that  $SC$ ,  $TC$  and the current *TestCases*. If the generated trace is feasible, then the first solution of its trace-guided TCG is added to the set of test cases. The process finishes either when  $SC$  is satisfied, or when the trace generator has already generated all traces up to  $TC$ . If the trace generator is complete (see below), this means that  $SC$  cannot be satisfied within the limit imposed by  $TC$ . Observe that for some selection criteria, e.g., *all-local-paths*, the calls to  $\text{firstOf-tgTCG}$  can be computed in parallel.

*Example 11.* Figure 13a shows a Java program made up of three methods: *lcm* calculates the least common multiple of two integers, *gcd* calculates the greatest common divisor of two integers, and *abs* returns the absolute value of an integer. Figure 13b shows the equivalent CLP-translated program. Method *lcm* is translated into predicates *lcm*, *cont*, *try* and *div*. As per Section 3.2, the translation preserves the control flow of the program and transforms iteration into recursion (e.g. method *gcd*). Note that the example has been chosen deliberately small and simple to ease comprehension. Let us consider the TCG for method *lcm* with *program-points* for points  $\mu$  and  $\kappa$  as selection criterion. Let us assume that

<pre> int lcm(int a,int b) {   if (a &lt; b) {     int aux = a;     a = b;     b = aux;   }   int d = gcd(a,b);   try {     return abs(a*b)/d;   } catch (Exception e) {     return -1;   } }  int gcd(int a,int b) {   int res;   while (b != 0) {     res = a%b;     a = b;     b = res;   };   return abs(a); }  int abs(int a) {   if (a &gt;= 0)     return a;   else     return -a; } </pre>	<pre> lcm([A,B],[R],_,_,E,lcm(1,[T])) :-   A #&gt;= B,   cont([A,B],[R],_,_,E,T). lcm([A,B],[R],_,_,E,lcm(2,[T])) :-   A #&lt; B,   cont([B,A],[R],_,_,E,T). cont([A,B],[R],_,_,E,cont(1,[T,V])) :-   gcd([A,B],[G],_,_,E,T),   try([A,B,G],[R],_,_,E,V). try([A,B,G],[R],_,_,E,try(1,[T,V])) :-   M #= A*B,   abs([M],[S],_,_,E,T),   div([S,G],[R],_,_,E,V). try([A,B,G],[R],_,_,exc,try(2,[ ])). div([A,B],[R],_,_,ok,div(1,[ ])) :-   B #\= 0,   R #= A/B. div([A,0],[-1],_,_,catch,div(2,[ ])). gcd([A,B],[D],_,_,E,gcd(1,[T])) :-   loop([A,B],[D],_,_,E,T). loop([A,0],[F],_,_,E,loop(1,[T])) :-   abs([A],[F],_,_,E,T). loop([A,B],[E],_,_,G,loop(2,[T])) :-   B #\= 0,   body([A,B],[E],_,_,G,T). body([A,B],[R],_,_,E,body(1,[T])) :-   B #\= 0,   M #= A mod B,   loop([B,M],[R],_,_,E,T). body([A,0],[R],_,_,exc,body(2,[ ])). abs([A],[A],_,_,ok,abs(1,[ ])) :-   A #&gt;= 0. abs([A],[-A],_,_,ok,abs(2,[ ])) :-   A #&lt; 0. </pre>
(a) Java source code	(b) CLP-translation

Fig. 13: Guided TCG Example: Java (left) and CLP-translated (right) programs.

the trace generator starts generating the following two traces:

$$\begin{aligned}
t_1 &: \text{lcm}(1, [\text{cont}(1, [\text{G}, \text{check}(1, [A, \text{div}(2, [])])])]) \\
t_2 &: \text{lcm}(2, [\text{cont}(1, [\text{G}, \text{check}(1, [A, \text{div}(2, [])])])])
\end{aligned}$$

The first iteration does not add any test case since trace  $t_1$  is unfeasible. Trace  $t_2$  is proved feasible and a test case is generated. The selection criterion is now satisfied and therefore the process finishes. The following test case is obtained for the program-points criterion for method `lcm` and program points  $\textcircled{u}$  and  $\textcircled{k}$ . This

particular case illustrates specially well how guided TCG can reduce the number of produced test cases through adequate control of the selection criterion.

<i>Constraint store</i>	<i>Trace</i>
$\{A=B=0, Out=-1\}$	$1cm(1, [cont(1, [gcd(1, [loop(1, [abs(1, []))])]),$ $try(1, [abs(1, []), div(2, []))])])])$

□

There are two properties of high importance in guided TCG, *completeness* and *effectiveness*. Intuitively, a concrete instantiation of the guided TCG scheme is *complete* if it never reports that the coverage criterion is not satisfied when it is indeed satisfiable. *Effectiveness* is related to the number of iterations the algorithm performs. These two properties depend completely on the trace generator. A trace generator is *complete* if it produces an over-approximation of the set of traces satisfying the coverage criterion. Its *effectiveness* depends on the number of redundant and/or unfeasible traces it generates: the larger the number, the less effective the trace generator.

**Trace Generators for Structural Coverage Criteria.** Let us now describe a general approach to build complete and effective trace generators for structural coverage criteria by means of program transformations. Then, we describe in detail an instantiation for the **all-local-paths** coverage criteria.

The *trace-abstraction* of a program can be defined as follows. Given a CLP-translated program with traces  $P$ , its trace-abstraction is obtained as follows: for every rule of  $P$ , (1) remove all atoms in the body of the rule except those corresponding to rule calls, and (2) remove all arguments from the head and from the surviving atoms of (1) except the last one (i.e., the trace term).

*Example 12.* Figure 14 shows the trace-abstraction for the CLP-translated program of Figure 13b. Observe that the trace-abstraction basically corresponds the control-flow graph of the CLP-translated program. □

The trace-abstraction can be directly used as a trace generator as follows: (1) Apply the termination criterion in order to ensure finiteness of the process. (2) Select, in a post-processing, those traces that satisfy the selection criterion. Such a trace generator produces on backtracking a superset of the set of traces of the program satisfying the coverage criterion. Note that, this can be done as long as the criteria are structural. The obtained trace generator is by definition complete. However, it can be very ineffective and inefficient due to the large number of unfeasible and/or unnecessary traces that it can generate.

In the following, we propose a concrete, and more effective, instantiation for the **all-local-paths** coverage criteria. As we will see, this is done by taking advantage of the notion of partial traces and the implicit information on the concrete coverage criteria. A concrete instantiation for the **program-points** coverage criteria is described at [39].

---

```

lcm(lcm(1, [T])) :- cont(T).
lcm(lcm(2, [T])) :- cont(T).
cont(cont(1, [T, V])) :- gcd(T), try(V).
try(try(1, [T, V])) :- abs(T), div(V).
try(try(2, [])).
div(div(1, [])).
div(div(2, [])).
gcd(gcd(1, [T])) :- loop(T).
loop(loop(1, [T])) :- abs(T).
loop(loop(2, [T])) :- body(T).
body(body(1, [T])) :- loop(T).
body(body(2, [])).
abs(abs(1, [])).
abs(abs(2, [])).

```

---

Fig. 14: Trace-abstraction

**An Instantiation for the all-local-paths Coverage Criterion.** Let us start from the trace-abstraction program and apply a syntactic program slicing which removes from it the rules that do not belong to the considered method.

**Definition 6 (slicing for all-local-paths coverage criterion).** *Given a trace-abstraction program  $P$  and an entry method  $M$ :*

1. *Remove from  $P$  all rules that do not belong to method  $M$ .*
2. *In the bodies of remaining rules, remove all calls to rules which are not in  $P$ .*

The obtained sliced trace-abstraction, together with the termination criterion, can be used as a trace generator for the all-local-paths criterion for a method. The generated traces will have free variables in those trace arguments that correspond to the execution of other methods, if any.

<pre> lcm(lcm(1, [T])) :- cont(T). lcm(lcm(2, [T])) :- cont(T). cont(cont(1, [G, T])) :- try(T). try(try(1, [A, T])) :- div(T). try(try(2, [])). div(div(1, [])). div(div(2, [])). </pre>	<pre> lcm(1, [cont(1, [G, try(1, [A, div(1, [])])])]) lcm(1, [cont(1, [G, try(1, [A, div(2, [])])])]) lcm(1, [cont(1, [G, try(2, [])])]) lcm(2, [cont(1, [G, try(1, [A, div(1, [])])])]) lcm(2, [cont(1, [G, try(1, [A, div(2, [])])])]) lcm(2, [cont(1, [G, try(2, [])])]) </pre>
---	--

Fig. 15: Slicing of method lcm for all-local-paths criterion.

*Example 13.* Figure 15 shows on the left the sliced trace-abstraction for method `lcm`. On the right is the finite set of traces that is obtained from such trace-abstraction for any *loop-k* termination criterion. Observe that the free variables `G`, resp. `A`, correspond to the sliced away calls to methods `gcd` and `abs`.  $\square$

Let us define the predicates: `computeSlicedProgram(M)`, that computes the sliced trace-abstraction for method *M* as in Definition 6; `generateTrace(M,TC,Trace)`, that returns in its third argument, on backtracking, all partial traces computed using such sliced trace-abstraction, limited by the termination criterion *TC*; and `traceGuidedTCG(M,TC,Trace,TestCase)`, which computes on backtracking the set `tgTCG` (definition of Trace-guided TCG above), failing if the set is empty, and instantiating on success `TestCase` and `Trace` (in case it was partial). The guided TCG scheme in Algorithm 1, instantiated for the *all-local-paths* criterion, can be implemented in Prolog as follows:

```
(1) guidedTCG(M,TC) :-
(2)     computeSlicedProgram(M),
(3)     generateTrace(M,TC,Trace),
(4)     once(traceGuidedTCG(M,Trace,TC,TestCase)),
(5)     assert(testCase(M,TestCase,Trace)),
(6)     fail.
(7) guidedTCG(_,_).
```

Intuitively, given a (possibly partial) trace generated in line (3), if the call in line (4) fails, then the next trace is tried. Otherwise, the generated test case is asserted with its corresponding trace which is now fully instantiated (in case it was partial). The process finishes when `generateTrace/3` has computed all traces, in which case it fails, making the program exiting through line (7).

*Example 14.* The following test cases are obtained for the *all-local-paths* criterion for method `lcm`:

Constraint store	Trace
{A>=B}	<code>lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[]))])],try(1,[abs(1,[]),div(1,[])]))])]</code>
{A=B=0,Out=-1}	<code>lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[]))])],try(1,[abs(1,[]),div(2,[])]))])]</code>
{B>A}	<code>lcm(2,[cont(1,[gcd(1,[loop(1,[abs(1,[]))])],try(1,[abs(1,[]),div(1,[])]))])]</code>

This set of 3 test cases achieves full code and path coverage on method `lcm` and is thus a perfect choice in the context of unit-testing. In contrast, the original, non-guided, TCG scheme with *loop-2* as termination criterion produces 9 test cases.  $\square$

A thorough experimental evaluation was performed in [39] which demonstrates the applicability and effectiveness of guided TCG.



## 4 TCG of Concurrent Programs

The focus of this section is on the development of automated techniques for testing *concurrent objects*.

### 4.1 Concurrent Objects

The central concept of the concurrency model is that of *concurrent object*. Concurrent objects live in a *distributed* environment with asynchronous and unordered communication by means of asynchronous method calls, denoted  $y ! m(\bar{z})$ . Method calls may be seen as triggers of concurrent activity, spawning new tasks (so-called *processes*) in the called object. After an asynchronous call of the form  $x = y ! m(\bar{z})$ , the caller may proceed with its execution without blocking on the call. Here  $x$  is a *future variable* which allows synchronizing with the completion of task  $m(\bar{z})$ . In particular, the instruction **await**  $x?$  allows checking whether  $m$  has finished. In this case, execution of the current task proceeds and  $x$  can be used for accessing the return value of  $m$  via the instruction  $x.\text{get}$ . Otherwise, the current task releases the processor to allow another available task to take it.

A synchronous call of the form  $x = y.m(\bar{z})$ , is internally transformed into the statement sequence  $w = y ! m(\bar{z}); \text{if } (this == y) \text{ await } w?; x = w.\text{get}$ , where  $w$  is a fresh future variable. This is because when the synchronous call is executed on the same object *this* we do not want to block this object (this would lead to a deadlock on the object *this*). Instead, we use an **await** instruction that will allow that the execution of the synchronous call to  $m$  can start to execute. The statement  $x = w.\text{get}$  blocks the execution of the current object until  $m(\bar{z})$  on  $y$  returns a value. The **if** statement avoids a deadlock when the object  $y$  is equal to *this*. For simplicity we assume that all methods return a value.

*Example 15.* Fig. 16 shows the implementation of class **A**, which contains two integer fields and five methods. Method **sumFacts** computes  $\sum_{k=ft}^{ft+(n-1)} k!$  by asynchronously invoking **fact** on object **ob**. The **await** instruction before entering the loop allows releasing the processor if **ft** is negative. Once **ft** takes a non-negative value, the task can resume its execution and enter the loop. For instance, the asynchronous call  $f = ob ! \text{fact}(3, this)$ ; in **sumFacts** will add the task **fact**(3, **this**) to the queue of **ob**. When this task starts executing, it will add the task **fact**(2, **ob**) on the object **this**, which in turn will add **fact**(1, **this**) on **ob** and so on, in such a way that the factorial is computed in a distributed way between the two objects. Note that the calls are synchronized on future variables. This means that until the recursive call **fact**(1, **this**) is not completed the other tasks are suspended on their corresponding **await** conditions.  $\square$

Let us briefly present the semantics for the concurrency instructions. An *object* is a term  $ob(o, t, h, Q)$  where  $o$  is the object identifier,  $t$  is the identifier of the *active task* that holds the object's lock or  $\perp$  if the object's lock is free,  $h$  is its local heap and  $Q$  is the set of tasks in the object. A *task* is a term  $tk(t, m, l, s)$

<pre> class A(Int n, Int ft) {   Int sumFacts(A ob) {     Fut&lt;Int&gt; f; Int res=0;     Int m = this.n;     await this.ft &gt;= 0;     while (m &gt; 0) {       f = ob ! fact(this.ft, this);       await f ?;       Int a = f.get;       res = res + a;       this.ft = this.ft + 1;       m = m - 1;     }     return res;   } } </pre>	<pre> Int fact(Int k, A ob){   Fut &lt;Int&gt; f; Int res = 1;   if (k &lt;= 0) res = 1;   else { f = ob ! fact(k - 1, this);         await f ?; res = f.get;         res = k * res;       }   return res; } Int setN(Int a) { this.n=a; return 0; } Int setFt(Int b) { this.ft=b; return 0; } Int set(Int a, Int b){   this.setN(a); this.setFt(b);   return 0; } </pre>
--	---

Fig. 16: ABS running example.

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  $\epsilon$  if the task has terminated.

A *state* or *configuration*  $S$  has the form  $o_0 \cdot o_1 \cdots o_n$ , where  $o_i \equiv \text{ob}(o_i, t_i, h_i, Q_i)$ . The execution of a program from a method  $m$  starts from an initial state  $S_o = \{\text{ob}(0, 0, \perp, \{tk(0, m, l, \text{body}(m))\})\}$ . Here,  $l$  maps parameters to their initial values (null in case of reference variables),  $\text{body}(m)$  is the sequence of instructions in method  $m$ , and  $\perp$  stands for the empty heap.

Fig. 17 presents the semantics of the concurrent objects. As objects do not share their states, the semantics can be presented as a macro-step semantics [41] (defined by means of the transition “ $\rightarrow$ ”) in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to a *release point*, i.e., a point in which the object’s processor becomes idle  $\perp$  (due to an **await** or **return** instruction). In this case, we apply rule MSTEP to select an available task from an object, namely we apply the function  $\text{selectObject}(S)$  to select non-deterministically one object in the state with a non-empty queue  $Q$  and  $\text{selectTask}(Q)$  to select non-deterministically one task of  $Q$ .

The transition  $\leadsto$  defines the evaluation within a given object. We sometimes label transitions with  $o \cdot t$ , the name of the object  $o$  and task  $t$  selected (in rule MSTEP) or evaluated in the step (in the transition  $\leadsto$ ). The notation  $h[\bar{f} \mapsto l(\bar{y})]$  (resp.  $l[x \mapsto v]$ ) stands for the result of storing  $l(\bar{y})$  in the fields  $\bar{f}$  (resp.  $v$  in  $x$ ).

The remaining sequential instructions are standard and thus omitted. In NEWOB, an active task  $t$  in object  $o$  creates an object  $o'$  of class  $D$  which is introduced to the state with a free lock. Here  $h'$  stands for a default mapping on

$$\begin{aligned}
& (\text{MSTEP}) \frac{\text{selectObject}(S) = \text{ob}(o, \perp, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, \text{selectTask}(\mathcal{Q}) = t, S \xrightarrow{o \cdot t} S'}{S \xrightarrow{o \cdot t} S'} \\
& (\text{NEWOBJ}) \frac{t = \text{tk}(t, m, l, x = \text{new } D(\bar{y}); s), \text{fresh}(o'), h' = \text{newhp}(D), l' = l[x \mapsto o'], \text{class } D(\bar{f})}{\text{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{ob}(o, t, h, \mathcal{Q} \cup \{\text{tk}(t, m, l', s)\}) \cdot \text{ob}(o', \perp, h'[f \mapsto l(\bar{y})], \{\})} \\
& (\text{ASYNC}) \frac{t = \text{tk}(t, m, l, y = x ! m_1(\bar{z}); s), l(x) = o_1, \text{fresh}(t_1), l_1 = \text{buildLocals}(\bar{z}, m_1, l)}{\text{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \cdot \text{ob}(o_1, \_, \_, \mathcal{Q}') \rightsquigarrow \text{ob}(o, t, h, \mathcal{Q} \cup \{\text{tk}(t, m, l[y \mapsto t_1], s)\}) \cdot \text{ob}(o_1, \_, \_, \mathcal{Q}' \cup \{\text{tk}(t_1, m_1, l_1, \text{body}(m_1))\})} \\
& (\text{AWAIT1}) \frac{t = \text{tk}(t, m, l, \langle \text{await } y?; s \rangle), l(y) = t_1, \text{tk}(t_1, \_, \_, s_1) \in \text{Objects}, s_1 = \epsilon(v)}{\text{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{ob}(o, t, h, \{\text{tk}(t, m, l, s)\} \cup \mathcal{Q})} \\
& (\text{AWAIT2}) \frac{t = \text{tk}(t, m, l, \langle \text{await } y?; s \rangle), l(y) = t_1, \text{tk}(t_1, \_, \_, s_1) \in \text{Objects}, s_1 \neq \epsilon(v)}{\text{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{ob}(o, \perp, h, \{\text{tk}(t, m, l, \langle \text{await } y?; s \rangle)\} \cup \mathcal{Q})} \\
& (\text{GET}) \frac{t = \text{tk}(t, m, l, \langle x = \text{get}.y; s \rangle), l(y) = t_1, \text{tk}(t_1, \_, \_, s_1) \in \text{Objects}, s_1 = \epsilon(v)}{\text{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{ob}(o, t, h, \{\text{tk}(t, m, l[x \mapsto v], s)\} \cup \mathcal{Q})} \\
& (\text{RETURN}) \frac{t = \text{tk}(t, m, l, \text{return } x; s)}{\text{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{ob}(o, \perp, h, \{\text{tk}(t, \_, \_, \epsilon(l(x)))\} \cup \mathcal{Q})}
\end{aligned}$$

Fig. 17: Summarized Semantics for Distributed and Concurrent Execution

the fields of class  $D$  initialized with the values of  $l(\bar{y})$ . ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier  $t_1$  which is associated to the corresponding future variable  $y$  in  $l$ . We have assumed that  $o \neq o_1$ , but the case  $o = o_1$  is analogous, the new task  $t_1$  is simply added to  $\mathcal{Q}$  of  $o_1$ .

The remaining rules define the concurrent execution within each distributed object. In AWAIT1, the future variable we are awaiting for points to a finished task and the **await** can be completed. The finished task  $t_1$  is looked up in all objects in the current state (denoted **Objects**). Otherwise, AWAIT2 yields the lock so that any other task of the same object can take it. GET blocks the object until the task is finished. When RETURN is executed, the return value is stored in  $v$  so that it can be obtained by the future variable that points to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction  $\epsilon(v)$ ) but it does not disappear from the state as its return value may be needed later on in an **await**. In what follows, a *derivation*  $S_0 \longrightarrow \dots \longrightarrow S_n$  from an initial state  $S_0$  of an object system is a sequence of macro-steps (applications of rule MSTEP). Since the execution is non-deterministic, multiple derivations are possible from an initial state.

*Example 16.* For instance, let us consider the following code corresponding to some method  $m$  of some class  $B$ .

(a)  $A \ x = \text{new } A(5,10);$   
 (b)  $\text{Fut}\langle \text{Int} \rangle f;$   
 (c)  $f = x ! \text{fact}(2,x);$   
 (d)  $\text{await } f?;$   
 (e)  $z = f.\text{get};$

where class  $A$  is that in Ex. 15. We start from the initial state  $S_0 = \{\text{ob}(0, 0, \perp, \{tk(0, m, l_0, (a) \cdot \dots (e))\})\}$ . By applying consecutively rules NEWOB, ASYNC and AWAIT2 to (a), (c) and (d) respectively we get:

$$S_1 = \{\text{ob}(0, 0, \perp, \{tk(0, m, l_0, (d) \cdot (e))\}), \text{ob}(1, \perp, h_1, \{tk(2, \text{fact}, l_2, \text{body}(\text{fact}))\})\}$$

where  $l_0(f) = 2$ ,  $l_2(k) = 2$ ,  $l_2(\text{ob}) = 1$  and  $h_1(n) = 5$ ,  $h_2(\text{ft}) = 10$ . We apply now a macro step on object 1, by reducing task 2. In this case the macro step stops when executing  $\text{await } f?$  of method  $\text{fact}$ , and the state is modified as follows:

$$S_2 = \{\text{ob}(0, 0, \perp, \{tk(0, m, l_0, (d) \cdot (e))\}), \text{ob}(1, 2, h_1, \{tk(2, \text{fact}, l_2, \text{await } f?; \dots), tk(3, \text{fact}, l_3, \text{body}(\text{fact}))\})\}$$

where  $l_2(f) = 3$ ,  $l_3(k) = 1$ ,  $l_3(\text{ob}) = 1$ . Similarly as done before, task with identifier 3 is now reduced, stopping the derivation when we reach  $\text{await } f?$ :

$$S_3 = \{\text{ob}(0, 0, \perp, \{tk(0, m, l_0, (d) \cdot (e))\}), \text{ob}(1, 2, h_1, \{tk(2, \text{fact}, l_2, \text{await } f?; \dots), tk(3, \text{fact}, l_3, \text{await } f?; \dots), tk(4, \text{fact}, l_4, \text{body}(\text{fact}))\})\}$$

where  $l_3(f) = 4$ ,  $l_4(k) = 0$ ,  $l_4(\text{ob}) = 1$ . Now only task 4 can be reduced and applying rule RETURN we get:

$$S_4 = \{\text{ob}(0, 0, \perp, \{tk(0, m, l_0, (d) \cdot (e))\}), \text{ob}(1, 2, h_1, \{tk(2, \text{fact}, l_2, \text{await } f?; \dots), tk(3, \text{fact}, l_3, \text{await } f?; \dots), tk(4, \perp, l_4, \epsilon(1))\})\}$$

Now, task 3 can be reduced by applying first AWAIT1 and after RETURN:

$$S_5 = \{\text{ob}(0, 0, \perp, \{tk(0, m, l_0, (d) \cdot (e))\}), \text{ob}(1, 2, h_1, \{tk(2, \text{fact}, l_2, \text{await } f?; \dots), tk(3, \perp, l_3, \epsilon(1)), tk(4, \perp, l_4, \epsilon(1))\})\}$$

Similarly we reduce task 2 and after task 0 from object 0 and we finally get:

$$S_6 = \{\text{ob}(0, 0, \perp, \{tk(0, m, l_0, \epsilon)\}), \text{ob}(1, 2, h_1, \{tk(2, \perp, l_2, \epsilon(2)), tk(3, \perp, l_3, \epsilon(1)), tk(4, \perp, l_4, \epsilon(1))\})\}$$

where  $l_0(z) = 2$ . □

Given an initial state, a naïve exploration of the search space to reach all possible system configurations does not scale. The challenge is then in avoiding the exploration of redundant states which lead to the same configuration. Partial-order reduction (POR) [16, 20] 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 [18] which maintains for each configuration a backtrack set, which is updated during the execution of the program when it realizes that a non-deterministic choice must be tried. Recently, TransDPOR [45] extends DPOR to take advantage of the transitive dependency relations in actor systems to explore fewer configurations than DPOR. As noticed in [32, 45], their effectiveness highly depend on the actor selection order.

In our semantics in Fig. 16, functions *selectObject* and *selectTask* can be implemented with novel strategies and heuristics to further prune redundant state exploration, and they can be easily integrated within the aforementioned algorithms. For instance, *selectObject* could try to find a *stable object*, i.e., an object to which no other actor will post messages. Basically, this means that the object is autonomous since its execution does not depend on any other actor and thus no backtracking is required from that point. Furthermore, when temporal stability of any object cannot be proved, it is possible to look for heuristics that assign a weight to the messages according to the error that the object-selection strategy may make when proving stability w.r.t. them. Finally, function *selectTask* can be defined to select independent tasks according to the independence notion defined in [8], which basically establishes that two tasks are independent if they access disjoint parts of the shared memory. Note that this would avoid non determinism reordering among tasks.

## 4.2 Coverage and Termination Criteria for Concurrent Objects

As commented in Sec. 2.1, an important problem in symbolic execution is that, since the input data is unknown, the execution tree to be traversed is in general infinite. Hence it is required to integrate a *termination criterion* which guarantees that the length of the paths traversed remains finite while at the same time an interesting set of test cases is generated, i.e., certain code *coverage* is achieved.

**Task-Level coverage and Termination Criteria.** Given a task executing on an object, we aim at ensuring its local termination by leveraging existing Coverage Criteria (CC for short) developed in the sequential setting to the context of concurrent objects. We focus on the *loop-k* coverage criteria [26] described in Sec. 2.1, which limits the number of times we iterate on loops to a threshold  $K_l$  (other existing criteria would pose similar problems and solutions). However applying the task-level CC to all tasks *does not guarantee termination*. This is because we can switch from one task to another an infinite number of times. For example, consider the symbolic execution of  $ob_1 ! \text{fact}(n, ob_2)$ , where method *fact* is defined in Ex. 15. We circularly switch from object  $ob_1$  to object  $ob_2$  an infinite number of times because each asynchronous call in one object adds another call on the other object (see Ex. 16). This is not detected by the task-level CC because each method invocation is a new task. Intuitively, we get the following situation, where we show in each state the value of the queues in both objects. In each step the corresponding call to *fibo* is always selected.

$$\begin{array}{ll}
\{\text{ob}_1, \text{ob}_2\}, Q_{\text{ob}_1} = \{\text{fact}(n, \text{ob}_2)\}, Q_{\text{ob}_2} = \{\} & \longrightarrow \\
\{\text{ob}_1, \text{ob}_2\}, Q_{\text{ob}_1} = \{\text{await } f?; \dots\}, Q_{\text{ob}_2} = \{\text{fact}(n_1, \text{ob}_1)\} & \xrightarrow{n_1 = n - 1} \\
\{\text{ob}_1, \text{ob}_2\}, Q_{\text{ob}_1} = \{\text{fact}(n_2, \text{ob}_2), \text{await } f?; \dots\}, Q_{\text{ob}_2} = \{\text{await } f?; \dots\} & \xrightarrow{n_2 = n_1 - 1} \\
\dots & \longrightarrow \dots
\end{array}$$

The same problem can happen even with a single object, e.g., in method `sumFacts` when executing `await (ft >= 0)`, there is an infinite branch in the evaluation tree, corresponding to the case `ft < 0` which can be re-tried forever. I.e., we can apply infinitely the rule `AWAIT2` in Fig. 17 on the task `await (ft >= 0)`, whose effect is to extract the task from the queue, to prove that the task does not hold, and to put again the task in the queue.

**Task-Switching Coverage and Termination Criteria.** In both examples above we can observe that the problem, in presence of concurrency relies, not only on loops, but also on the number of task switches allowed per object. Thus, the number of task switches can be limited by simply allowing a fixed and global number of task switching. However, it might happen that, due to excessive task switching in certain objects, others are not properly tested (i.e., their tasks exercised) because the *global* number of allowed task switches has been exceeded. For example, suppose that we add the instructions `B ob2 = new B(); ob2 ! q();` before the return in method `sumFacts`, where `B` is a class that implements method `q` but whose code is not relevant. Then, as the evaluation for the *while* loop generates an infinite number of task switches (because of the `await` instruction in the loop), the evaluation of the call `ob2 ! p();` is not reached. Thus, in order to have fairness in the process and guarantee proper coverage from the concurrency point of view, we propose to *limit the number of task switches per object* (i.e., per concurrency unit).

### 4.3 Task Interleavings in TCG

An important problem in TCG of concurrent languages is that, when a task  $t$  suspends, there could be other tasks on the same object whose execution at this point could interleave with  $t$  and modify the information stored in the heap. It is essential to consider such task interleavings in order not to lose any important path. For example, let us consider a class `C` with two fields `int n, f`, and a method `p` in `C` defined as: `int p(){n = 0; await (f > 0); if(n >= 0) return 1; else return 2;}`. Suppose a call of the form `x = o ! p(); await x?; y = x.get`. The symbolic execution of `p`, will in principle consider just one path (the one that goes through the `if` branch), giving as result always `y = 1`. There can be however another task (suspended in the queue of the object `o`) which executes when `p` suspends in `await (f > 0)` and writes a negative value on `n`. This would exercise the `else` branch when `p` resumes, giving as result `y = 2`. For example, suppose that the method `void set(){n = -1;}` belongs to class `C` and that `set()` is in the queue when executing `await (f > 0)`, and that is executed before `f > 0` holds. Then the execution of `p()` will try the `else` branch.

The questions that we solve in this section are: (a) is it possible to consider all interleavings that affect the method’s coverage? (b) do we have means to discard useless interleavings? (i.e., those which do not add new paths). As regards (a), it is not enough to assume that there is one instance of each method call in the queue as further coverage is possible by introducing multiple instances of the same method. Even though termination is guaranteed by the limit imposed on the number of task switches in Sec. 4.2 (i.e., the length of the queue is finite), it is more appropriate to define an additional coverage criteria in this new dimension by fixing the maximum length of a queue in order to achieve a more meaningful coverage.

In order to answer question (b), we start by characterizing the notion of useless interleaving. Starting from the set of all methods in the class of the method under test, we propose a sequence of *prunings* which ensure that only useless interleavings are eliminated. The objective is to over-approximate, for each method  $m$ , the set  $\text{related}(m)$ , which contains all methods whose interleaved execution with  $m$  can lead to a solution not considered before. The remaining ones are useless interleavings. Starting from the set of all methods in the class of the method under test, we propose a sequence of *prunings* which ensure that only useless interleavings are eliminated.

(*Pruning 1*) The first refinement is to discard methods which do not modify the heap, i.e., *pure* methods. Purity can be syntactically proved by checking that the method does not contain any instruction of the form  $\text{this.f} = x$  and that methods (transitively) invoked from it are pure. Using this pruning on Ex. 15, we get  $\text{related}(\text{sumFacts}) = \{\text{sumFacts}, \text{set}, \text{setN}, \text{setFt}\}$ .

(*Pruning 2*) The second pruning amounts to considering only *directly impure* methods (ignoring transitive calls), i.e., those which write directly on fields. Let  $p$  be the method under test,  $m$  be a directly impure method and  $q$  a method that invokes  $m$ . The intuition is that by considering  $m$  alone, we execute it from a more general context, while its execution from  $q$  will be just more specific (since  $q$  will have added additional constraints). Hence, it will not add additional local traces for  $p$ . With this pruning,  $\text{related}(\text{sumFacts}) = \{\text{sumFacts}, \text{setN}, \text{setFt}\}$ .

(*Pruning 3*) The third pruning consists in considering only the interleavings with those methods that write (directly) on fields which are used (read or written) before an `await`, and read after an `await`. These sets are easily computed by just looking for instructions  $\text{this.f} = x$  and  $x = \text{this.f}$  in the corresponding program fragments. Given a field  $f$ , the intuition for this condition is that, if  $f$  has not been accessed before the `await` then there is no information about the field. Thus,  $\text{related}(\text{sumFacts}) = \{\text{sumFacts}, \text{setFt}\}$ .

#### 4.4 Related Work on TCG of Thread-based Concurrency

As it happens with actor-based systems, the main difficulties in TCG of thread-based systems are related to the scalability when considering thread interleavings. In thread-based systems, this problem is exacerbated. In [37], a symbolic

execution framework which combines symbolic execution with model checking is presented to detect safety violations. Safety properties are represented by using logical formalisms understood by the model checker or that can be inserted in the code as annotations. The model checker, when doing symbolic execution, is able to report counterexamples which violate the correctness safety criterion. Furthermore, when generating test cases, the model checker generates the paths that fulfill the safety property. To reduce the number of thread interleavings, the model checker uses partial order reduction techniques [20] as we do. An advantage on this technique is the possibility of handling native calls through mixed concrete-symbolic solving. The main drawback of this framework is that satisfiability of constraints is checked at the end of each branch of the symbolic tree, what it might be unfeasible. Thus, they use preconditions on the symbolic input values in order to avoid the exploration of branches which violate the precondition. In contrast to [37], our CLP-approach is able to discard a branch in the symbolic tree once the associated constraint are unsatisfiable.

Other approaches that use techniques different from ours are [29, 43, 44]. The work [29] combines dynamic symbolic execution (concolic testing) with unfoldings. The unfolding approach allows constructing a compact representation of the interleavings and thus the new testing algorithm may use this information to guide the symbolic execution, avoiding irrelevant interleavings. This new approach achieves in some cases an exponential gain when compared with existing dynamic partial-order reduction based approaches [18, 45]. Basically, the point is that in the previous approaches, the number of explored interleavings depends on the order in which processes are executed, but in this new approach it does not, since interleavings are computed a priori.

In [43, 44], a runtime algorithm to monitor executions for multithreaded Java and possibly detect safety violations is presented. From a concrete execution, they automatically extract a partial order causality from a sequence of read/write events on shared variables. Basically they extract, for a shared variable, the sequence of write/reads/write to that variable in the execution. Thus any permutation of these events can be considered an execution of the program if and only if it does not contradict the partial order. The main drawbacks is the state explosion since a large number of unreachable branches may be explored.

As an improvement of the previous work, in [42], a novel approach uses concolic execution (a combination of symbolic and concrete execution) to test shared-memory in multithreaded programs by using an algorithm based on race-detection and flipping. From a concrete execution, they determine the partial order relation or the exact race conditions between the processes in the execution path. Afterwards, such processes involved in races are flipped by generating new thread schedules and generating new test inputs. Hence, differently to the previous conservative approaches, in this work they explore one path from each partial order, avoiding possible warnings that could never occur in a real execution.



## 5 Conclusions

This tutorial summarizes the basic principles used in TCG by symbolic execution. It first discusses the main challenges that TCG currently poses: the efficient handling of heap-manipulating programs, compositionality, and guiding the process. It then overviews a particular instantiation of the generic TCG framework that uses CLP as enabling technology. We will review the main features, advantages and implementation of this CLP-approach. Finally, we discuss the extension of the basic framework to handle concurrent actor systems.

*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 projects TIN2008-05624 and TIN2012-38137, and by the CM project S2013/ICE-3006.

## References

1. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
3. E. Albert, I. Cabañas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutiérrez. jPET: an Automatic Test-Case Generator for Java. In *WCRE'11*, pages 441–442. IEEE Computer Society, 2011.
4. Elvira Albert, María García de la Banda, Miguel Gómez-Zamalloa, José Miguel Rojas, and Peter Stuckey. A CLP Heap Solver for Test Case Generation. *Theory and Practice of Logic Programming*, 13(4-5):721–735, July 2013.
5. Elvira Albert, Miguel Gómez-Zamalloa, José Miguel Rojas, and Germán Puebla. Compositional CLP-based Test Data Generation for Imperative Languages. In María Alpuente, editor, *LOPSTR 2010 Revised Selected Papers*, volume 6564 of *Lecture Notes in Computer Science*, pages 99–116. Springer-Verlag, 2011.
6. S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
7. S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS'08*, LNCS 4963. Springer, 2008.
8. G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
9. C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.
10. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE'11*, pages 1066–1071. ACM, 2011.
11. L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

12. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
13. F. Degraeve, T. Schrijvers, and W. Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *LOPSTR’09*, LNCS 6037, pages 128–142. Springer, 2010.
14. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In *Logic Programming*, LNCS 3668, pages 67–82. Springer, 2005.
15. C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In *TAP’07*, LNCS 4454, pages 169–188. Springer, 2007.
16. Javier Esparza. Model checking using net unfoldings. *Sci. Comput. Program.*, 23(2-3):151–195, 1994.
17. R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
18. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.
19. P. Godefroid. Compositional Dynamic Test Generation. In *POPL’07*, pages 47–54. ACM, 2007.
20. Patrice Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1991.
21. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.
22. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP’10 Special Issue*, 10(4–6):659–674, 2010.
23. A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *CL’00*, LNAI 1861, pages 399–413. Springer, 2000.
24. N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *ASE’00*, pages 219–228. IEEE Computer Society, 2000.
25. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
26. W.E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.
27. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
28. The Java Modelling Language homepage. <http://www.eecs.ucf.edu/~leavens/JML/index.shtml>, 2013.
29. Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Using unfoldings in automated testing of multithreaded programs. In Michael Goedicke, Tim Menzies, and Moto-shi Saeki, editors, *ASE*, pages 150–159. ACM, 2012.
30. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS’03*, LNCS 2619, pages 553–568. Springer, 2003.
31. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.
32. Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In *FASE*

- 2010, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.
33. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.
  34. K. Marriott and P. J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
  35. C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
  36. R. A. Müller, C. Lembeck, and H. Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *IASTEDSE’04*, pages 365–371. ACTA Press, 2004.
  37. Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehrlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.
  38. C. S. Păsăreanu and W. Visser. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, 2009.
  39. José Miguel Rojas and Miguel Gómez-Zamalloa. A Framework for Guided Test Case Generation in Constraint Logic Programming. In *LOPSTR 2012*, volume 7844 of *Lecture Notes in Computer Science*, pages 176–193. Springer, 2013.
  40. José Miguel Rojas and Corina S. Păsăreanu. Compositional Symbolic Execution through Program Specialization. In *BYTECODE 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, March 2013.
  41. Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.
  42. Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.
  43. Koushik Sen, Grigore Rosu, and Gul Agha. Online efficient predictive safety analysis of multithreaded programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2004.
  44. Koushik Sen, Grigore Rosu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005*, volume 3535 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2005.
  45. 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 *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.
  46. N. Tillmann and J. de Halleux. Pex: White Box Test Generation for .NET. In *TAP’08*, LNCS 4966, pages 134–153. Springer, 2008.
  47. Markus Triska. The Finite Domain Constraint Solver of SWI-Prolog. In *FLOPS*, LNCS 7294, pages 307–316, 2012.
  48. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.
  49. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

## Appendix B

Article “*Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing*”, [3]

# 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

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.

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

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  $A_1, \dots, A_n$ . 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  $\epsilon$  if the task has terminated. A *state* or

$$\begin{array}{l}
\text{(MSTEP)} \quad \frac{\text{selectActor}(S) = \text{act}(a, \perp, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, \text{selectTask}(a) = t, S \xrightarrow{a \cdot t} S'}{S \xrightarrow{a \cdot t} S'} \\
\text{(SETFIELD)} \quad \frac{t = \text{tsk}(t, m, l, \text{this}.f = y; s)}{\text{act}(a, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{act}(a, t, h[f \mapsto l(y)], \mathcal{Q} \cup \{\text{tsk}(t, m, l, s)\})} \\
\text{(GETFIELD)} \quad \frac{t = \text{tsk}(t, m, l, x = \text{this}.f; s)}{\text{act}(a, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{act}(a, t, h, \mathcal{Q} \cup \{\text{tsk}(t, m, l[x \mapsto h(f)], s)\})} \\
\text{(NEWACTOR)} \quad \frac{t = \text{tsk}(t, m, l, x = \text{new } D; s), \text{fresh}(a'), h' = \text{newheap}(D), l' = l[x \rightarrow a']}{\text{act}(a, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{act}(a, t, h, \mathcal{Q} \cup \{\text{tsk}(t, m, l', s)\}) \cdot \text{act}(a', \perp, h', \{\})} \\
\text{(ASYNC)} \quad \frac{t = \text{tsk}(t, m, l, x ! m_1(\bar{z}); s), l(x) = a_1, \text{fresh}(t_1), l_1 = \text{buildLocals}(\bar{z}, m_1, l)}{\text{act}(a, t, h, \mathcal{Q} \cup \{t\}) \cdot \text{act}(a_1, \rightarrow, \rightarrow, \mathcal{Q}') \rightsquigarrow \text{act}(a, t, h, \mathcal{Q} \cup \{\text{tsk}(t, m, l, s)\}) \cdot \text{act}(a_1, \rightarrow, \rightarrow, \mathcal{Q}' \cup \{\text{tsk}(t_1, m_1, l_1, \text{body}(m_1))\})} \\
\text{(RETURN)} \quad \frac{t = \text{tsk}(t, m, l, \text{return}; s)}{\text{act}(a, t, h, \mathcal{Q} \cup \{t\}) \rightsquigarrow \text{act}(a, \perp, h, \mathcal{Q})}
\end{array}$$

**Fig. 1.** Summarized Semantics for Distributed and Concurrent Execution

configuration  $S$  has the form  $a_0 \cdot a_1 \cdots a_n$ , where  $a_i \equiv \text{act}(a_i, t_i, h_i, \mathcal{Q}_i)$ . The execution of a program from a method  $m$  starts from an initial state  $S_0 = \{\text{act}(0, 0, \perp, \{\text{tsk}(0, m, l, \text{body}(m))\})\}$ . Here,  $l$  maps parameters to their initial values (null in case of reference variables),  $\text{body}(m)$  is the sequence of instructions in method  $m$ , and  $\perp$  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  $\text{selectActor}(S)$  to select non-deterministically one *active* actor in the state (i.e., an actor with a non-empty queue) and  $\text{selectTask}(a)$  to select non-deterministically one task of  $a$ ’s queue. The transition  $\rightsquigarrow$  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  $\rightsquigarrow$ ). 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  $h' = \text{newheap}(D)$  stands for a default initialization on the fields of class  $D$ . ASYNC spawns a new task (the initial state is created by  $\text{buildLocals}$ ) 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  $\text{act}(a, \perp, h, \{\})$ . 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

```

1: procedure Explore( $E$ )
2:    $S = last(E)$ ;
3:   updateBackSets( $E, S$ );
4:    $a = selectActor(S)$ ;
5:   if  $a \neq \epsilon$  then
6:      $back(S) = \{a\}$ ;
7:      $done(S) = \emptyset$ ;
8:     while  $\exists(a \in back(S) \setminus done(S))$  do
9:        $done(S) = done(S) \cup \{a\}$ ;
10:      for all  $t \in selectTask(a)$  do
11:        Explore( $E \cdot next(S, a \cdot t)$ );

```

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

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  $E = S_0 \xrightarrow{a_1 \cdot t_1} \dots \xrightarrow{a_n \cdot t_n} S_n$  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

<pre> { /* main Block */   Reg rg = new Reg;   Worker<sub>1</sub> wk1 = new Worker<sub>1</sub>();   Worker<sub>2</sub> wk2 = new Worker<sub>2</sub>();   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;} } </pre>	<pre> class Worker<sub>1</sub> {   void q(Reg rg) {     rg ! m(); // m     return;   } }  class Worker<sub>2</sub> {   void h(Reg rg) {     rg ! t(); // t     return;   } } </pre>
--	---

**Fig. 3.** Running Example

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 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  $\epsilon$  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<sub>1</sub>* and *Worker<sub>2</sub>*, together with a *main* block from which the execution starts. In Fig. 4 we show the search tree built by executing  $Explore(E_0)$ , where  $E_0 = S_{ini} \xrightarrow{main} S_0$ , 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



#### 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, 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, Q) ∈ 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  $A_1::m_1 \rightarrow A_2::m_2 \rightarrow \dots \rightarrow A_k::m_k$ , 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  $A_1::m_1 \rightarrow \dots \rightarrow A_n::m_n \in Ch(A_1::m_1)$ ; or*
2. *For all chains  $A_1::m_1 \rightarrow \dots \rightarrow A_n::m_n \in Ch(A_1::m_1)$ ,  $l(x) \neq a$  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  $\mathcal{Q}_3 = \{tsk(t_3, m, l_3, \{y!p(); \text{return}; \})\}$ ,  $l_3(y) = a_2$ ,  $body(B :: p) = \{x = \text{this}.f; x!q(); \text{return}; \}$ , 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** 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  $a \in \text{class}(A_n)$  w.r.t. a  $\text{task}(t, m_1, l, \_)$  of an actor  $\text{act}(a', \_, h', \_) \in S$  breaking its stability ( $a \neq a'$ ,  $\text{class}(a') = A_1$ ) 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  $x \in \text{dom}(l)$  or  $h'(f)=a$ , for some  $f \in \text{Fields}(A_1)$  and  $n=2$ , then  $st(a, t, Ch, S)=k_a$ .
- (b) If  $l(x)=a$ , for some  $x \in \text{dom}(l)$  or  $h'(f)=a$ , for some  $f \in \text{Fields}(A_1)$  and  $n > 2$ , then  $st(a, t, Ch, S)=k_b$ .
- (c) Otherwise, i.e.,  $l(x) \neq a$ , for all  $x \in \text{dom}(l)$  and  $h'(f) \neq a$ , for all  $f \in \text{Fields}(A_1)$ , then  $st(a, t, Ch, S)=k_c$ .

The *stability level* of an actor  $a \in S$ ,  $\text{class}(a)=A_n$ , w.r.t. a task  $\text{task}(t, m_1, l, \_)$  from  $\text{act}(a', \_, h', \_) \in S$ ,  $\text{class}(a')=A_1$ , 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  $t \in \mathcal{Q}'$ ,  $\text{act}(a', t, h', \mathcal{Q}') \in S$ ,  $a \neq a'$ .

Given a state  $S = a_1 \cdot \dots \cdot a_n$ , 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  $n$ th element in the Fibonacci sequence in a distributed fashion. The computation starts with the execution of a task  $\text{fib}(3)$  on actor  $a_1$ , which in turn generates two actors  $a_2$  and  $a_3$  with  $Q_{a_2} = \{\text{fib}(2)\}$  and  $Q_{a_3} = \{\text{fib}(1)\}$ .

```

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

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 `fib(2)`. This generates two more actors  $a_4$  and  $a_5$  with  $Q_{a_4} = \{\text{fib}(1)\}$  and  $Q_{a_5} = \{\text{fib}(0)\}$ . 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,  $Q_{a_1} = \{\text{res}(1)\}$ , and  $Q_{a_2} = \{\text{res}(1), \text{res}(0)\}$ . 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` on it. However,  $a_2$  is stable, but we cannot determine it syntactically since there is a call chain  $\text{Fib}::\text{res} \rightarrow \text{Fib}::\text{res} \rightarrow \text{Fib}::\text{res}$  (i.e. we can reach from  $\text{Fib}::\text{res}$  to  $\text{Fib}::\text{res}$  through  $\text{Fib}::\text{res}$ ), 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  $\text{indep}(t_1, t_2)$ , if for any complete execution  $S_0 \longrightarrow \dots \longrightarrow S_n$  with  $t_1 < t_2$ , there exists another execution  $S_0 \longrightarrow \dots \longrightarrow S_n$  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  $\text{act}(a, -, -, \mathcal{Q}) \in S$  and  $\text{tsk}(t, m, -, s) \in \mathcal{Q}$ . We define the set  $W(t)$  as  $\{f \mid \text{this}.f = y \in s\}$ . Similarly, the set  $R(t)$  is defined as  $\{f \mid x = \text{this}.f \in s\}$ . The following theorem is an immediate consequence of the definition of independent task above. We denote by  $\text{indep}(t_1, t_2)$  that  $t_1$  and  $t_2$  are independent.

**Theorem 2 (sufficient condition for tasks independence).** *Given a state  $S$ , an actor  $\text{act}(a, -, -, \mathcal{Q}) \in S$  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  $\text{indep}(t_1, t_2)$  holds.*



9:	<b>for all</b> $t \in \text{selectTask}(a)$ <b>do</b>
10:	$\text{unmark}(a); \text{mark}(t, a);$
11:	$\text{Explore}(E \cdot \text{next}(S, a \cdot t))$

**Fig. 6.** Refining Algorithm 2 with Task Selection

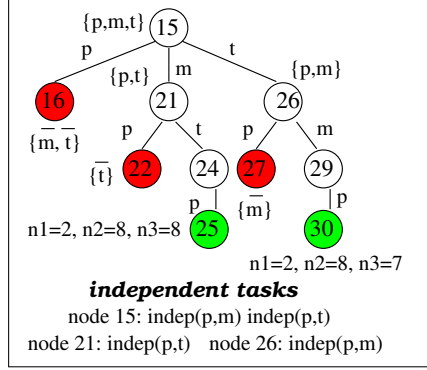
Note that since the actor state is local, i.e., fields cannot be accessed from other actors. Thus, all accesses to the heap are on the actor `this`.

### 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  $\langle t, \text{flag} \rangle$ , where  $t$  is a task and  $\text{mark}$  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  $\text{selectTask}(a)$  returns the list of unmarked tasks in the queue  $Q$  of  $a$ , i.e, those tasks of the form  $\langle t, \text{false} \rangle$ ; (2) procedure  $\text{unmark}(a)$  traverses  $Q$  and changes the flag  $\text{mark}$  to  $\text{false}$ ; and (3) procedure  $\text{mark}(t, a)$  sets the flag  $\text{mark}$  to  $\text{true}$  for all tasks which are independent with  $t$  and occur in  $Q$  after  $t$ .

Intuitively the task selection process works as follows. Given  $\text{act}(a, -, -, Q) \in S$ ,  $Q$  contains a list  $[t_1, \dots, t_n]$  of tasks. These tasks are selected one by one traversing  $Q$  (line 10 of Alg. 2). This means that if  $t_i$  is selected by  $\text{selectTask}(a)$  and  $t_i$  is independent from  $t_j$ , then  $i < j$ , i.e., the task  $t_i$  is selected before  $t_j$ . Furthermore, procedure  $\text{mark}(t_i, a)$  puts the flag  $\text{mark}$  of  $t_j$  to  $\text{true}$ . 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  $\text{mark}$  of  $t$  is  $\text{true}$ . At this point, all tasks in  $\text{rg}$  have the flag  $\text{mark}$  set to  $\text{false}$ . Thus  $\text{selectTask}(\text{rg})$  returns the list  $[p, m, t]$ . Procedure  $\text{unmark}$  does nothing. The execution of  $\text{mark}(p, \text{rg})$  then sets the flag  $\text{mark}$  of  $m$  and  $t$  to  $\text{true}$  since  $\text{indep}(p, m)$  and  $\text{indep}(p, t)$ . This branch is therefore cut at node 16 ( $\text{selectTask}(\text{rg})$  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  $\text{mark}(p, \text{rg})$  sets the flag of  $t$  to  $\text{true}$  since we have the independence relation  $\text{indep}(p, t)$ . Hence at node 22 the branch is cut ( $\text{selectActor}(\text{rg})$  returns the empty list). Similarly, at node 27 the branch is cut



**Fig. 7.** Pruning due to task-selection

because of  $\text{indep}(p, m)$ . 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.

## 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  $\pi$  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

Test	No task sel. reduction				With task. sel. reduct.				Speedup	
	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

**Table 1.** Experimental evaluation

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.

A relevant point to note, which is not shown in the table, is that no back-tracking 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.

**Acknowledgments.** 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. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
2. E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In *Proc. of ESEC/FSE'13*, pp. 595–598. ACM, 2013.
3. G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
4. L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.
5. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07, LNCS 4421*, pp. 316–330. Springer, 2007.
6. J. Esparza. Model checking using net unfoldings. *Sci. Comput. Program.*, 23(2-3):151–195, 1994.
7. C. Flanagan, Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proc. of POPL'05*, pp. 110–121. ACM, 2005.
8. P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proc. of CAV'91, LNCS 531*, pp. 176–185. Springer, 1991.
9. P. Haller, M. Odersky. Scala actors: Unifying Thread-Based and Event-Based Programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
10. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers), LNCS 6957*, pp. 142–164. Springer, 2012.
11. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.
12. S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In *Proc. of FASE'10, LNCS 6013*, pp. 308–322. Springer, 2010.
13. K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *Proc. of FASE'06, LNCS 3922*, pp. 339–356. Springer, 2006.
14. Setak: A Framework for Stepwise Deterministic Testing of Akka Actors. <http://mir.cs.illinois.edu/setak>.
15. B. Steensgaard. Points-to Analysis in almost Linear Time. In *Proc. of POPL'91*, pp. 32–41, ACM Press, 1996.
16. S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *FMOODS/FORTE'12, LNCS 7273*, pp. 219–234. Springer, 2012.

## Appendix C

Article “*Test Case Generation of Actor Systems*”, [4]

# 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 priori, 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:

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

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  $A_1, \dots, A_n$ . 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



$$\begin{array}{l}
(\text{mstep}) \frac{\text{select}A(S) = \text{ac}(\mathbf{r}, \perp, h, Q), Q \neq \emptyset, \text{select}T(\mathbf{r}) = t, S \xrightarrow{\mathbf{r} \cdot t} S'}{S \xrightarrow{\mathbf{r} \cdot t} S'} \\
(\text{setf}) \frac{t = \text{tk}(t, m, l, \text{this}.f = y; s)}{\text{ac}(\mathbf{r}, t, h, Q \cup \{t\}) \rightsquigarrow \text{ac}(\mathbf{r}, t, h[f \mapsto l(y)], Q \cup \{\text{tk}(t, m, l, s)\})} \\
(\text{getf}) \frac{t = \text{tk}(t, m, l, x = \text{this}.f; s)}{\text{ac}(\mathbf{r}, t, h, Q \cup \{t\}) \rightsquigarrow \text{ac}(\mathbf{r}, t, h, Q \cup \{\text{tk}(t, m, l[x \mapsto h(f)], s)\})} \\
(\text{new}) \frac{t = \text{tk}(t, m, l, x = \text{new } D; s), n = \text{fresh}(), h' = \text{newheap}(D), l' = l[x \rightarrow \mathbf{r}_n^D]}{\text{ac}(\mathbf{r}, t, h, Q \cup \{t\}) \rightsquigarrow \text{ac}(\mathbf{r}, t, h, Q \cup \{\text{tk}(t, m, l', s)\}) \cdot \text{ac}(\mathbf{r}_n^D, \perp, h', \{\})} \\
(\text{asy}) \frac{t = \text{tk}(t, m, l, x ! m_1(\bar{z}); s), l(x) = \mathbf{r}_1, t_1 = \text{fresh}(), \mathbf{r} \neq \mathbf{r}_1, l_1 = \text{newlocals}(\bar{z}, m_1, l)}{\text{ac}(\mathbf{r}, t, h, Q \cup \{t\}) \cdot \text{ac}(\mathbf{r}_1, t', h', Q') \rightsquigarrow \text{ac}(\mathbf{r}, t, h, Q \cup \{\text{tk}(t, m, l, s)\}) \cdot \text{ac}(\mathbf{r}_1, t', h', Q' \cup \{\text{tk}(t_1, m_1, l_1, \text{bd}(m_1))\})} \\
(\text{return}) \frac{t = \text{tk}(t, m, l, \epsilon)}{\text{ac}(\mathbf{r}, t, h, Q \cup \{t\}) \rightsquigarrow \text{ac}(\mathbf{r}, \perp, h, Q)}
\end{array}$$

**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  $\text{ac}(\mathbf{r}_n^C, t, h, Q)$  where  $\mathbf{r}$  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  $\perp$  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  $\mathbb{V} = \mathbb{Z} \cup \text{Ref} \cup \{\text{null}\}$  and  $\text{Ref}$  stands for the set of references of the form  $\mathbf{r}_n^C$ . Whenever it is clear from the context, we will omit  $n$  and  $C$  from actor identifiers by using only  $\mathbf{r}$ . A *task* is a term  $\text{tk}(t, m, l, s)$  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  $\epsilon$  to denote an empty sequence of instructions. A *state*  $S$  has the form  $\mathbf{r}_0 \cdot \mathbf{r}_1 \cdot \dots \cdot \mathbf{r}_n$ , where  $\mathbf{r}_i$  is used to refer to the whole actor  $\text{ac}(\mathbf{r}_i^{C_i}, t_i, h_i, Q_i)$  and  $\mathbf{r}_i \neq \mathbf{r}_j$ ,  $1 \leq i, j \leq n$ ,  $i \neq j$ .

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  $\text{select}A(S)$  to select non-deterministically one *active* actor in the state (i.e., an actor with a non-empty queue) and  $\text{select}T(\mathbf{r})$  to select non-deterministically one task of  $\mathbf{r}$ 's queue. The transition  $\rightsquigarrow$  defines the evaluation within a given actor. We sometimes label transitions with  $\mathbf{r} \cdot t$ , the name of the actor  $\mathbf{r}$  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

<pre> <b>void</b> ft(<b>int</b> n) {   <b>if</b> (n &gt; this.mx) {     this ! wk(n,this.mx);     this ! dg(n-this.mx);   } <b>else</b> {     this ! wk(n,n);     this ! rp(1);   } } </pre>	<pre> <b>void</b> wk(<b>int</b> n,<b>int</b> h) {   <b>while</b> (h &gt; 0){     this.r = this.r * n;     n = n - 1;     h = h - 1;   } } </pre>	<pre> <b>void</b> dg(<b>int</b> n) {   Fact wkr = <b>new</b> Fact(this,this.mx);   wkr ! ft(n); } <b>void</b> rp(<b>int</b> x) {   this.r = this.r * x;   <b>if</b> (this.b != null) this.b ! rp(this.r); } </pre>
--	--	--

**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  $\mathbf{r}$  creates an actor  $\mathbf{r}_n^D$  of class  $D$  with a fresh identifier  $n = \text{fresh}()$ , which is introduced to the state with a free lock. Here  $h' = \text{newheap}(D)$  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  $\mathbf{r} \neq \mathbf{r}_1$ , but the case  $\mathbf{r} = \mathbf{r}_1$  is analogous, the new task  $t_1$  is added to  $Q$  of  $\mathbf{r}$ . In what follows, a *derivation*  $E$  is a sequence  $S_0 \rightarrow \dots \rightarrow S_k$  of macro-steps (applications of (**mstep**)) starting from an initial state  $\text{ac}(\mathbf{r}_0^C, \perp, h, \{tk(0, m, l, bd(m))\})$ , 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  $\text{ac}(\mathbf{r}, \perp, h, \{\})$ . 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  $\mathbf{a} ! \text{ft}(n)$ ). Actor **a** computes  $n * (n-1) * \dots * (n - \mathbf{a}.mx + 1)$  by means of task  $\text{wk}(n, \mathbf{a}.mx)$ , and *delegates* to another actor the rest of the computation, by means of task  $\text{dg}(n - \mathbf{a}.mx)$ . When an actor is asked to compute the factorial of an  $n$ , which is smaller than its  $mx$ , then the call  $\text{this} ! \text{wk}(n, n)$  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  $S_0 = \text{ac}(\mathbf{r}_0, \perp, h_0, \{tk(0, \text{ft}, l_0, bd(\text{ft}))\})$ , where  $h_0(r) = 1$ ,  $h_0(b) = \text{null}$ ,  $h_0(mx) = 2$  and  $l_0(n) = 5$ , i.e., we want to compute factorial of 5 with  $\text{this}.mx$  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  $\mathbf{r}_i$  and the local variables of task  $t_i$  respectively.

- (a)  $S_0 \xrightarrow{0} \text{ac}(\mathbf{r}_0, 0, h_0, \{tk(1, \text{wk}, [n \mapsto 5, h \mapsto 2], bd(\text{wk})), tk(2, \text{dg}, [n \mapsto 3], bd(\text{dg}))\}) \xrightarrow{(1,2)^*}$   
(b)  $\text{ac}(\mathbf{r}_0, \perp, [r \mapsto 5 * 4], \{\}) \cdot \text{ac}(\mathbf{r}_1, \perp, [r \mapsto 1, b \mapsto \mathbf{r}_0], \{tk(3, \text{ft}, [n \mapsto 3], bd(\text{ft}))\}) \xrightarrow{(3)^*}$

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

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

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

In the last step we have computed  $h_0(\mathbf{r})=5*4$ , 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  $\mathbf{r}$  in  $\mathbf{r}_0$ .  $\square$

### 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  $\{\langle X<0, Y=-X \rangle, \langle X \geq 0, Y=X \rangle\}$  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  $X<0$  on the input and  $Y=-X$  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  $X, Y, F, G, \dots \in \mathcal{V}$ , an *atomic constraint*  $\varphi$  is of the form:

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

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

$$\begin{array}{l}
(\text{mstep})_{\Phi} \frac{\text{select}A(\mathcal{S}) = \text{ac}(\text{ref}, \perp, \neg, \mathcal{Q}, \neg), \mathcal{Q} \neq \emptyset, \text{select}T(\text{ref}) = t, \mathcal{S} \sqcap \mathcal{I} \xrightarrow{\text{ref} \cdot t}_{\Phi} \mathcal{S}' \sqcap \mathcal{I}'}{\mathcal{S} \sqcap \mathcal{I} \xrightarrow{\text{ref} \cdot t}_{\Phi} \mathcal{S}' \sqcap \mathcal{I}'} \\
(\text{setf})_{\Phi} \frac{t = \text{tk}(t, m, \rho, \text{this}.f = y; s), \theta' = \theta[f \mapsto F], \varphi = \{F \doteq \rho(y)\}}{\text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t\}, \Phi) \rightsquigarrow_{\Phi} \text{ac}(\text{ref}, t, \theta', \mathcal{Q} \cup \{t'\}, \Phi \cup \varphi)} \\
(\text{getf})_{\Phi} \frac{t = \text{tk}(t, m, \rho, x = \text{this}.f; s), \rho_1 = \rho[x \mapsto X], \varphi = \{X \doteq \theta(f)\}}{\text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t\}, \Phi) \rightsquigarrow_{\Phi} \text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{\text{tk}(t, m, \rho_1, s)\}, \Phi \cup \varphi)} \\
(\text{new})_{\Phi} \frac{t = \text{tk}(t, m, \rho, x = \text{new } D; s), n = \text{fresh}(), \rho_1 = \rho[x \mapsto X], \Phi_1 = \text{init}(D), \varphi = \{X \doteq \mathbf{r}_n^D\}}{\text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t\}, \Phi) \rightsquigarrow_{\Phi} \text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{\text{tk}(t, m, \rho_1, s)\}, \Phi \cup \varphi) \cdot \text{ac}(\mathbf{r}_n^D, \perp, \theta_s, \{\}, \Phi_1)} \\
(\text{asy})_{\Phi_1} \frac{\begin{array}{l} t = \text{tk}(t, m, \rho, x ! m_1(\bar{z}); s), \rho(x) \text{ is object-bounded in } \Phi \\ \Phi \models \rho(x) \doteq \text{ref}', t_1 = \text{fresh}(), \text{fresh}(m_1(\bar{w})\{s_1; \}), \Phi' = \Pi_{\rho(\bar{z})} \Phi \cup \{\rho_s(\bar{w}) \doteq \rho(\bar{z})\} \end{array}}{\text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t\}, \Phi) \cdot \text{ac}(\text{ref}', \neg, \theta', \mathcal{Q}_1, \Phi_1) \rightsquigarrow_{\Phi} \text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t'\}, \Phi) \cdot \text{ac}(\text{ref}', \neg, \theta', \mathcal{Q}_1 \cup \{\text{tk}(t_1, m_1, \rho_s, s_1)\}, \Phi_1 \cup \Phi')} \\
(\text{asy})_{\Phi_2} \frac{\begin{array}{l} t = \text{tk}(t, m, \rho, x ! m_1(\bar{z}); s), \text{class}(x) = D, \rho(x) \text{ is not object-bounded in } \Phi \\ t_1 = \text{fresh}(), \text{fresh}(m_1(\bar{w})\{s_1; \}), \Phi' = \Pi_{\rho(\bar{z})} \Phi \cup \{\rho_s(\bar{w}) \doteq \rho(\bar{z})\} \end{array}}{\text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t\}, \Phi) \cdot \text{ac}(\mathbf{s}_n^D, \neg, \theta', \mathcal{Q}_1, \Phi_1) \rightsquigarrow_{\Phi} \text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t'\}, \Phi \cup \{\rho(x) \doteq \mathbf{s}_n^D\}) \cdot \text{ac}(\mathbf{s}_n^D, \neg, \theta', \mathcal{Q}_1 \cup \{\text{tk}(t_1, m_1, \rho_s, s_1)\}, \Phi_1 \cup \Phi')} \\
(\text{asy})_{\Phi_3} \frac{\begin{array}{l} t = \text{tk}(t, m, \rho, x ! m_1(\bar{z}); s), \text{class}(x) = D, \rho(x) \text{ is not object-bounded in } \Phi, n = \text{fresh}() \\ t_1 = \text{fresh}(), \text{fresh}(m_1(\bar{w})\{s_1; \}), \Phi' = \{\theta_s(\text{this}) \doteq \mathbf{s}_n^D\} \cup \Pi_{\rho(\bar{z})} \Phi \cup \{\rho_s(\bar{w}) \doteq \rho(\bar{z})\} \end{array}}{\text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t\}, \Phi) \sqcap \mathcal{I} \rightsquigarrow_{\Phi} \text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t'\}, \Phi \cup \{\rho(x) \doteq \mathbf{s}_n^D\}) \cdot \text{ac}(\mathbf{s}_n^D, \perp, \theta_s, \{\text{tk}(t_1, m_1, \rho_s, s_1)\}, \Phi') \sqcap \mathcal{I} \cup \{\langle \mathbf{s}_n^D, \theta_s \rangle\}} \\
(\text{return})_{\Phi} \frac{t = \text{tk}(t, m, \rho, \epsilon)}{\text{ac}(\text{ref}, t, \theta, \mathcal{Q} \cup \{t\}, \Phi) \rightsquigarrow_{\Phi} \text{ac}(\text{ref}, \perp, \theta, \mathcal{Q}, \Phi)}
\end{array}$$

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

instruction. References of the form  $\mathbf{s}_n^C$  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  $\Phi$  a conjunction of atomic constraints. We use simply  $\mathbf{r}$  (resp.  $\mathbf{s}$ ) instead of  $\mathbf{r}_n^C$  (resp.  $\mathbf{s}_n^C$ ) when the values of  $C$  and  $n$  are irrelevant. We use  $\text{ref}$  to refer either to  $\mathbf{r}$  or  $\mathbf{s}$  and sometimes set notations to refer to  $\Phi$ .

Fig. 3 presents the operational semantics of symbolic execution for the concurrent instructions (the sequential ones are standard). A *symbolic state* has the form  $\mathcal{S} \sqcap \mathcal{I}$ , 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  $(\text{asy})_{\Phi_3}$ , since it is the only rule that modifies  $\mathcal{I}$ . A *symbolic actor* is represented as  $\text{ac}(\text{ref}, t, \theta, \mathcal{Q}, \Phi)$ , where  $\text{ref} \in \text{Ref}^*$  is the actor identifier,  $t$  is the identifier of the active task,  $\mathcal{Q}$  is the queue of symbolic pending tasks,  $\Phi$  is a set of constraints involving the fields of the actor and the variables of its tasks, and  $\theta : \mathcal{F}(C) \cup \{\text{this}\} \mapsto \mathcal{V}$  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  $\Phi$ . A *symbolic task*  $\text{tk}(t, m, \rho, s)$  of a method  $m$  in a class  $C$  contains the sequence of instructions  $s$

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

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  $\rho$  variable  $x$  to the fresh variable  $X$ . Rule **(setf)** $_\Phi$  updates the field mapping  $\theta$  with the fresh variable  $F$ , and stores the new constraint  $F = \rho(y)$  in  $\Phi$ . Since a field is modified, the mapping  $\theta$  in the actor must be updated. However, in rule **(getf)** $_\Phi$ , the field  $f$  is read and thus, it is not required to update  $\theta$  but  $\rho$ . Rule **(new)** $_\Phi$  adds the constraint  $X = \mathbf{r}_n^D$  to  $\Phi$  and updates  $\rho$ . The function  $\Phi_1 = \text{init}(D)$  initializes  $\Phi_1$  with the corresponding initialization of the fields in  $D$  and the **this** actor, i.e.,  $\theta_s(\text{this}) = \mathbf{r}_n^D \in \Phi_1$  and if a field  $f$  in  $D$  is initialized to a value  $v$ , then  $\theta_s(f) = v$  will be in  $\Phi_1$ . Rule **(return)** $_\Phi$  allows us to apply rule **(mstep)** $_\Phi$ . A main aspect is the treatment of asynchronous calls  $x ! m_1(\bar{z})$ , which distinguishes three cases:

1. **(asy)** $_{\Phi_1}$ : *Object  $x$  exists in the store.* This condition is checked by seeing if  $x$  is bounded in  $\Phi$ . Formally we say that  $x$  is *object-bounded* in  $\Phi$  if  $\rho(x) \in \text{vars}(\Phi)$  and  $\Phi \models \rho(x) = \text{ref}'$ , for some actor  $\text{ref}'$ . In this case, the task  $m_1$  is introduced in the queue of actor  $\text{ref}'$ . Here  $\text{fresh}(m_1(\bar{w})\{s_1; \})$  is a fresh renaming of the variables in  $m_1$ . We use  $\Pi_{\rho(\bar{z})}\Phi$  to denote the projection of  $\Phi$  on the variables  $\rho(\bar{z})$ , i.e., the constraints in  $\Phi$  involving the input parameters  $\bar{z}$ . Note that the constraint  $\Phi'$  is added to  $\Phi_1$  in actor  $\text{ref}'$  in order to store the relation between the formal and actual parameters of method  $m_1$ .
2. **(asy)** $_{\Phi_2}$ : *Object  $x$  is compatible with objects in the state but  $\rho(x)$  is not bounded in  $\Phi$ .* If  $\rho(x)$  is not bounded in  $\Phi$ , this means either that  $\rho(x) \notin \text{vars}(\Phi)$ , or that  $\Phi \not\models \rho(x) = \text{ref}'$ . 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  $\mathbf{s}^D$ . For example, for the instructions  $y = \text{new } D; x ! m_1(\bar{z})$  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  $\mathbf{s}_n^D$  and  $\rho(x) = \mathbf{s}_n^D$  in  $\Phi$ . Function  $\text{class}(x)$  returns the class of actor  $x$ .
3. **(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  $\mathbf{s}_n^D$  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 **(asy)** $_{\Phi_2}$  and **(asy)** $_{\Phi_3}$  are both applicable under the same conditions what generates non-determinism in symbolic execution.

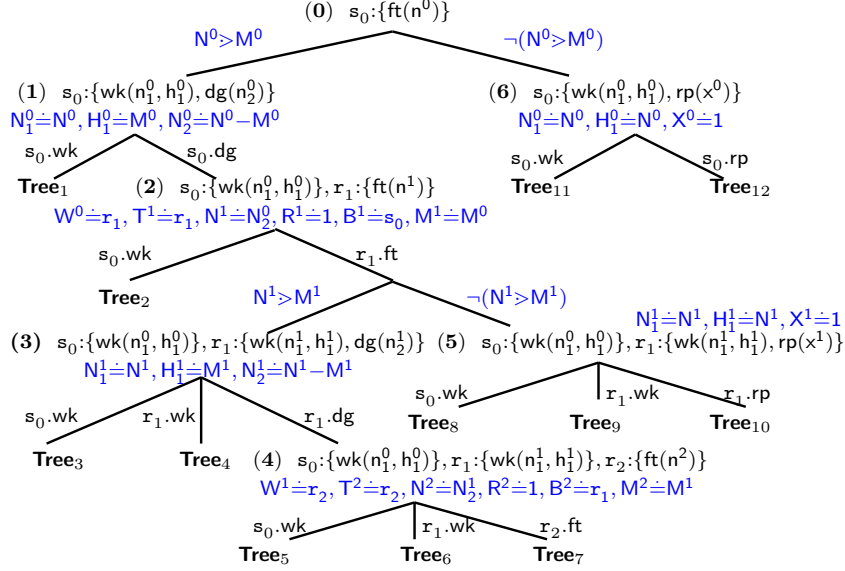


Fig. 4. Symbolic Execution for the Running Example

*Example 2.* Fig. 4 shows an excerpt of the symbolic execution tree of method `ft`. We write **Tree**<sub>*i*</sub>,  $1 \leq i \leq 12$ , 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  $s_2$ . Subscripts are used to generate fresh variables consecutively. The initial field renaming in the root node is  $\theta_s^0 = \{\text{this} \mapsto T^0, r \mapsto R^0, \text{mx} \mapsto M^0, b \mapsto B^0\}$ , the constraint attached to  $s_0$  is  $\Phi^0 = \{T^0 = s_0\}$  and the initial renaming for local variables in `ft`( $n^0$ ) is  $\rho_s^0 = \{n^0 \mapsto N^0\}$ . The left branch from node (0) corresponds to the **if** instruction for the call `ft`( $n^0$ ). The condition  $n^0 > \text{this.mx}$  produces the constraint  $\rho_s^0(n^0) > \theta_s^0(\text{mx})$ , i.e.,  $N^0 > M^0$ . Since  $\Phi^0 \models \theta_s^0(\text{this}) = s_0$  holds, rule  $(\text{asy})_{\Phi_1}$  can be applied to both asynchronous calls (the applications of  $(\text{asy})_{\Phi_2}$  and  $(\text{asy})_{\Phi_3}$  will be illustrated in the tree in Fig. 5). For the call `this ! wk`( $n^0, \text{this.mx}$ ), we generate  $\text{wk}(n_1^0, h_1^0)$  as fresh renaming for the method, together with the initial renaming  $\rho_1^0 = \{n_1^0 \mapsto N_1^0, h_1^0 \mapsto H_1^0\}$ . Hence, the constraints  $N_1^0 = N^0$  and  $H_1^0 = M^0$  are added to the constraints for  $s_0$ . Similarly, the constraint  $N_2^0 = N^0 - M^0$  originates from using `dg`( $n_2^0$ ) as renaming for `dg`( $n - \text{this.mx}$ ). The right branch of node (0) is associated to the **else** of method `ft`. In this case, the renaming  $\rho_1^0$  associated to task  $s_0.\text{wk}$  maps  $n_1^0$  (resp.  $h_1^0$ ) to  $N_1^0$  (resp.  $H_1^0$ ). Similarly, the initial renaming  $\rho_2^0$  for task  $s_0.\text{rp}$  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  $s_0.\text{dg}$  executes the call to `dg` and thus, after applying rules  $(\text{new})_{\Phi}$  and  $(\text{asy})_{\Phi_1}$ , a new actor  $r_1$  appears in (2) with a corresponding call to `ft` in its queue. The constraint  $W^0 = r_1$  is added to the constraints of actor  $s_0$ , where  $W^0$  is the fresh renaming for variable `wkr`.

From (2), if we execute  $r_1.ft$ , branches (3) and (5) are generated. From (3) the execution of  $r_1.dg$  creates a new actor  $r_2$  (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).**  $ac(ref, t, \theta, \mathcal{Q}, \Phi)$  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  $ac(ref, t, \theta, \mathcal{Q}', \Phi) \in \mathcal{S}_n$  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  $C_1:m_1 \rightarrow C_2:m_2 \rightarrow \dots \rightarrow C_k:m_k$ , 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  $C_2:m_2 \rightarrow C_{k-1}:m_{k-1}$ , 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).**  $ac(ref^{C_n}_{\cdot, \cdot, \cdot, \cdot}, \mathcal{Q}, \cdot) \in \mathcal{S}$ , is temporarily stable in  $\mathcal{S}$ , if for every  $ac(ref^{C_1}_{\cdot, \cdot, \cdot, \cdot}, \mathcal{Q}_1, \cdot) \in \mathcal{S}$ ,  $ref^{C_n} \neq ref^{C_1}$  and for every  $tk(\cdot, m_1, \cdot, \cdot) \in \mathcal{Q}_1$ , one of the following conditions holds:



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

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  $r_1$  contains  $\theta_s^1(b) = B^1$ . Points 1 and 2 of Th. 1 does not hold, since from  $ft$  (in the queue of  $r_1$ ) there exists a call to  $rp$  and  $rp$  and  $wk$  are not independent. Besides,  $B^1 \models s_0$  occurs as constraint in (2) and thus Point 3 of Th. 1 neither holds. Hence, actor  $s_0$  is not temporarily stable. However, actor  $r_1$  is temporarily stable in (2), since task  $wk$  in the queue of  $s_0$  does not call any method of class **Fact**. This means that **Tree**<sub>2</sub> in Fig. 4 is redundant and hence not expanded. A similar reasoning allows us to conclude that trees **Tree**<sub>3</sub>, **Tree**<sub>5</sub>, **Tree**<sub>6</sub> and **Tree**<sub>8</sub> are redundant. To illustrate the need of condition 2, consider a state with two actors  $r_0, r_1$ , with task  $dg$  resp.  $ft$  in the queue of  $r_0$  resp.  $r_1$  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], **Tree**<sub>1</sub> and **Tree**<sub>4</sub> are detected redundant, since tasks  $wk$  and  $dg$  are independent.  $\square$

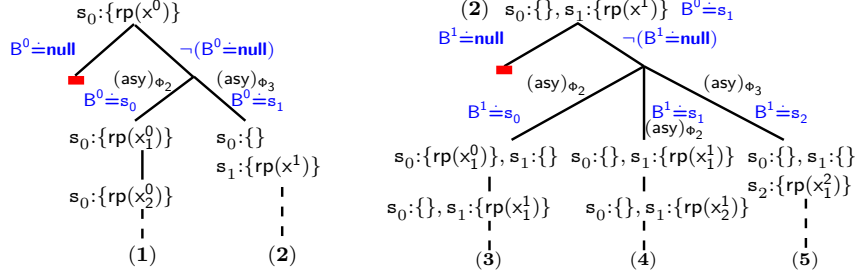
## 5 Generation of Test Cases

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  $\text{rp}(x)$  is shown 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  $\text{rp}$  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  $B^0 \doteq s_1$ .  $\square$

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

**Definition 2 (task-level CTC).** Let  $k \in \mathbb{N}^+$ . A symbolic execution  $\mathcal{E} \equiv S_0 \sqcap \mathcal{I}_0 \xrightarrow{*}_\Phi S_n \sqcap \mathcal{I}_n$  satisfies the task-level CTC iff for all actor  $\text{ref} \in S_n$ , it holds that  $\text{ref}$  has been selected at most  $k$  times in  $\mathcal{E}$  by rule  $(\text{mstep})_\Phi$ .

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  $(\text{new})_\Phi$  or  $(\text{asy})_{\Phi_3}$  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  $(\text{asy})_{\Phi_3}$ , which generates a new actor  $s_1$  when executing the asynchronous call  $\text{this.b} ! \text{rp}(\text{this.r})$ . 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  $(\text{asy})_{\Phi_3}$  generates a new actor in each execution step and thus the number of new actors grows infinitely. By annotating the instruction  $\text{this.b} ! \text{rp}(\text{this.r})$  in method  $\text{rp}$  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.  $\square$

The idea of the next CTC is to limit the application of the instructions, which introduce new actors (**new** and  $(\text{asy})_{\Phi_3}$ ) 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  $\langle x = \text{new } C, c_o \rangle$ , and  $\langle x = y ! m(\bar{z}), c_o \rangle$  respectively. When a task executes a **new** or  $(\text{asy})_{\Phi_3}$  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  $\text{fields}(\text{ref}, \Phi, \theta)$ , where  $\text{ref}$  is the reference of the actor,  $\Phi$  are the constraints for its field values and  $\theta$  is the renaming relating constraint variables in  $\Phi$  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  $\text{args}(m(\bar{x}), \Phi, \rho)$  to denote the constraints  $\Phi$  imposed on  $\bar{x}$ , together with the initial renaming  $\rho$ , which keeps the association between  $\bar{x}$  and  $\Phi$ . 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  $\mathcal{E} \equiv S_0 \sqcap \mathcal{I}_0 \xrightarrow{*}_{\Phi} S_n \sqcap \mathcal{I}_n$  be a complete symbolic execution such that  $S_0 \sqcap \mathcal{I}_0$  is an initial state, where  $\mathcal{I}_0 = \{\langle \mathbf{s}_0^C, m(\bar{x}), \theta_s, \rho_s \rangle\}$ . The test case for  $\mathcal{E}$  is defined as the tuple  $\langle \mathcal{A}_I, \mathcal{A}_O \rangle$ , where:*

$$\begin{aligned} \mathcal{A}_I &= \{\text{args}(m(\bar{x}), \Phi_I, \rho_s) \mid \text{ac}(\mathbf{s}_0^C, \_, \_, \_, \Phi) \in S_n, \Phi_I = \Pi_{\rho_s(\bar{x})}\Phi\} \cup \\ &\quad \{\text{fields}(\mathbf{s}_0^C, \Phi_I, \theta_s) \mid \text{ac}(\mathbf{s}_0^C, \_, \_, \_, \Phi) \in S_n, \Phi_I = \Pi_{\theta_s(\mathcal{F}(C))}\Phi\} \cup \\ &\quad \{\text{fields}(\mathbf{s}_k^D, \Phi_I, \theta'_s) \mid \text{ac}(\mathbf{s}_k^D, \_, \_, \_, \Phi) \in S_n, \langle \mathbf{s}_k^D, \theta'_s \rangle \in \mathcal{I}_n, \Phi_I = \Pi_{\theta'_s(\mathcal{F}(D))}\Phi\} \\ \mathcal{A}_O &= \{\text{fields}(\text{ref}_k^D, \Phi_O, \theta) \mid \text{ac}(\text{ref}_k^D, \_, \_, \_, \Phi) \in S_n, \Phi_O = \Pi_{\theta(\mathcal{F}(C))}\Phi\} \end{aligned}$$

In the above definition, we can observe that the test cases are given in terms of the constraints in  $\Phi$ . An essential aspect is that the renamings  $\rho_s$  and  $\theta_s$  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  $\Phi$  on the original names for the input arguments that were stored in  $\rho_s$ , (2) in **fields** the constraints for the actor fields that are obtained by projecting  $\Phi$  on the initial names for the actor fields that are stored in  $\theta_s$ . 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  $\theta$  until the last state.

*Example 6.* Let us consider the TCG of method  $\text{ft}(n^0)$  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 **Tree**<sub>7</sub> of Fig. 4, where:

$$\begin{aligned} \mathcal{A}_I &= \{\text{args}(\text{ft}(n^0), \{N^0 \doteq 3\}, \rho_s^0), \text{fields}(s_0^{\text{Fact}}, \{M^0 \doteq 1, B^0 \doteq \text{null}\}, \theta_s^0)\} \\ \mathcal{A}_O &= \{\text{fields}(s_0^{\text{Fact}}, \{M_a^0 \doteq 1, B_b^0 \doteq \text{null}, R_c^0 \doteq R^0 * 3 * 2 * 1\}, \theta_f^0), \text{fields}(s_1^{\text{Fact}}, \{M_d^1 \doteq 1, \\ &\quad B_e^1 \doteq s_0^{\text{Fact}}, R_f^1 \doteq 2\}, \theta_f^1), \text{fields}(s_2^{\text{Fact}}, \{M_g^2 \doteq 1, B_h^2 \doteq s_1^{\text{Fact}}, R_i^2 \doteq 1\}, \theta_f^2)\} \end{aligned}$$

being  $\mathcal{A}'_O$  as  $\mathcal{A}_O$  but replacing the first entry for  $s_0^{\text{Fact}}$  by  $\text{fields}(s_0^{\text{Fact}}, \{M_a^0 \doteq 1, B_b^0 \doteq \text{null}, R_c^0 \doteq R^0 * 3 * 1 * 1\}, \theta_f^0)$ . The renamings  $\theta_s^0$  and  $\rho_s^0$  are defined in Ex. 2 and the remaining ones are defined as  $\theta_f^i(\text{mx}) = M_-^i$ ,  $\theta_f^i(\text{b}) = B_-^i$ ,  $\theta_f^i(\text{r}) = R_-^i$ , 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  $s_1^{\text{Fact}}$ ), 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  $\mathcal{A}_1$ :

$$\begin{aligned} \mathcal{T}_I &= \{\text{args}(\text{ft}(n^0), \{n^0 \doteq 3\}), \text{fields}(s_0^{\text{Fact}}, \{\text{mx} \doteq 1, \text{b} \doteq \text{null}, \text{r} \doteq 1\})\} \\ \mathcal{T}_O &= \{\text{fields}(s_0^{\text{Fact}}, \{\text{mx} \doteq 1, \text{b} \doteq \text{null}, \text{r} \doteq 3 * 2 * 1\}), \\ &\quad \text{fields}(s_1^{\text{Fact}}, \{\text{mx} \doteq 1, \text{b} \doteq s_0^{\text{Fact}}, \text{r} \doteq 2\}), \text{fields}(s_2^{\text{Fact}}, \{\text{mx} \doteq 1, \text{b} \doteq s_1^{\text{Fact}}, \text{r} \doteq 1\})\} \end{aligned}$$

In this case, only field **r** of actor  $s_0^{\text{Fact}}$  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

Benchm.	Ignoring task indep. info				Exploiting task indep. info				Reduction	
	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 infor-

mation. 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. Degraeve, 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.

## Appendix D

Article “*Testing Abstract Behavioral Specifications*”, [24]

# Testing Abstract Behavioral Specifications <sup>★</sup>

Peter Y. H. Wong<sup>1</sup>, Richard Bubel<sup>2</sup>, Frank S. de Boer<sup>3</sup>, Miguel Gómez-Zamalloa<sup>4</sup>, Stijn de Gouw<sup>3</sup>,  
Reiner Hähnle<sup>2</sup>, Karl Meinke<sup>5</sup>, Muddassar Azam Sindhu<sup>6</sup>

<sup>1</sup> SDL, Amsterdam

<sup>2</sup> Department of Computer Science, Technische Universität Darmstadt

<sup>3</sup> CWI, Amsterdam

<sup>4</sup> DSIC, Complutense University of Madrid

<sup>5</sup> School of Computer Science and Communication, KTH Royal Institute of Technology, Stockholm

<sup>6</sup> Department of Computer Science, Quaid-i-Azam University, Islamabad

The date of receipt and acceptance will be inserted by the editor

**Abstract** We present a range of testing techniques for the Abstract Behavioral Specification (ABS) language and apply them to an industrial case study. ABS is a formal modeling language for highly variable, concurrent, component-based systems. The nature of these systems makes them susceptible to the introduction of subtle bugs that are hard to detect in the presence of steady adaptation. While static analysis techniques are available for an abstract language such as ABS, testing is still indispensable and complements analytic methods. We focus on fully automated testing techniques including blackbox and glassbox test generation as well as run-time assertion checking, which are shown to be effective in an industrial setting.

## 1 Introduction

Model-based testing is of particular importance in the context of complex *concurrent* and *highly variable* software systems. The nature of these systems makes them susceptible to the introduction of subtle bugs that are hard to spot and easy to overlook in the presence of steady adaptation. When developing software systems with high variability, for example, in the context of product line engineering [27], typically different products are generated that compute the same result (commonality) but which have differing non-functional requirements (variability), such as security levels, performance, etc. The availability of test cases with a good degree of code coverage is essential to ensure that these different products compute the same result.

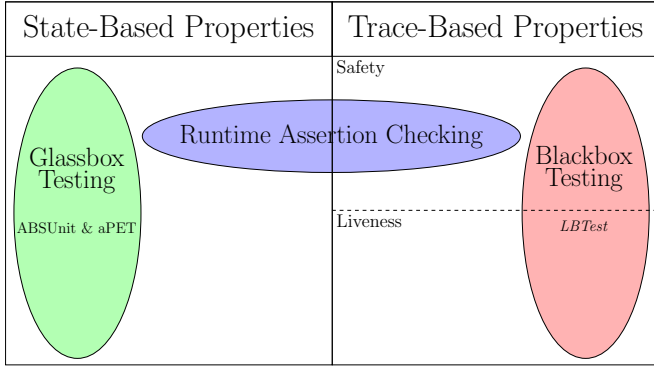
In this paper we work with a model-centric approach based on the *Abstract Behavioral Specification* (ABS) language [18,15]. ABS is an industry-strength, executable modeling language intended for highly variable, concurrent, component-based systems. ABS software models abstract away from implementation details, but retain essential behavioral aspects. ABS has an easy-to-understand concurrency model, yet permits to model precisely synchronous as well as asynchronous operations with state changes. It has been carefully designed to make static analysis techniques feasible, including type checking, deadlock analysis, resource analysis, and even functional verification [12]. Static analyses provide formal assurances of the quality, correctness and trustworthiness of ABS models. Yet they do not render testing obsolete: functional verification is often expensive and non-automatic—formal verification cannot keep up with frequent changes that typically occur during development. In addition, analysis techniques address the correctness of source code or bytecode, but do not cover compilation to machine executable code or possible bugs in runtime environments. This is where model-based testing becomes important. A selection of tests with good coverage that are run on a regularly (e.g., nightly) basis, help to discover bugs at an early stage. In addition, to guard against regression, one may generate test cases from one product variant to validate the behavior of other or later versions. Variability in software systems clearly increases the need for testing. For this reason it is very valuable that the primitives provided by ABS to describe variability also allow one to cleanly separate testing code from production code as illustrated in the ABSUnit framework in Sect. 5 below.

Testing and glassbox test generation require the system under test to be executable. This renders testing a product-level rather than a family-level activity in product line engineering [27]. In this paper we do not discuss

---

<sup>★</sup> This research is partially funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (<http://www.hats-project.eu>).





**Figure 1.** An overview of ABS testing techniques

testing at the family level, which is still an open research challenge. The issue is discussed further in Sect. 9.

Fig. 1 gives an overview of the ABS testing techniques and how they complement each other. Glassbox testing and test generation are realised on top of the ABSUnit framework and the aPET automatic test generator [1]. Glassbox techniques need access to the source code under test and are mainly suitable for testing state-based functional properties. In contrast to this, blackbox testing is used to test whether an ABS model satisfies trace-based safety or liveness properties. For this the learning-based testing tool *LBTest* [23] is used. *LBTest* does not require access to the source code and incrementally learns instead a model by observing system runs. Finally, runtime assertion checking (RAC) is used to complement glassbox and blackbox testing. It allows to check safety properties as well as state-based functional properties. Runtime assertion checking does not need explicit test cases, but instruments ABS models with assertions derived from given requirements.

Both static and dynamic analysis techniques are made available through the ABS tool suite [30]. The tool suite provides compiler backends that take ABS models and generate either executable programs in implementation languages such as Java and Scala, or rewriting systems in the language of Maude for simulation-based analyses.

In the following sections, we illustrate our testing techniques and the associated tools with an industrial case study that has been modelled with ABS [31]. The case study is described in Sect. 2. An overview of the ABS language is provided in Sect. 3. In Sect. 4 we focus on a language feature of ABS called Delta Modeling that permits modular and incremental specification of variability as well as systematic code reuse. The subsequent sections each cover one of our three testing techniques for ABS: Sect. 5 describes glassbox test generation; Sect. 6 describes run-time assertion checking, and Sect. 7 describes blackbox testing.

The purpose of this paper is *not* a detailed presentation of the theoretical foundations or the tools themselves, but to show how they are applied to a common case study and how they complement each other to in-

crease confidence in the correctness of a model. For the theory behind the employed testing techniques and detailed tool descriptions we refer to [1, 2, 11, 10, 24, 23].

Together, the technologies discussed in this paper constitute a comprehensive tool box for test automation suitable for a wide range of scenarios. While most testing approaches focus on one class of properties or on one testing approach, in this paper we demonstrate that the ABS platform plus Delta Modeling allow tightly integrated blackbox *and* glassbox, state-based *and* trace-based, static *and* dynamic testing. In Sect. 8 we show this to be the basis for a *concerted* usage of different testing approaches that exploits their complementarity.

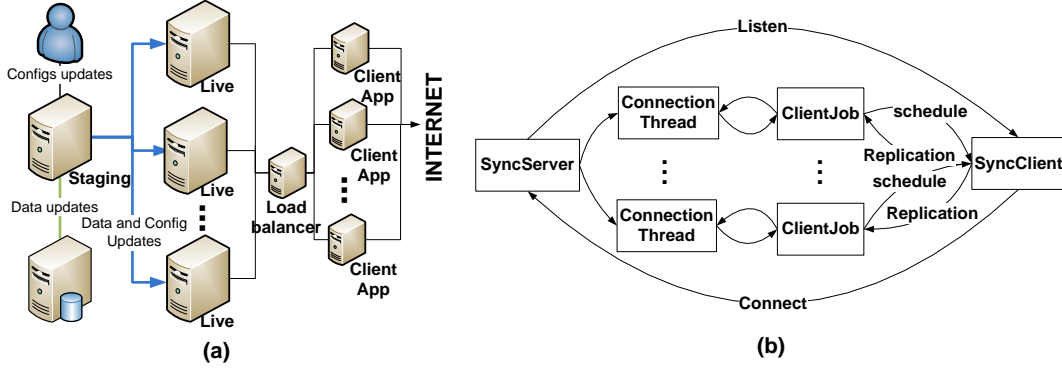
## 2 An Industrial Case Study

The Fredhopper Access Server (FAS) is a distributed, concurrent OO system that provides search and merchandising services to e-Commerce companies. FAS provides to its clients structured search and navigation capabilities within the client's data. Fig. 2(a) shows the architecture used to deploy FAS at a customer site.

FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to a *Replication Protocol*. The Replication Protocol is implemented by a *Replication System*, which consists of a *SyncServer* at the staging environment and one *SyncClient* for each live environment. The SyncServer determines the schedule of replication jobs, as well as their contents, while SyncClient receives data and configuration updates according to the schedule.

Fig. 2(b) shows the interactions in the Replication System. Informally, the Replication Protocol is as follows: the SyncServer begins by listening for connections from SyncClients. A SyncClient creates and schedules a *ClientJob* object that connects to the SyncServer. The SyncServer then creates a *ConnectionThread* to communicate with the SyncClient's ClientJob. The ClientJob asks the ConnectionThread for a *replication*, receives a sequence of file updates according to the schedule from the ConnectionThread and terminates. A complete description of the protocol can be found in [31]. In this paper we focus on the behavior of SyncClient and ClientJob.

Previously we have modeled the Replication System in ABS [31]. In this paper we specify some high-level behavioral properties about the model from which test cases, test runs and assertions are derived. The model and the specifications are provided by software engineers at SDL Fredhopper. We have also taken this case study as a usability exercise of ABS language and its tool suite [30]. While the current production version of the



**Figure 2.** (a) An example FAS deployment and (b) Interactions in the Replication System

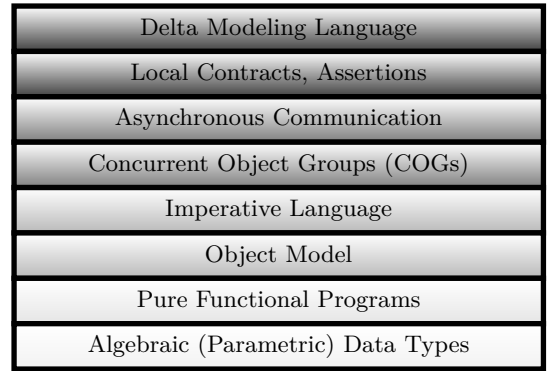
Replication System is implemented in Java, our aim is to conduct analyses on the ABS model and to generate executable production code of the Replication System from ABS. By conducting analyses on the ABS model, the generated production code would have much better guarantees over both verified and tested properties than the existing system.

### 3 Abstract Behavioral Modeling

ABS is an abstract, executable, object-oriented modeling language with a formal SOS-style semantics [18], targeting distributed systems with a high degree of variability. Many complex software systems, such as distributed services and consumer appliance software fall in this category.

Fig. 3 shows those parts of the layered architecture of ABS that are used throughout this paper: at the base are functional abstractions around a standard notion of parametric algebraic data types (ADTs). Next we have an OO-imperative layer similar to (but much simpler than) JAVA. The concurrency model of ABS is two-tiered: at the lower level it is similar to that of JCoBox [29] that generalizes the concurrency model of Creol [19] from single concurrent objects to concurrent object groups (COGs). COGs encapsulate synchronous, multi-threaded, shared state computation on a single processor. On top of this is an actor-based model with asynchronous calls, message passing, active waiting, and future types. An essential difference to thread-based concurrency is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows to write concurrent programs in a much less error-prone way than in a thread-based model and makes ABS models suitable for static analysis. Specifically, the ABS concurrency model excludes race conditions on shared data.

Fig. 4 shows some data types and interfaces used in the case study. The interface `ClientJob` models a Client-



**Figure 3.** Layered Architecture of ABS

Job, while interface `DataBase` models the database of the underlying file system of the SyncClient. The algebraic data type (ADT) `Content` models the file system of FAS environments in ABS. ADTs allow specifying immutable values in functional expressions and to abstract away from implementation details such as hardware environment, file content, or operating system specifics. Specifically, `Content` is either a `File`, where an integer (e.g., its size) is taken to represent the content of a single file, or it is a directory `Dir` with a mapping of names to `Content`, thereby, modeling a file system structure with hierarchical name space.

Interface `ClientJob` has two methods: `register(sid)` takes an integer parameter that identifies the version of the data the replication would update the live environment to; it tests whether the live environment already contains this update (it also prepares the underlying database for a possible new incoming update, but this is irrelevant for our presentation). Method `file(id)` takes a `String` value specifying the absolute path to a file stored in the live environment and returns a `Maybe` value which is either an integer representing the file content or the value `Nothing` if no such file exists.

In interface `DataBase` the method `hasFile(id)` takes the absolute path to a file and tests whether this file exists in the live environment; `getContent(id)` also takes

```

1 data Content =
2   File(Int content) | Dir(Map<String,Content>);
3
4 interface ClientJob {
5   Bool register(Int sid);
6   Maybe<Int> file(String id);
7 }
8
9 interface DataBase {
10  Bool hasFile(String id);
11  Content getContent(String id);
12 }

```

Figure 4. Data types and Interfaces

```

1 def Bool isFile(Content c) =
2   case { File(_) => True; _ => False; };
3
4 class ClientJobImpl(DataBase db)
5 implements ClientJob {
6   Maybe<Int> file(String id) {
7     Fut<Bool> he = db!hasFile(id); await he?;
8     Bool hasfile = he.get;
9     Maybe<Int> result = Nothing;
10    if (hasfile) { // if1
11      Fut<Content> f = db!getContent(id);
12      await f?; Content c = f.get;
13      if (isFile(c)) { //if2
14        result = Just(content(c));
15      }
16    }
17    return result;
18  }
19 }

```

Figure 5. Method file and auxiliary function

a path to a file and returns a **Content** value representing the content of the file identified by the input parameter.

Fig. 5 shows the implementation of method `file(id)` in class `ClientJobImpl`. It has an instance field `db` of type `DataBase`. The ADT function `isFile(c)` takes a `Content` value and returns `True` iff value `c` records a file; `content(c)` is a partial *selector* function that returns the argument of the constructor `File` (line 14).

Method `file` is implemented using the ABS features of asynchronous calls, message passing, active waiting, and future types. It first calls `hasFile(id)` on object `db` asynchronously to access the underlying file system (line 7). This call spawns a new task and returns a *future* variable `he` as a place-holder for the result of the call to `hasFile(id)`. The statement “`await he?`” suspends the current task until `he` is resolved. The result can now safely (without blocking) be accessed with `he.get` (line 8).

```

1 delta AlternativePath;
2 modifies class ClientJobImpl {
3   modifies Maybe<Int> file(String id) {
4     id = "data2/" + id;
5     Maybe<Int> res = original(id);
6     return res;
7   }
8 }

```

Figure 6. Delta AlternativePath

## 4 Delta Modeling

ABS classes do not admit code inheritance and do not define types: all object type declarations are strictly to interfaces. Code reuse is, instead, realized in the paradigm of Delta-Oriented Programming [28]. The ABS Delta Modeling Language (DML) feature [7] implements delta-oriented programming in ABS. Deltas are named entities that describe the code changes associated with the realization of new features. The result is a separation of concern between variability at the architectural/design level and algorithmic/data type aspects. This helps early prototyping and avoids a disconnect between a system’s architecture and its implementation.

For example, suppose we provide an alternative implementation of `ClientJobImpl` that accesses replication data at a different top-level directory. Fig. 6 shows a code delta `AlternativePath` that modifies the method `file` of class `ClientJobImpl`. Here the method takes a `String` value specifying the absolute path to a file and a new top-level directory as its prefix. The call **original** invokes the original implementation of `file` shown in Fig. 5, thereby achieving code reuse.

Deltas have similarities to aspects, however, their granularity is coarser (it is at the method level), they are more structured, and their application is explicitly invoked. This makes it possible to reason about the effect of changes to behavior caused by delta application [16].

Apart from addressing code reuse and variability, the DML also helps glassbox testing, in particular, for obtaining the preconditions (and invariants) of the system under test as well as for asserting its postconditions (and invariants). In the next section we shall see how deltas help to implement unit tests without code cluttering.

## 5 Glassbox Testing

Glassbox testing takes the software’s internal structure into account, which is typical for unit testing or regression testing. We present an approach for (automated) test case generation (TCG) of glassbox tests for ABS. This comprises the tools `ABSUnit`—a JUnit-like testing framework—and `aPET`, a TCG tool.

```

1 [Suite] interface AbsUnitTest {
2   [Before] Unit setup();
3   [DataPoint] Set<Pair<Int,Int>> inputData();
4   [Test] Unit testMethod1(Pair<Int,Int> comp);
5 }

```

**Figure 7.** Typical ABSUnit test interface

## 5.1 Fundamental Approach

### 5.1.1 The ABSUnit Framework

ABSUnit is an instance of the well-known XUnit test framework [17]. As usual, the first step is to implement the ABSUnit tests and to group them into test suites. ABSUnit provides the annotations [DataPoint], [Before], [After] and [Test] to indicate the purpose of a method as data input provider for parametric tests, as a fixture to set up or shut down the test environment, or as an actual unit test. The annotation [Suite] is used for an interface representing a test collection.

Fig. 7 shows a typical annotated interface for a test suite. The actual test is provided by a class implementing the interface. To specify test oracles, ABSUnit provides assertion methods such as `assertEquals(Comparator)` or `assertThat(Matcher)` (inspired by Hamcrest, see <http://code.google.com/p/hamcrest/>).

As explained in Sect. 4, ABS strictly separates subtyping and code reuse. Only interfaces declare types and can subtype each other. For testing this has two main consequences: first, there is *no* root object and thus one cannot rely on a common interface and the presence of, for example, an `equals` method. Instead, `assertEquals` uses a comparator that knows how to compare two instances of a specific kind. Second, and more importantly, implementing tests often requires to access or to change class internals (e.g., to check intermediate results or to shortcut complex initialization procedures). Here, the DML of the previous section provides an elegant solution: instead of cluttering the code base with auxiliary code, all test-related changes are organized into separate test deltas. Those deltas are only selected during product testing, but are absent from the actually shipped product. In short, in ABS *test code becomes a product feature* that is selectable at product generation time.

ABSUnit generates glue code which is responsible for test creation, test invocation (with the input provided by datapoint methods) and for setting up the test environment using fixtures. The ABSUnit test executor runs the tests and records events such as test start, passed input parameters, scheduling decisions and the test status (pass, violated assertion, or deadlock). This information is used to present and explain the test outcome.

### 5.1.2 Automatic TCG with aPET

Automatic test generation is done with aPET. By analyzing the source code, glassbox TCG aims at automatically obtaining a small set of tests with a high code coverage degree. This is in contrast to random input data generators requiring an impractically large number of inputs to reach acceptable coverage. Moreover, the maintenance of vast test suites is also impractical.

Glassbox TCG is usually done by means of *symbolic execution* [20], which represents all program execution paths up to a certain threshold, obtaining a constraint system for each symbolic path. Constraints can be seen as path conditions whose fulfillment by input data ensures that execution takes such path. Hence, solutions to path constraints can be considered as test cases.

The system aPET realizes the *Constraint Logic Programming* (CLP)-based approach to TCG [14]. The backtracking-based evaluation mechanism and constraint solving facilities of CLP are well matched to the purpose of symbolic execution. The core schema consists of two independent phases: (i) the ABS program under test is translated into an equivalent CLP program, and (ii) the CLP program is symbolically executed in CLP relying on CLP’s execution mechanism. This schema has the important property of being *flexible* and *generic*, in the sense that the second phase is essentially independent of the language for which symbolic execution has to be performed. The concrete features of the target language are abstracted in the translation and uniformly represented in CLP.

Application of this schema to the concurrent language ABS involves four steps:

1. Define an ABS to CLP compiler.
2. Implement the ABS concurrency-related operations in CLP. The scheduling policy definition is left parametric.
3. Define an appropriate coverage criterion for concurrent objects, with independent limits on both the number of task interleavings allowed and the number of loop unwindings performed in each parallel component.
4. Implement the generation of interleavings with tasks that could be initially present in the object’s queue and whose execution can affect the execution of the method under test in case it suspends. See [1] for details.

## 5.2 Tool Description

The aPET engine is implemented in the Prolog CLP system. It is packaged as a binary executable with a command-line interface. Its integration within the ABS tool suite, which is implemented in Java as an Eclipse plugin, is realized as follows: In the ABS tool suite, a handler is activated when the user requests to generate



tests for a selected set of methods in the current ABS file. The handler collects a set of user-defined parameters and the *abstract syntax tree* of the ABS program under-test, and invokes the aPET engine. The parameters include among others, the coverage criterion, the scheduling policy and the level of task interleavings to be considered. The aPET engine then compiles the provided ABS program into a CLP program and symbolically executes it according to the the provided parameters. As a result, a set of tests is generated automatically for each requested method via XML. The aPET handler finally generates ABSUnit executable tests from the XML.

Let us observe that each test exercises a different path of execution and include an automatically synthesized test oracle. As no specifications are used, aPET generates the test oracles from the actual results of the program induced by the corresponding path constraints. With such test oracles all tests will trivially pass. Therefore, the test oracles can be seen as templates that the user has to confirm or to modify.

### 5.3 Case Study

Let us consider method `file` of class `ClientJobImpl` (see Fig. 5), and as coverage criterion, *path-coverage* limited to paths with at most one loop iteration or recursive call. Note that several functions involved in the computation of method `file` are recursive. Using this coverage criterion, aPET generates 6 tests, that correspond to the following situations:

- (i) a file named “” is searched in an empty file system;
- (ii) file “a” is searched in an empty file system;
- (iii) file “a” is searched in a file system with just an empty folder named “a”;
- (iv) file “a” is searched in a file system with a folder named “a” that contains a file named “a”;
- (v) file “a” is searched in a file system with a folder named “” that contains a file named “a”; and
- (vi) file “a” is searched in a file system that just contains a file named “a”.

In the first 5 tests the return value is `Nothing`, whereas in the last one the return value is `Just(0)` (0 being the content of the file). Strings are generated starting with the empty string, then generating alphabetically strings of length 1, etc.

Fig. 8 shows the test method `testFile` that is automatically generated for test case (vi) above. Its implementation first invokes `setHeap` (line 10) to set up the initial heap, which consists of two objects `c` and `b` of types `ClientJob` and `DataBase`. Next, method `file(id)` is called on `c` and asserts that the return value is as expected. It also invokes the generated method `assertHeap` to assert that the invocation of `file(id)` changed the heap as expected.

In addition, two delta modules are automatically created to provide additional infrastructure for executing

```

1 [Fixture] interface JobTest {
2   [Test] Unit testFile();
3 }
4
5 [Suite]
6 class JobTestImpl implements JobTest {
7   ClientJob c; DataBase b; ABSAssert aut;
8   { aut = new ABSAssertImpl(); }
9   Unit testFile() {
10    this.setHeap();
11    Maybe<Int> r = c.file("a");
12    aut.assertTrue(Just(0) == r);
13    this.assertHeap();
14  }
15  Unit setHeap() { }
16  Unit assertHeap() { }
17 }

```

Figure 8. Generated test case

```

1 delta MDeltaForClientJob;
2 adds interface MClientJob extends ClientJob {
3   Unit setDB(DataBase b);
4   DataBase getDB();
5 }
6 modifies class ClientJobImpl adds MClientJob {
7   adds Unit setDB(DataBase b) { this.db = b; }
8   adds DataBase getDB() { return db; }
9 }

```

Figure 9. Modification Delta

```

1 delta TestDelta;
2 modifies class JobTestImpl {
3   modifies Unit setHeap() {
4     b = new DataBase();
5     b.setRdir(Pair("r", Entries(InsertAssoc(
6       Pair("a", Content(0)), EmptyMap))));
7     c = new ClientJobImpl(null);
8     c.setDB(b);
9   }
10  modifies Unit assertHeap() {
11    DataBase x = c.getDB();
12    Pair<String, Content> p = x.getRdir();
13    aut.assertTrue(p == Pair("r", Entries(InsertAssoc(
14      Pair("a", Content(0)), EmptyMap))));
15  }
16 }

```

Figure 10. Test Delta

test cases. Delta module `MDeltaForClientJob`, displayed in Fig. 9, completes existing interfaces and classes to permit easy setup of their initial state. For example, it provides getter and setter methods for the database object. Similar delta modules exist for the other interfaces. The delta `TestDelta`, depicted in Fig. 10, modifies the methods `setHeap` (lines 3 – 9) and `assertHeap` (lines 10 – 15) to set up the initial heap and check the final heap. Here `TestDelta` initializes the underlying file system to a pair of `String` value “r” and `Entries(...)`, where “r” is the name of the top level directory of the file system and the `Entries` value models a file named “a” with content 0 (lines 5 – 8). The delta also asserts that this value does not change when `file(id)` is executed (lines 11 – 14).

## 6 Run-Time Assertion Checking

Run-time assertion checking (RAC) is a very useful technique for detecting faults, and it is applicable during any program execution context, including debugging, testing, and production. Compared to program logics, RAC emphasizes *executable specifications*. While program logics statically cover all possible execution paths, RAC is a fully automated, on-demand validation process which applies to the actual program runs.

Assertions are inherently state-based in that they describe properties of the program variables, i.e., fields of classes and local variables of methods. As such, assertions in general cannot be used to specify the *interaction protocol* or *history* (i.e., the trace of incoming and outgoing method calls or returns) between objects. This is in contrast to other formalisms such as message sequence charts and sequence diagrams. Nor do assertions support interface specifications (fundamental in ABS, as all object references are typed by interfaces), since interfaces are stateless and contain only method signatures. There exist many interesting approaches to run-time monitoring of histories, including PQL [22], Tracematches [3], JmSeq [25], LARVA [8], Jass [4], and JavaMOP [5]. However, none of these address the integration into the general context of run-time assertion checking: they allow specifying protocol-oriented properties, but do not provide a systematic solution to specify the data-flow of the valid histories. Hence, the question arises how to integrate protocol-oriented properties and assertions into a single formalism, in a manner amenable to automated verification, in particular to run-time checking.

### 6.1 Fundamental Approach

In [11] we identified attribute grammars with conditional productions and annotated with assertions as powerful and user-friendly specifications of histories. This approach was extended to coboxes in [10]. Grammars specify *invariant* properties of the ongoing behavior (of a

single object, a COG, or an entire ABS model) and as such must be prefix-closed. Context-free grammars express the protocol structure (i.e., orderings between events) of the valid histories in a declarative manner. Context-free grammars, however, do not take data into account, such as actual parameters and return values of method calls. The question arises how to specify the *data flow* of the valid histories. To this end we extend the grammars with attributes. Terminals in the grammar have *built-in* attributes such as the actual parameters, return value and the identity of the caller and callee. Non-terminals have *user-defined* attributes which define data properties of sequences of terminals. Assertions annotating this attribute grammar then provide a natural way to express user-defined properties of these attributes. In other words, assertions specify the allowed attribute values of histories. This does not yet allow to directly express *data-dependent* protocols. Such protocols are quite common in practice, for example, the `next` method of a Java Iterator may not be called, whenever method `hasNext` was called directly before and returned false. Conditional productions address this problem.

To support focussing on a particular behavioral aspect of communication involving data-dependent protocols, we use the general mechanism of a *communication view*. A communication view is a partial mapping from events to grammar terminals. Events not associated to terminals are projected away and play no role in the grammar. This reduces the size of the histories, allows using intuitive names for the selected events and keeps the size and complexity of the grammars low. Moreover, communication views enable the introduction of abstractions of the communication by identifying two distinct events with the same grammar terminal.

In summary, the valid event histories are represented as words generated by an extended attribute grammar. Grammar productions (possibly conditional) specify the valid protocol structure of histories, while assertions express the valid data-flow of histories.

### 6.2 Tool Description

Our RAC combines three components: the parser generator ANTLR, the ABS compiler, and the meta programming system Rascal [21]. The ABS compiler generates JAVA code for the attribute definitions in the attribute grammar. The result is an attribute grammar defined in the syntax of ANTLR [26]. ANTLR, a JAVA parser generator, then generates a lexer and a parser for the grammar in JAVA.

Rascal is a general meta-programming language tailored for program transformations. We extended Rascal with support for ABS. Our RAC uses Rascal for several tasks: it first parses the communication view, the ABS method signatures, and the attribute grammar. Based on the parsing results, it generates code for a history

```

1 local view ClientJobProtocol specifies ClientJob {
2   return Bool register(Int sid) r,
3   call Maybe<Int> file(String id) f,
4   call Content DataBase.getContent(String id) c
5 }

```

Figure 11. Communication View

class (a datatype suitable to represent the communication history of an ABS object or COG) and instruments ABS source code around method calls and returns to update the current history. The history class calls the JAVA parser (which was generated by ANTLR) when the history is updated to obtain new attribute values.

### 6.3 Case Study

We consider the `ClientJob` interface in Fig. 4 introduced in Sect. 3 with the following property: in a replication session, the `register(sid)` method is called initially with `sid` indicating the version of data the replication would update the client to. The method returns a `Bool` value indicating whether the client accepts this replication. If the returned value is `True` then the method `file(id)` may be called one or more times, each time with a unique `String` value representing the absolute path of a file. After each invocation of `file(id)`, an *outgoing* method invocation on `getContent(id)` of `Database` may be made with a value that must be the same absolute path as that supplied in the preceding method `file(id)`.

The communication view in Fig. 11 introduces the relevant events which can be referred to in the grammar by the terminals `r`, `f`, and `c`. Fig. 12 shows the attribute grammar formalizing the property stated informally above. Attribute definitions are written between normal brackets `('` and `')`. The first production formalizes the call to `register(sid)`, where the inherited attribute `rg` stores the return value and the attribute `ns` contains the `List` of file names processed so far by `file(id)` (initially, `Nil`). Note that epsilon productions are used to make the grammar prefix-closed, and that all attributes are inherited (i.e. passed down the parse tree) since the attributes of the non-terminals on the right-hand side of each grammar production are defined in terms of the attributes of the non-terminals on the left-hand side. The second production captures a call to `file(id)` and checks that the current `id` is new in `ns`. The condition `{ T.rg == True }` formalizes that the value returned by `register(sid)` was `True`. The third production handles the outgoing call and checks that the filenames match. It also allows to call `file(id)` again via the non-terminal `T`.

Some data types used in the grammar are defined in Fig. 13. Function `contains(ss,e)` checks whether the list `ss` contains the element `e`, while `head(ss)` is a partial

```

1 data List<A> = Nil | Cons(A head, List<A> tail);
2 def Bool contains<A>(List<A> ss, A e) =
3   case ss {
4     Nil => False ;
5     Cons(e, _) => True;
6     Cons(_, xs) => contains(xs, e);
7   };

```

Figure 13. List data type

selector function that returns the first element of a non-empty list `ss`.

We have developed two versions of our RAC approach for Java and ABS. Using the Java version we have successfully integrated runtime assertion checking into the software lifecycle at SDL Fredhopper. Full detail can be found in [9]. Using the ABS version we have conducted experiments with the ABS model of the Replication System and have consequently detected crucial protocol violation in the model. Full detail of this case study can be found in [10].

## 7 Blackbox Testing

### 7.1 Fundamental Approach

Learning-based testing (LBT) [23] is an emerging paradigm for *black-box requirements-testing* that encompasses the three steps of: (i) automated test case generation (ATCG), (ii) test execution, and (iii) test verdict (the oracle step). LBT is related to model-based testing (MBT). However, where MBT starts from a system design model which is then used to generate test cases, in LBT a model is inferred automatically from an SUT implementation using computational learning methods (*reverse engineering*). This approach has advantages for testing systems which are undocumented, and for agile development methods where the cost of model development and model synchronisation with code updates is considered too high.

LBT is an iterative procedure that attempts to generate a large volume of high quality test cases. On each iteration, the currently inferred model is checked against a user requirement to search for a counterexample to requirement correctness. For this process, requirements must be formalised within a logic, such as first-order or temporal logic. This allows constraint solving or model checking technology to be used in the search for counterexamples. If a counterexample to correctness can be found, this must be executed on the system under test to determine whether it is a true negative or a false negative. True negatives can be returned as failed test cases. False negatives can be integrated, via a *learning algorithm*, into the inferred model to refine its accuracy. In

S	::= $\epsilon$	r T (T.rg = r.result; T.ns = Nil;)
T	::= $\epsilon$	{ T.rg == True }? f { <b>assert</b> ! contains(T.ns, f.id); } V (V.ns = Cons(f.id, T.ns); V.rg = T.rg;)
V	::= $\epsilon$	c { <b>assert</b> head(V.ns) == c.id; } T (T.ns = V.ns; T.rg = V.rg;)

**Figure 12.** Attribute Grammar for the ClientJob Behavior

this way, the inferred model will converge to a complete and correct model of the system under test, as increasing numbers of test cases are executed. Note that in the case that no counterexample can be found, some other test case generation method must be used to proceed with the iteration (see below). If the learning algorithm always converges correctly, and if counterexample search is a decidable problem, then LBT is a sound and complete method of testing. However, for large industrial SUTs, complete learning may not be feasible in the time available. For this reason, many optimisations of learning must be considered. One such optimisation is *incremental learning*, which can infer an incomplete model from relatively little test data. Such incomplete models can nevertheless uncover SUT errors. Further details about learning optimisation can be found in [23].

The Fredhopper access server is an example of a reactive system that can be learned as a state machine. Indeed any client-server architecture can be modeled and learned in this way. In this case, an *automata inference algorithm* is needed to reverse engineer models, and temporal logic is widely considered to be the most useful logic to formalise user requirements. Then efficient model checking algorithms can be employed to search for counterexamples. An early application of LBT to testing reactive systems was given in [24], and since then other classes of reactive systems have also been tested in this way (see e.g. [13]).

To interpret the testing results obtained for the Fredhopper access server, it will be helpful to consider in more detail the abstract LBT algorithm used. An LBT architecture automatically generates a large number of high-quality test cases by combining a model checking algorithm with a learning algorithm and a random test case generator. Note that *active learning algorithms*, which can generate their own queries are both appropriate (i.e. they can generate test cases) and efficient (i.e. in polynomial time). These three algorithms are integrated with the system under test (SUT) in an iterative feedback loop (see Fig. 14). On each iteration of this loop, a new test case is generated by one of the three TCG methods, i.e.: (i) model checking the most recent learned model  $m_n$  of the SUT against a formal user requirement  $\Phi$  and choosing any counter example to correctness; (ii) using the active learning algorithm to generate a membership query; (iii) random test case generation. The LBT tool must *interleave* these three TCG methods to achieve an overall testing strategy that is efficient.

Whichever TCG method is used, the new test case  $i_n$  is then executed on the SUT with outcome  $o_n$ . The outcome of a test case is judged as a *pass*, *fail* or *warning*. This is done after each model checking step, by generating a predicted output  $p_n$  (obtained from  $m_n$ ) that can be compared with the observed output  $o_n$  (from the SUT). Each new input/output pair  $(i_n, o_n)$  is used to update the current model  $m_n$  to a refined model  $m_{n+1}$ , which ensures that the iteration can proceed again. The overall LBT architecture is illustrated by the diagram in Fig. 14.

## 7.2 Tool Description

A platform for learning-based testing known as *LBTest* (see [?]) has been developed for blackbox testing of ABS and other reactive systems models. The *LBTest* tool supports the integration of different model inference algorithms with different model checkers to conduct experiments in learning-based testing. The main inputs to the tool are the SUT and a set of formal user requirements to be tested. For formal requirements modeling, the main language currently supported is *propositional linear temporal logic* (PLTL). LTL is chosen, since it naturally models the black-box (input/output) behaviour of reactive systems. The restriction of LTL to propositions only (i.e. PLTL rather than full first-order LTL) is because: (i) PLTL model checking is decidable, and (ii) there exist fast algorithms for model checking PLTL formulas such as BDD based methods.

Note that PLTL formulas can express both *safety properties* which may not be violated, and *liveness properties*, including *use cases*, which specify intended behaviors. Some liveness properties cannot be refuted in finite time (for example termination properties). For such types of properties, *LBTest* is able to issue a *warning verdict* that a test case has never been seen to have passed. Therefore, both types of requirements are amenable to testing using *LBTest*.

Currently in *LBTest*, only one model checker is supported, which is NuSMV [6]. This model checker has been adopted mainly for its stability and wide user base. In principle, any other model checker or even a bounded model checker for PLTL could also be used. The learning algorithm currently available in *LBTest* is the IKL learning algorithm described in [24], which is an algorithm for learning deterministic Boolean-valued Kripke structures.



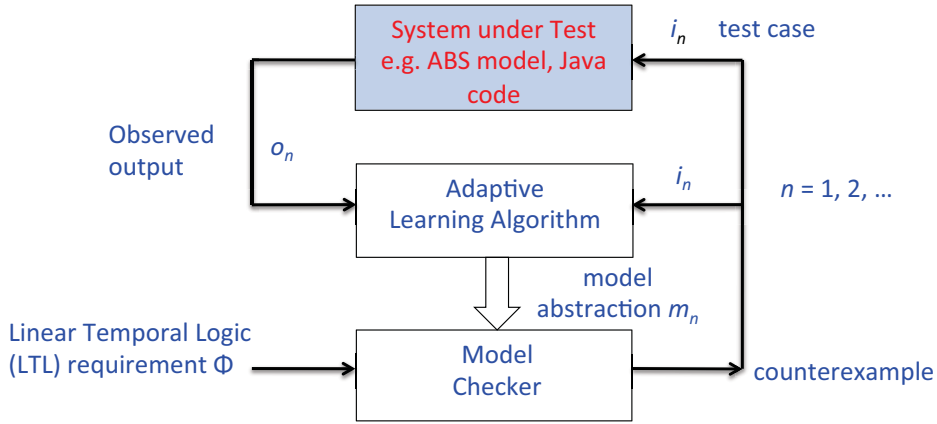


Figure 14. Architecture of learning-based testing

Other automata learning algorithms are currently being investigated for their performance in testing.

### 7.3 Tool Interface

For practical testing, propositional linear temporal logic is much too low level to express user requirements succinctly. Therefore, the language PLTL is augmented with finite symbolic data types, which support user defined data type declarations. A data type declaration  $\Sigma$  includes type declarations and value declarations. A type declaration defines a finite set of output types  $t_1, \dots, t_n$ . (Note that a single input type, with the reserved type name  $in$ , is always assumed.) A value declaration for each type  $input, t_1, \dots, t_n$  consists of a finite set of constant symbols  $c_1, \dots, c_m$  of that type. This data type declaration constitutes the only interface specification needed for the system under test. Each symbolic data value (both input and output) must be mapped into a concrete native data value according to the programming language used for the SUT. This mapping code is stored in a thin wrapper between the SUT and LBTest, which also acts as a communication manager between LBTest as a server and the SUT as a client.

We were interested to test the interaction between a SyncClient and a ClientJob by learning the SyncClient as a deterministic Kripke structure (Moore machine) over the input data type

$$\Sigma_{in} = \{setAcceptor, schedule, searchjob, businessJob, dataJob, connectThread, noConnectionThread\}$$

Four relevant output data types were identified as follows:

$$\Sigma_{schedules} = \{\phi, \{search\}, \{business\}, \{business, search\}, \{data\}, \{data, search\},$$

$$\{data, business\}, \{data, business, search\}\}.$$

$$\Sigma_{state} = \{Start, WaitToBoot, Boot, WaitToReplicate, WorkOnReplicate, End\},$$

$$\Sigma_{jobtype} = \{nojob, Boot, SR, BR, DR\},$$

$$\Sigma_{files} = \{readonly, writeable\}.$$

### 7.4 Case Study

The *LBTest* tool was applied to the problem of black-box testing an ABS model of the Fredhopper FAS case study described in Sect. 2. An executable SUT was obtained by compiling the ABS model into JAVA code, and writing a thin wrapper to encode the symbolic input and output data types in Java. A total of 11 user requirements were modeled in PLTL. For example, requirement 9 was: “The SyncClient cannot modify its underlying file system (files = readonly) unless it is in state WorkOnReplicate.” A PLTL formalisation is:

$$\begin{aligned} & \mathbf{G} (state = WorkOnReplicate \rightarrow \\ & \mathbf{X} (files = writable \mathbf{U} state \in \{End, WaitToReplicate\}) \\ & \quad \wedge state \neq WorkOnReplicate \rightarrow \\ & \mathbf{X} (files = readonly \mathbf{U} state = WorkOnReplicate)) \end{aligned}$$

Table 1 gives the results obtained by running *LBTest* to test these 11 user requirements on the FAS SyncClient. For each requirement, Table 1 breaks down the total number of test cases used into three figures (columns 5, 6 and 7) which count the test cases generated by each of the three different TCG methods: *model checker*, *learner* and *random*. The total testing time (column 3) is the total time taken to execute all three types of test cases, which were interleaved. For each requirement, Table 1 gives the final verdict (column 2) i.e. *pass/fail/warning*. Column 4 gives the size of the learned hypothesis model at test termination. To terminate each

experiment, a maximum time bound of 5 hours was chosen. However, if the hypothesis model size had not changed over 10 consecutive random tests, then testing was terminated earlier than this.

Thus for example: Requirement 1 was tested for a total of 5 hours using 50,942 test cases, of which 50,897 were generated by the learning algorithm, 45 were generated randomly, and 0 were generated by the model checker. We see that learner generated queries dominate, though generally this is influenced by the kind of learning algorithm used (here IKL). In fact, looking across all requirements we can see that the ratio of random plus model checker queries to learner queries is about 1:1000. This means that each new model  $m_{n+1}$  is inferred from  $m_n$  after intervals of about 1000 learner queries. This ratio is a property of the IKL learning algorithm itself, and can only be influenced by choosing other learning algorithms.

Around 10,000 test cases per hour were generated, executed and evaluated. We can see that this test throughput does not vary much across the 11 different requirements. On large SUTs, test throughput is mainly determined by the average execution speed of a single test case. Since Requirement 1 was passed, while 45 random and 0 model checker test cases were used, we can infer that the model checker was called 45 times, but on each occasion it failed to find a counterexample, so that a random test case was used instead.

Finally notice that the number of states in the final hypothesis automaton is rather small (8 states). The other requirements yield hypothesis automata of similar sizes. These figures suggest that while the total state space of the access server is almost certainly very large (a completely accurate model would have an infinite state space), the system abstraction learned by LBTest to analyse each specific property can be quite small. Nevertheless, it is not clear whether complete learning of the state space has been achieved for any requirement. Complete learning is not only difficult to achieve, in a black-box testing regime it is even difficult to detect. For we have no direct access to the SUT code, and in any case equivalence checking the SUT code (an arbitrary program) with the learned model is infeasible. This highlights the importance of incremental learning in black-box testing context. The development of appropriate *coverage models*, to answer this question in a relative way, is an important open problem for the field.

Nine out of eleven requirements were passed. For requirements 8 and 9, LBTest gave warnings corresponding to tests of liveness requirements that were never seen to have passed. A careful analysis of these requirements showed that both involved using the U (strong until) operator. When this was replaced with a W (weak until) operator no warnings for Requirement 9 were seen. Recall that under the *strong interpretation* of  $p$  until  $q$ , written  $pUq$ , then  $q$  must eventually become true. However under the *weak interpretation* of  $p$  until  $q$ , written

**Table 2.** Metrics of JAVA and ABS of the Replication System

Metrics	JAVA	ABS
Nr. of lines of code	6400	3300
Nr. of classes	44	40
Nr. of interfaces	2	43
Nr. of data types	N/A	17

$pWq$ , then  $q$  may never become true if  $p$  holds forever. Thus using  $W$  instead of  $U$  usually gives a weaker user requirement that is easier to satisfy, i.e. less likely to yield test errors.

After replacing  $U$  by  $W$ , LBTest continued to produce warnings for Requirement 8. The final conclusion is that LBTest had successfully identified one error in the requirements and one error in the SUT.

## 8 Discussion

The ABS model of the Replication System considered in the case studies forms a part of the Fredhopper Access Server (FAS) whose current in-production JAVA implementation has over 150,000 lines of code, of which over 6,000 lines constitute the Replication System. Due to its concurrent behavior and the presence of numerous features, the Replication System is one of the most complex parts of FAS.

Table 2 shows metrics for the actual implementation and the ABS model of the Replication System. When comparing the numbers it is important to know that the ABS model includes modeling-level aspects such as deployment components and simulation of external inputs in the ABS model, which the JAVA implementation lacks. The ABS model includes also scheduling information, as well as models of file systems and data bases, whereas the JAVA implementation leverages libraries and its API. This accounts for >1,000 lines of ABS code. The construction of the first version of the ABS model took around 3 person months. The model was subsequently revised and extended to capture other behavioral aspects of the Replication System, such as timing information and variability. Furthermore, while the JAVA implementation is a relatively stable part of FAS, bugs had been identified and fixed. When there was a change in the JAVA implementation, the ABS model was then updated accordingly.

The quality assurance process at Fredhopper (as in many other software companies) includes automated testing. Unit tests are written manually to validate the behavior of methods and to detect regressions. A continuous integration server executes all unit tests every time a change is done to the code base of the product. To leverage the results reported in this paper, manually defined unit tests can be replaced by high coverage test cases *automatically* generated by aPET. System tests, on

**Table 1.** Performance of *LBTest* on the FAS case study

PLTL Req	Verdict	Total testing time (hours)	Hypothesis size (states)	MC queries	Learner queries	Random queries
Req 1	pass	5.0	8	0	50,897	45
Req 2	pass	5.0	15	2	49,226	13
Req 3	pass	1.7	11	0	16,543	17
Req 4	pass	2.1	11	0	20,114	14
Req 5	pass	2.5	11	0	24,944	17
Req 6	pass	2.3	11	0	23,215	16
Req 7	pass	2.1	11	0	18,287	17
Req 8	warning	1.9	8	15	18,263	12
Req 9	warning	3.8	15	18	35,831	18
Req 10	pass	2.7	11	0	26,596	19
Req 11	pass	4.6	11	0	45,937	21

the other hand, are executed twice a day on instances of FAS on a server farm. Two types of system tests are scenario and functional testing. Scenario testing executes a set of programs that emulate a user and interact with the system in predefined sequences of steps (scenarios). At each step they perform a configuration change or a query to FAS, make assertions about the response from the query, etc. Function testing executes sequences of queries, where each query-response pair is used to decide on the next query and the assertion to make about the response. Both types of tests require a running FAS instance and can be augmented with RAC techniques described in Sect. 6. Moreover, by formalising scenarios using PLTL, scenario testing can be augmented with blackbox testing using *LBTest*. In summary, the various testing approaches provided for ABS models have the potential to *substantially increase automation and coverage at the unit, scenario, and function testing level*.

The three test approaches discussed here should be used in concertation in such a way that their complementarity can be exploited. To give one example, given a high-level specification with ABS interfaces, one can generate test cases from class implementations using aPET to validate whether the implementations match the specification. We demonstrated this in Sect. 5 when we generated tests for the `ClientJobImpl` that cover all paths specified by a given coverage criteria. Another example is the combined application of *LBTest* and RAC during system testing. RAC makes assertions about object interaction which are specified in terms of attribute grammars as exemplified by our specification of a property of the `ClientJob` protocol. However, RAC checks those assertions only if corresponding execution paths are visited during a system run. Conversely, *LBTest* actively interacts with the SUT to learn a model that is then checked against PLTL formulae. This means *LBTest* attempts to trigger the execution paths corresponding to the formulae. Restricting the specification of properties to PLTL makes proving such properties on the model decidable. Note that *LBTest* checks both safety and liveness prop-

erties while run-time assertion checking aims merely at safety properties.

To achieve scalability and full automation at the same time, it was essential to work in a model-based framework. Our results would not have been possible at the level of implementation languages, such as JAVA or C++. On the other hand, it is perfectly possible to compile ABS test cases and runtime assertion checks into any of the target languages supported by ABS code generation, which includes JAVA.

We stress that, while ABS is a modeling language, it implements such concepts as interfaces, shared heap access, and asynchronous concurrent execution. This permits precise modeling and realistic simulation [30]. Delta modeling not only permits to factor out the commonality in modeled software, but is pragmatically very useful to achieve a clean separation between test code and productive code at product build time.

## 9 Conclusion

We presented a modeling framework based on the language ABS that enables extensive test automation for a complementary, yet fully integrated set of testing approaches. All techniques are fully implemented and were evaluated with an industrial case study. The different testing techniques cover different kinds of properties and complement each other with respect to their requirements such as having access to source code, or the availability of specifications in the form of assertions or temporal logic formulas (see also Fig. 1). We showed in particular that testing can be performed on models of highly distributed systems, and, even further, how formal methods enable us to automate large parts of testing and test case generation.

As future work, we would like to lift automated testing techniques from the product to the family level in product line engineering [27]. First ideas on how to approach this exist, such as sharing test cases (and test

runs) between products in case the products are identical or overlap with respect to the executed code. Work in the direction of compositionality [2] of glassbox test generation exhibits further potential to produce reusable test cases. In black box requirements testing, it becomes important to integrate product variability points into formal requirements languages such that during application engineering [27], when variability points are being instantiated for specific products, requirements may also be instantiated for those products.

## Acknowledgements

We thank Andreas Kohn for proof reading this paper.

## References

1. Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Towards Testing Concurrent Objects in CLP. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 98–108, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
2. Elvira Albert, Miguel Gómez-Zamalloa, José Miguel Rojas, and German Puebla. Compositional CLP-based test data generation for imperative languages. In *LOPSTR 2010 Revised Selected Papers*, volume 6564 of *LNCS*. Springer-Verlag, 2011.
3. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05*, pages 345–364, New York, NY, USA, 2005. ACM.
4. Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with Assertions. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 55(2):103 – 117, 2001.
5. Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA '07*, pages 569–588, New York, NY, USA, 2007. ACM.
6. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Verifier. In *Proc. of CAV 1999*, volume 1633 of *LNCS*, 1999.
7. Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudi Schlatte, and Peter Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer-Verlag, 2011.
8. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM '09*, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.
9. Frank S. de Boer, Stijn de Gouw, Einar Broch Johnsen, and Peter Y. H. Wong. Run-time assertion checking of data- and protocol-oriented properties of java programs: An industrial case study. *LNCS Transactions on Aspect-Oriented Software Development (TAOSD)*, 2013. Special Issue on Runtime Verification and Analysis. To appear.
10. Frank S. de Boer, Stijn de Gouw, and Peter Y. H. Wong. Run-time verification of coboxes. In *Proceedings of 11th International Conference on Software Engineering and Formal Methods*, volume 8137 of *LNCS*, pages 259–273, 2013.
11. Stijn de Gouw, Jurgen Vinju, and Frank de Boer. Prototyping a tool environment for run-time assertion checking in JML with Communication Histories. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs, FTFJP '10*, pages 6:1–6:7, New York, NY, USA, 2010. ACM.
12. Analysis Final Report, December 2012. Deliverable 2.7 of project FP7-231620 (HATS), available at <http://www.hats-project.eu>.
13. Lei Feng, Simon Lundmark, Karl Meinke, Fei Niu, Mudassar A. Sindhu, and Peter Y. H. Wong. Case studies in learning-based testing. In *Proc. Twenty Fifth IFIP Int. Conf. on Testing Software and Systems (ICTSS 2013)*, volume 8254 of *LNCS*, pages 164–179. Springer, 2013.
14. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming (ICLP'10) Special Issue*, 10 (4–6):659–674, July 2010.
15. Reiner Hähnle. The Abstract Behavioral Specification language: A tutorial introduction. In Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer-Verlag, 2013.
16. Reiner Hähnle, Ina Schaefer, and Richard Bubel. Reuse in software verification by abstract method calls. In Maria Paola Bonacina, editor, *Proc. 24th Conference on Automated Deduction (CADE), Lake Placid, USA*, volume 7898 of *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2013.
17. Paul Hamill. *Unit Test Frameworks*. O'Reilly Media, November 2004.
18. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.
19. Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, March 2007.

20. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
21. Paul Klint, Tijs van der Storm, and Jurgen Vinju. RAS-CAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.
22. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.
23. K. Meinke, F. Niu, and M. Sindhu. Learning-Based Software Testing: A Tutorial. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Communications in Computer and Information Science, pages 200–219. Springer-Verlag, 2012.
24. K. Meinke and M. Sindhu. Incremental learning-based testing for reactive systems. In *Proc Fifth Int. Conf. on Tests and Proofs (TAP2011)*, number 6706 in Lecture Notes in Computer Science, pages 134–151. Springer-Verlag, 2011.
25. B. Nobakht, M. M. Bonsangue, F. S. de Boer, and S. de Gouw. Monitoring method call sequences using annotations. In *Proceedings of the 7th international conference on Formal Aspects of Component Software*, FACS'10, pages 53–70, Berlin, Heidelberg, 2012. Springer-Verlag.
26. Terrence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.
27. Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
28. Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, September 2010.
29. Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.
30. Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.
31. Peter Y. H. Wong, Nikolay Diakov, and Ina Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study. In *Proc. of FoVeOOS 2011*, volume 7421 of *LNCS*, 2012.

## Appendix E

Tool Demo “*SYCO: A Systematic Testing Tool for Concurrent Objects*”, [7]

# SYCO: A Systematic Testing Tool for Concurrent Objects

Elvira Albert

Complutense University of Madrid  
elvira@fdi.ucm.es

Miguel Gómez-Zamalloa

Complutense University of Madrid  
mzamalloa@fdi.ucm.es

Miguel Isabel

Complutense University of Madrid  
miguelis@ucm.es

## Abstract

We present the concepts, usage and prototypical implementation of SYCO: a SYstematic testing tool for *Concurrent Objects*. The system receives as input a program, a selection of method to be tested, and a set of initial values for its parameters. SYCO offers a visual web interface to carry out the testing process and visualize the results of the different executions as well as the sequences of tasks scheduled as a sequence diagram. Its kernel includes state-of-the-art partial-order reduction techniques to avoid redundant computations during testing. Besides, SYCO incorporates an option to effectively catch *deadlock* errors. In particular, it uses advanced techniques which guide the execution towards potential deadlock paths and discard paths that are guaranteed to be deadlock free.

## 1. Motivation

Testing is the most widely-used methodology for software validation in industry. Several studies point out that it requires at least half of the total cost of a software project. Software testing tools urge especially in the context of concurrent programming. This is because writing correct concurrent programs is more difficult than writing sequential ones as with concurrency come additional hazards not present in sequential programs such as race conditions, deadlocks, and livelocks. In order to catch such errors, the testing tool must consider the non-determinism caused by the fact that an execution can lead to different solutions depending on the way that the involved tasks interleave, and, ideally, all possible interleavings must be considered. A systematic exploration of the state space is usually not feasible. A lot of research has been done in the context of testing and model checking with the aim of avoiding redundant state exploration as much as possible [1, 2, 5, 10, 11]. SYCO is a testing tool that targets the ABS concurrent objects language [8] and that incorporates state-of-the-art partial-order-reduction (POR) techniques to avoid redundant exploration.

Essentially, a concurrent object is a monitor that allows at most one *active* task to execute within the object. Scheduling among the tasks of an object is cooperative, or non-preemptive, meaning that the active task has to release the object lock explicitly (using the *await* or *return* instructions). Each object has an unbounded set of pending tasks. When the lock of an object is free, any task in the set of pending tasks can grab the lock and start executing. Each object has a local heap or memory (set of fields) which can

only be accessed from the owner object. The instruction  $f = \text{ob}!.m()$  creates an asynchronous task to execute method  $m$  on object  $\text{ob}$ . Synchronization can be performed using the *future variable*  $f$ , namely the instruction *await*  $f$  checks if the execution of the asynchronous task has finished. If not, the object lock is released and the task suspends until the value of  $f$  is ready. In contrast, the instruction  $v = f.\text{get}$  blocks the task until  $f$  is ready retaining the object lock. Once the execution of the task finishes, it assigns the obtained value to  $v$ .

**Running Example.** The following example simulates a simple communication protocol between a database and a worker.

```
1 {\main block
2   DB db = new DB();
3   Worker w = new Worker();
4   db!register(w);
5   w!work(db);
6 }
7 class DB{
8   Data data = ...;
9   Worker cl = null;
10  void register(Worker w){
11    Fut<Int> f = w!ping(5);
12    if (f.get == 5) cl = w;
13  }
14  Int getD(Worker w){
15    if (cl == w) return data;
16    else return null;
17  }
18 } // end class DB
19 class Worker{
20   Data data;
21   void work(DB db){
22     Fut<Data> f = db!getD(this);
23     data = f.get;
24   }
25   Int ping(Int n){return n;}
26 } // end of class Worker
```

The main method creates the two objects and invokes methods *register* and *work* resp. The *work* method of the worker simply accesses the database (invoking asynchronously method *getD*) and then blocks until it gets the result, which is assigned to its *data* field. The *register* method of the database, first checks that the worker is online (invoking asynchronously method *ping*), then blocks until it gets the result, and finally it registers the worker by storing its reference in its *cl* field. Method *getD* of the database returns its *data* field if the caller worker is registered, otherwise it returns *null*.

Depending on the sequence of interleavings, the execution of this program can finish: (i) as expected, i.e., with  $w.\text{data}$  having the same value as  $db.\text{data}$ , (ii) with  $w.\text{data} = \text{null}$ , or, (iii) in a deadlock. (i) happens when the worker is registered in the database (assignment in L12) before *getD* is executed. (ii) happens when *getD* is executed before the assignment at L12. A deadlock is produced if both *register* and *work* start executing before *getD* and *ping*. Sect. 2 and App. A illustrate how SYCO shows these different execution scenarios.

## 2. The SYCO Tool

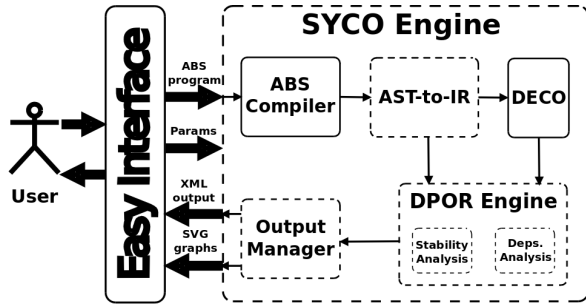
The figure above shows the main architecture of SYCO. Boxes with dash lines are internal components of SYCO whereas boxes with regular lines are external components. The user interacts with SYCO through its web interface which is provided by *EasyInterface* [7]. Basically *EasyInterface* provides a generic IDE which can be instantiated to different languages and compilers and where external plugins can be easily added. The SYCO engine receives an ABS program and a selection of parameters. The *ABS compiler*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

CC'16, Barcelona, Spain.

Copyright © 2016 ACM 978-1-55588-111-1/16/0001...\$15.00.  
<http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn>

compiles the program into an abstract-syntax-tree (AST) which is then transformed into the SYCO intermediate representation (IR). The *DPOR engine* carries out the actual systematic testing process. It comprises the ABS semantics, the DPOR algorithm of [2] and the *stability* and *dependencies* analyses of [2]. The *output manager* then generates the output in the format which is required by EasyInterface, including an XML file containing all the EasyInterface commands and actions and the SVG diagrams. In case a deadlock-guided testing is requested (see the corresponding parameter below), the *DECO* deadlock analyzer [6] is invoked, which returns a set of potential deadlock cycles that are then fed to the DPOR engine to guide the testing process (discarding non-deadlock executions) [4]. Let us note that other actor-based languages with similar features could be handled by SYCO just by providing a compiler to the SYCO IR.



The web interface of SYCO is available at [costa.ls.fi.upm.es/syco](http://costa.ls.fi.upm.es/syco). In App. A we detail how to use it with screenshots. Essentially, once the input program is ready, either selected from the available library of ABS programs or supplied by the user, a set of parameters are provided (or just left with by-default values), the SYCO engine is run and the output is obtained.

**Parameters.** The following parameters can be set:

- *Partial-order reduction*: It enables/disables POR.
- *Dependency over-approximation*: In case POR is applied, a central operation is the detection of independent tasks, which has to be over-approximated. SYCO includes the over-approximation of [11] which considers as dependent tasks those in the same actor, and, also, the enhancement of [2] for actors with local memory, which looks at field accesses within the involved tasks and considers as dependent only tasks belonging to the same actor and accessing at least a common field.
- *Deadlock-guided testing*: If this parameter is selected, the testing process is guided with the cycles inferred by DECO towards deadlocks, discarding non-deadlock executions, with the corresponding state space reduction. This is useful in the context of deadlock detection and debugging.

**Output.** As a result, SYCO outputs a set of executions. For each one, SYCO shows the output state and the sequence of tasks/interleavings and concrete instructions of the execution (highlighting the source code). Also, it allows showing a sequence diagram from which it can be observed the task/object executing and the asynchronous calls made (with arrows from caller to callee) at each time of the simulation, the waiting and blocking dependencies, the deadlock cycles, etc. See App. A for details. SYCO produces 6 executions for the running example with POR disabled. That covers all possible task interleavings that may occur. SYCO reports that 2 executions are deadlock executions corresponding to sequences  $\text{main} \rightarrow \text{register} \rightarrow \text{work}$  and  $\text{main} \rightarrow \text{work} \rightarrow \text{register}$ . Those correspond to scenario (iii) at the end of Sect. 1. Within the remaining 4 executions, two of them correspond to scenario (i) and the other two to scenario (ii). According to POR theory [2, 11], the remaining

4 executions can be grouped in two different equivalence classes, therefore 2 executions are redundant and only two different results are obtained. When POR is enabled, SYCO produces these 4 executions, the two deadlock executions, and, the executions corresponding to scenarios (i) and (ii).

### 3. Discussion and Related Work

We have presented a systematic tester for an actor-based concurrency model which incorporates state-of-the-art POR methods. The tool can be used online through its web interface and provides information about all possible (non-redundant) behaviors that the input concurrent program may have, including trace highlighting and detailed sequence diagrams. It also has support for deadlock detection and debugging, incorporating novel techniques for deadlock-guided testing [4] in which an external deadlock analyzer [6] is embedded. We claim that the tool is very useful for testing and debugging models of concurrent systems.

Several related tools exist, being the most relevant Microsoft's *CHES* [9] for .NET, *Concuerror* [5] for Erlang and *Basset* [10] for *ActorFoundry*. All of them incorporate state-of-the-art POR techniques. The most advanced in this sense is *Concuerror* which is equipped with the most recent *Optimal* DPOR algorithm [1]. Also, *Concuerror* is the only one providing graphical output similar to our sequence diagrams. None of them provides a web interface. Many other related tools exist in the context of *model-checking* that are left out of this comparison.

As regards future work, we are currently studying the most advanced POR techniques of [1] and the possibility of adapting them to our context. Also, we are in the process of incorporating the symbolic execution engine of [3] so that SYCO also allows performing static testing. Finally, we plan to work on other visualizations of the output like execution trees and step-by-step timelines.

### 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] 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.
- [3] 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. Available at <http://costa.ls.fi.upm.es/apet>.
- [4] E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. Technical report, 2015. Available at: <http://costa.ls.fi.upm.es/papers/costa/AlbertGI15.pdf>.
- [5] S. Aronis and K. Sagonas. Concuerror: Systematic concurrency testing of Erlang programs.
- [6] A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS 7892, pages 273–288. Springer, 2013.
- [7] S. Genaim and J. Doménech. The EasyInterface Framework, 2015.
- [8] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMOODS'10 (Revised Papers)*, LNCS 6957, pp. 142–164. Springer, 2012.
- [9] M. Musuvathi and S. Qadeer. Concurrency Unit Testing with CHES. Technical Report MSR-TR-2008-04, Microsoft Research, January.
- [10] D. Marinov, G. Agha, S. Lauterburg, R. K. Karmani. Basset: A Tool for Systematic Testing of Actor Programs. In *Proceedings of FSE 2010*.
- [11] 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.



## A. Tool Demonstration

We start describing the basic elements of the web interface of SYCO. We then show how to test the running example, analyzing the reported information and demonstrating the tool’s potential to help the programmer understanding the deadlock situations and finding a solution to avoid them. We finally show how to use the SYCO parameters.

### A.1 The SYCO Web Interface

Figure 1 shows the main window of the SYCO web interface provided by EasyInterface [7]. It comprises 5 main components: (1) the *file manager*, that contains a list of predefined examples as well as the files uploaded by the user; (2) the *code area*, where the user can edit the selected program; (3) the *outline view*, which includes an outline (list of classes, methods, etc.) of the selected file and module; (4) the *tools bar*, which includes several buttons to execute the main actions; and (5) the *console area* where the results and information of the execution are printed.

Let us go to the file manager, located at the left-hand side, and open the folder *syco\_examples*. The subfolder *deadlocks* contains the code of our running example, *cc2016.abs*. If we click over it, the code appears at the code area. Later, we will modify the code to obtain different results, specifically to remove the deadlock situations. By now, let us refresh the outline. This updates the right-hand side with the class and module information of our running example.

In the tools bar, next to the *Apply button*, there is a combo-box that contains all the applications available in this instance of EasyInterface. In this case, the only available application is SYCO, but let us note that we could add other tools that may co-exist with SYCO, e.g., the DECO deadlock analyzer [6]. With SYCO selected, we press on the Apply button. This produces the invocation of the SYCO engine with by-default parameters. Later, we will see how to configure the SYCO parameters. The main output information is then shown at the console area (see Figure 1).

### A.2 Analyzing the SYCO Output

SYCO first prints the number of executions explored in the systematic testing of the program under test. Then, it prints the output state and the execution trace for each execution. The output state (in blue color) contains all the objects created during the execution. Each object is represented as a term with three arguments: the object identifier, the object type or class, and the final values of the object fields. The execution trace (in red color) shows, for each time or macro-step of the execution, the object and task executing at this time. If we click over one time of the trace, the source program lines which are exercised at this time are highlighted (in yellow color) at the code area. This is shown in Figure 1 where the first time of the trace has been clicked.

To see the sequence diagram of a concrete execution, we press over “Click here to see the sequence diagram” which is found next to the execution number in the console view. A dialog box containing the sequence diagram shows up. Figure 2 (right) shows the sequence diagram of the first execution for our running example, which correspond to scenario (i), i.e., we get the expected result (see end of Sect. 1). At the left-hand side, a timeline is shown with the times of the execution, in both cases 7 times (0-6). Each vertical cluster corresponds to the activities performed by each object, and each node corresponds to the task executing at the corresponding object in the corresponding time. Objects are of the form *class\_id*, where *class* is the object type and *id* a unique object identifier. Tasks are of the form *id:method(pp)*, where *id* is a unique task identifier, *method* is the name of the method and *pp* the program point of the method from which the execution starts. Nodes also indicate why the execution of the associated task stopped. Nodes in

green color labeled with *return* correspond to tasks that have reach their return instruction; nodes in orange color label-led with *waiting for taskId* are tasks which have been suspended waiting for task *taskId*; and nodes in red color labeled with *blocked for taskId* are tasks which block the object waiting for task *taskId*. Finally, arrows from nodes to clusters indicate asynchronous calls or object creations.

The first sequence diagram can be understood as follows: During time 0, the object *main\_2* creates objects *DBimp\_0* and *WorkerImp\_1*, and spawns tasks *0:register* and *2:work*, respectively. Then, *main\_2* finishes its execution reaching its return statement (green color). During time 1, *DBimp\_0* tries to register *WorkerImp\_1* as its client. It executes lines 19, 20 and 21, where it gets blocked waiting for *WorkerImp\_1* to execute task *1:ping*. This node is in red color, since its execution finishes due to a blocking waiting. During times 2 and 3, *WorkerImp\_1* executes *ping* (in green color) and it gets blocked (in red color) waiting for *3:getData* to be executed by *DBimp\_0*. During times 4 and 5, *DBimp\_0* resumes the execution of method *register* from line 21, registers *DBimp\_0* as its client, and returns the stored data which is finally received by *WorkerImp\_1* at time 6. According to POR theory, the two first executions are redundant, see Figures 2 (right) and 3 (top), since the only difference between them is the order of tasks *0:register(21)* and *2:work(35)* which are independent. We are getting both because no POR is applied by default.

Let us now focus on the fourth execution. The output state shows that the worker *data* field ends with *null* value, which is not the expected value, corresponding to scenario (ii). Let us look at the associated sequence diagram which is shown at the bottom of Fig. 3. It can be observed that *DBimp\_0* is executing task *3:getData* before registering the worker as its client. Hence, the *getData*’s condition evaluates to *false* and, as a result, it returns *null*.

Looking at the sequence diagram of the third execution (see middle of Figure 3, we can observe a deadlock situation, since both *DBimp\_0* and *WorkerImp\_1* are blocked and, as we can see, they are squared in red color. During time 1, *DBimp\_0* gets blocked waiting for *WorkerImp\_1* to execute task *1:ping*. During the next time, instead of letting *WorkerImp\_1* executing, it gets blocked waiting for *DBimp\_0* to execute *getData*. Therefore, none of them can make any progress. Both tasks are shown by means of red solid edges to indicate that these are the ones responsible for the deadlock.

The main problem with this implementation, is that we are not in control of when the database registers the worker as its client. A possible solution to overcome these unexpected situations could be to write the main as follows:

```

27 { \main block
28   DB db = new DB();
29   Worker w = new Worker();
30   Fut<Unit> f = db!register(w);
31   await f?;
32   w!work(db);
33 }
```

If we execute SYCO again, we only get one execution whose output state is the expected one, i.e., we only produce scenario (i).

### A.3 The SYCO Parameters

If we press over the “Settings” button at the tool bar, the EasyInterface parameters window shows up (see Figure 2), which allows configuring the available SYCO parameters of the available applications. The first SYCO parameter is *Partial-order reduction* whose by-default value is *No*. Hence, by default, SYCO explores all possible executions. In our running example, those are the obtained 6 executions. As already noted, the first two executions are redundant since the only difference between them is the order of tasks *0:register(21)* and *2:work(35)* which are independent. If we

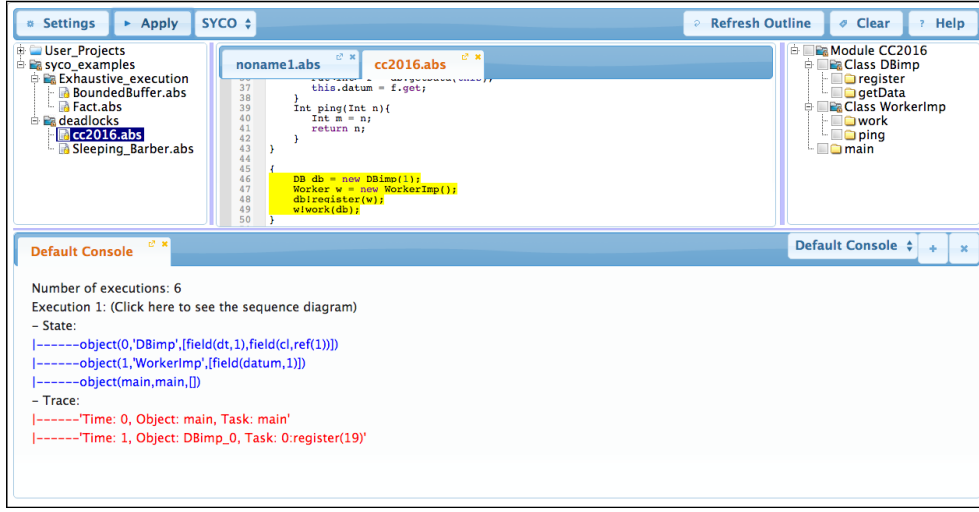


Figure 1. Main window of the SYCO web interface

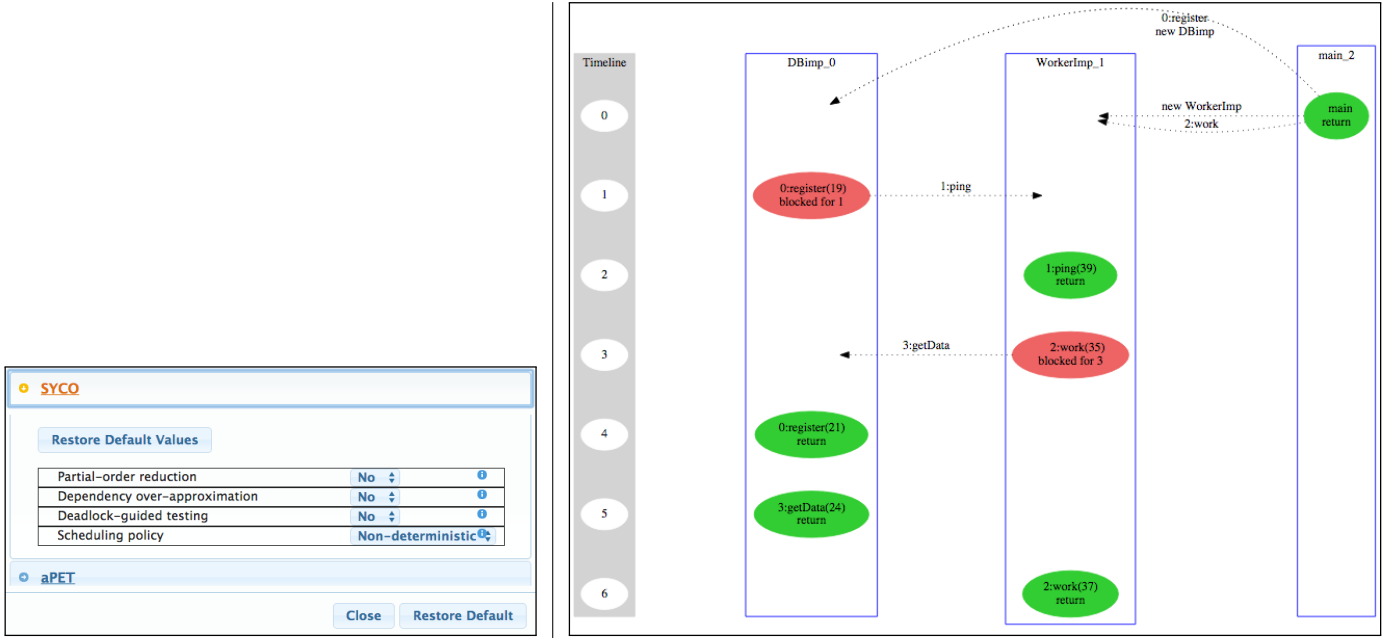


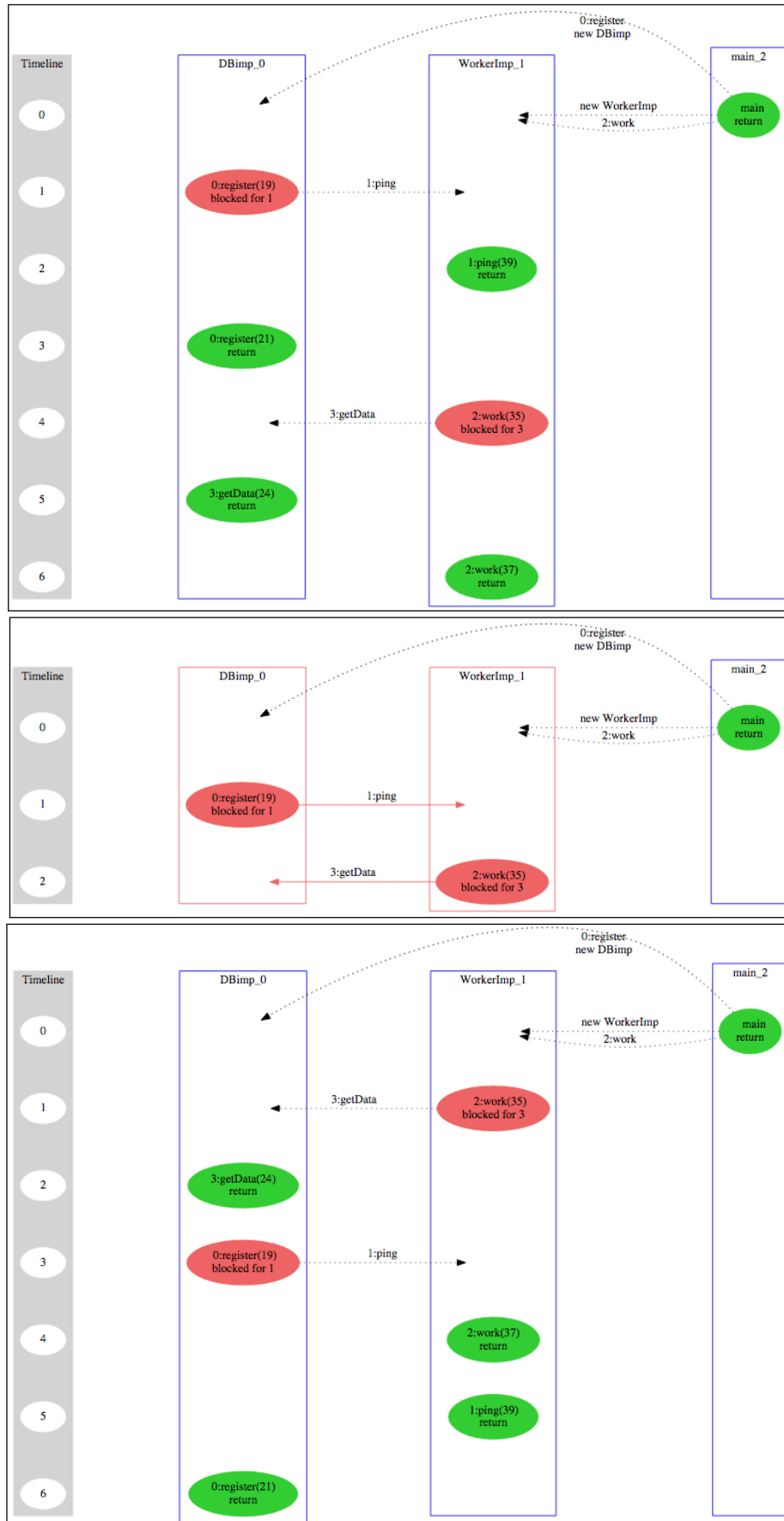
Figure 2. The SYCO parameters window and 1st sequence diagram

select value “yes” for this parameter, then SYCO applies POR, obtaining just 4 executions, one for scenario (i), one for scenario (ii) and the two possible deadlocks of scenario (iii).

The parameter *Deadlock-guided testing* activates the guided search towards deadlocks. This is useful in case we are interested in proving deadlock freedom or finding a concrete deadlock trace. If we select this option, we then obtain just the two deadlock traces among the 6 possible executions. It is important to highlight that this is not a post-process. SYCO invokes the DECO deadlock analyzer and used its results to detect when a deadlock cycle is unfeasible in an execution stopping it, gaining time and reducing in part the combinatorial explosion. In the running example, deadlock situations only happen when both *DBimp\_0* and *WorkerImp\_1* get blocked. As we can see in Figure 2 (right), *WorkerImp\_1* executes task *1:ping*, so *DBimp\_0* will eventually resume task *0:register*. From this point on, the deadlock cycle obtained by DECO is unfeasible and SYCO stops this execution at time 2.

The effect of parameter *Dependency over-approximation* cannot be observed with our running example in its present form. Let us modify the code by adding instructions **await** *f*? before the instructions *v = f.get*. This allows avoiding deadlocks and having a greater parallelism than the one in the previous modification. With this modification, if we run SYCO with by-default parameters, we obtain 20 executions. Activating the option *Partial-order reduction* downsizes it up to 8 executions.

Nevertheless, we can observe that some of them lead to the same final state, even though these executions have different partial-orders. This kind of redundancy is based on the concept of task independence. Two tasks are independent when the order in which they are executed does not affect the final result. In this new example, we have two independent tasks in *WorkerImp\_1*: *ping* and *work.await* (*work.await* being the resumption task once *getData* has been executed by *DBimp\_0*). The 2nd parameter over-approximates the independent tasks and allows the explored executions to be reduced up to 6.



**Figure 3.** 2nd, 3rd and 4th sequence diagrams

## Appendix F

Article “*Combining Static Analysis and Testing for Deadlock Detection*”, [2]

# Combining Static Analysis and Testing for Deadlock Detection

## Technical Report (including Proofs)

Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel

Complutense University of Madrid (UCM), Spain

**Abstract.** Static deadlock analyzers might be able to verify the absence of deadlock, but when they detect a potential deadlock cycle, they provide little (or even none) information on their output. Due to the complex flow of concurrent programs, the user might not be able to find the source of the anomalous behaviour from the abstract information computed by static analysis. This paper proposes the combined use of static analysis and testing for effective deadlock detection in asynchronous programs. Our main contributions are: (1) We present an enhanced semantics which allows an early detection of deadlocks during testing and that can give to the user a precise description of the deadlock trace. (2) We combine our testing framework with the abstract descriptions of potential deadlock cycles computed by an existing static deadlock analyzer. Namely, such descriptions are used by our enhanced semantics to guide the execution towards the potential deadlock paths (while other paths are pruned). When the program features a deadlock, our combined use of static analysis and testing provides an effective technique to find deadlock traces. While if the program does not have deadlock, but the analyzer inaccurately spotted it, we might be able to prove deadlock freedom.

## 1 Introduction

In concurrent programs, *deadlock* is one of the most common programming errors and, thus, a main goal of verification and testing tools for concurrent programs is, respectively, proving deadlock freedom and *deadlock detection*. We consider an *asynchronous* language which allows spawning asynchronous tasks at distributed locations, and has two operations for blocking and non-blocking synchronization with the termination of asynchronous tasks. In this setting, in order to detect deadlocks, all possible *interleavings* among tasks executing at the distributed locations must be considered. Basically, each time that the processor can be released, any of the available tasks can start its execution, and all combinations among the tasks must be tried, as any of them might lead to deadlock.

Static analysis and testing are two different ways of detecting deadlocks that often complement each other and thus it seems quite natural to combine them. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks, months or years after releasing the application. This aspect of static analysis is especially important

in security assurance, because security attacks try to exercise an application in unpredictable and untested ways. However, when a deadlock is found, state-of-the-art analysis tools [11, 12, 9, 17] provide little (and often none) information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and locations), it is essential to know the task interleavings that have occurred and the locations involved in the deadlock, i.e., provide a concrete *deadlock trace* that allows the programmer to identify and fix the problem. In contrast, testing consists in executing the application for concrete input values. The primary advantage of testing for deadlock detection is that it can provide the deadlock trace with all information that the user needs in order to fix the problem. There are two shortcomings though: (1) Since not all inputs can be tried, there is no guarantee of deadlock freedom. (2) Although recent research tries to avoid redundant exploration as much as possible [10, 20, 8, 1, 4, 1], the search space (without redundancies) can be huge. This is a threaten to the application of testing in concurrent programming.

This paper proposes a seamless combination of static analysis and testing for effective deadlock detection as follows: an existing static deadlock analysis [11] is first used to obtain *abstract* descriptions of potential deadlock cycles which are then used to guide a testing tool in order to find associated deadlock traces (or discard them). Technically, the main contributions of the paper are:

1. We extend a standard semantics for asynchronous programs with information about the task interleavings made, and the status of tasks (i.e., awaiting, blocked, or finished). The extended semantics will allow us: (1) to provide deadlock traces when a deadlock is found, (2) an early detection of deadlock states during execution and (3) its combined use with static analysis.
2. We provide a formal characterization of *deadlock state* which can be checked along the execution, and allows us to early detect deadlocks even in complex situations in which there are one or several locations that keep on executing (maybe even go into an infinite computation) while, due to blocking call chains in other locations, the execution will eventually lead to deadlock.
3. We present a new methodology to detect deadlocks which combines testing and static analysis as follows: the deadlock cycles inferred by static analysis are used by our extended semantics to guide the testing process towards paths that might lead to a deadlock cycle and discard deadlock-free paths.
4. The implementation in the aPET system [5], the definition of several deadlock-based testing criteria, and a thorough experimental evaluation. Our experiments show that we can find deadlock traces for the potential deadlock cycles with a significant reduction of the required state exploration.

## 2 Asynchronous Programs: Syntax and Semantics

We consider a distributed programming model with explicit locations. Each location represents a processor with a procedure stack and an unordered buffer of pending tasks. Initially all processors are idle. When an idle processor's task buffer is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the buffers of any

$$\begin{array}{c}
\text{(MSTEP)} \quad \text{selectLoc}(S) = \text{loc}(o, \perp, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, \text{selectTask}(o) = \text{tsk}(tk, m, l, s), \\
\frac{S \diamond \rho_0 \xrightarrow{o \cdot tk}^* S' \diamond \rho}{S \xrightarrow{o \cdot tk} S'} \\
\text{(NEWLOC)} \quad \frac{tk = \text{tsk}(tk, m, l, x = \text{new } D; s), \text{fresh}(o'), h' = \text{newheap}(D), l' = l[x \rightarrow o']}{\text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \rho_0 \rightsquigarrow \text{loc}(o, tk, h, \mathcal{Q} \cup \{\text{tsk}(tk, m, l', s)\}) \cdot \text{loc}(o', \perp, h', \{\}) \diamond \rho_0} \\
\text{(ASYNC)} \quad \frac{tk = \text{tsk}(tk, m, l, y = x!m_1(\bar{z}); s), l(x) = o_1, \text{fresh}(tk_1), l_1 = \text{buildLocals}(\bar{z}, m_1, l)}{\text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot \text{loc}(o_1, \_, \_, \mathcal{Q}') \diamond \rho_0 \rightsquigarrow \text{loc}(o, tk, h, \mathcal{Q} \cup \{\text{tsk}(tk, m, l, s)\}) \cdot \text{loc}(o_1, \_, \_, \mathcal{Q}' \cup \{\text{tsk}(tk_1, m_1, l_1, \text{body}(m_1))\}) \cdot \text{fut}(y, o_1, tk_1, \text{ini}(m_1)) \diamond \rho_0} \\
\text{(RETURN)} \quad \frac{tk = \text{tsk}(tk, m, l, \text{return}; s), \rho_1 = \text{return}}{\text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \rho_0 \rightsquigarrow \text{loc}(o, \perp, h, \mathcal{Q} \cup \{\text{tsk}(tk, m, l, \epsilon)\}) \diamond \rho_1} \\
\text{(AWAIT1)} \quad \frac{tk = \text{tsk}(tk, m, l, y.\text{await}; s), \text{tsk}(tk_1, \_, \_, s_1) \in \mathbf{Ob}, s_1 = \epsilon}{\text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_0 \rightsquigarrow \text{loc}(o, tk, h, \mathcal{Q} \cup \{\text{tsk}(tk, m, l, s)\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_0} \\
\text{(AWAIT2)} \quad \frac{tk = \text{tsk}(tk, m, l, pp:y.\text{await}; s), \text{tsk}(tk_1, \_, \_, s_1) \in \mathbf{Ob}, s_1 \neq \epsilon, \rho_1 = pp : y.\text{await}}{\text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_0 \rightsquigarrow \text{loc}(o, \perp, h, \mathcal{Q} \cup \{tk\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_1} \\
\text{(BLOCK1)} \quad \frac{tk = \text{tsk}(tk, m, l, y.\text{block}; s), \text{tsk}(tk_1, \_, \_, s_1) \in \mathbf{Ob}, s_1 = \epsilon}{\text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_0 \rightsquigarrow \text{loc}(o, tk, h, \mathcal{Q} \cup \{\text{tsk}(tk, m, l, s)\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_0} \\
\text{(BLOCK2)} \quad \frac{tk = \text{tsk}(tk, m, l, pp:y.\text{block}; s), \text{tsk}(tk_1, \_, \_, s_1) \in \mathbf{Ob}, s_1 \neq \epsilon, \rho_1 = pp:y.\text{block}}{\text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_0 \rightsquigarrow \text{loc}(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot \text{fut}(y, \_, tk_1, \_) \diamond \rho_1}
\end{array}$$

Fig. 1. Semantics of Asynchronous Programs

processor, including its own, and synchronize with the termination of tasks. The language uses *future variables* to check if the execution of an asynchronous task has finished. An asynchronous call  $m(\bar{z})$  spawned at location  $x$  is associated with a future variable  $f$  as follows  $f = x ! m(\bar{z})$ . Instructions  $f.\text{block}$  and  $f.\text{await}$  allow, respectively, blocking and non-blocking synchronization with the termination of  $m$ . When a task completes, or when it is awaiting with a non-blocking `await` for a task that has not finished yet, its processor becomes idle again, chooses the next pending task, and so on. The number of distributed locations need not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to a *concurrent object* and can be dynamically created using the instruction `new`. The program consists of a set of methods of the form  $M ::= T \ m(\bar{T} \ \bar{x})\{s\}$ , where statements  $s$  take the form  $s ::= s \mid x = e \mid \text{if } e \text{ then } s \text{ else } s \mid \text{while } e \text{ do } s \mid \text{return} \mid b = \text{new} \mid f = x ! m(\bar{z}) \mid f.\text{await} \mid f.\text{block}$ . For the sake of generality, the syntax of expressions  $e$  and types  $T$  is left open.

Fig. 1 presents the semantics of the language. The information about  $\rho$  in bold font is part of the extensions for testing in Sec. 4 and should be ignored by now. A *state* or *configuration* is a set of locations and future variables  $o_0 \cdots o_n \cdot \text{fut}_0 \cdots \text{fut}_m$ . A *location* is a term  $\text{loc}(o, tk, h, \mathcal{Q})$  where  $o$  is the location identifier,  $tk$  is the identifier of the *active task* that holds the location's lock or  $\perp$  if the location's lock is free,  $h$  is its local heap, and  $\mathcal{Q}$  is the set of tasks in the location. A *future variable* is a term  $\text{fut}(id, o, tk, m)$  where  $id$  is a unique future variable

identifier,  $o$  is the location identifier that executes the task  $tk$  awaiting for the future, and  $m$  is the initial program point of  $tk$ . A *task* is a term  $tsk(tk, m, l, s)$  where  $tk$  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  $\epsilon$  if the task has terminated. We assume that the execution starts from a **main** method without parameters. The initial state is  $St = \{loc(0, 0, \perp, \{tsk(0, main, l, body(main))\})\}$  with an initial location with identifier 0 executing task 0. Here,  $l$  maps local variables to their initial values (**null** in case of reference variables) and  $\perp$  is the empty heap.  $body(m)$  is the sequence of instructions in method  $m$ , and we can know the program point  $pp$  where an instruction  $s$  is in the program as follows  $pp:s$ .

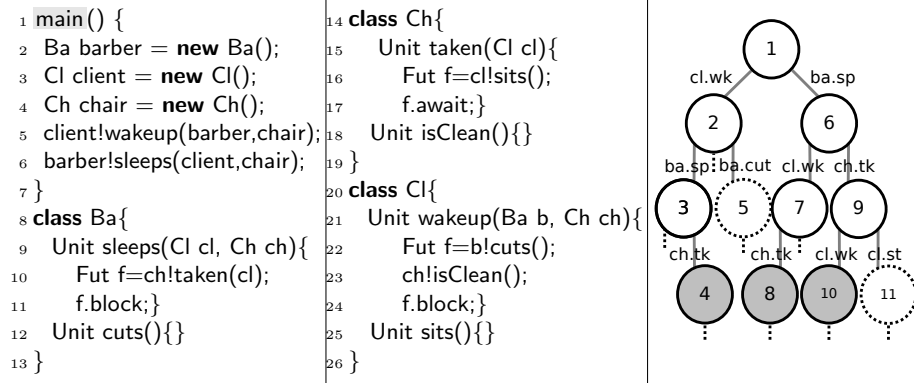
As locations do not share their states, the semantics can be presented as a macro-step semantics [19] (defined by means of the transition “ $\longrightarrow$ ”) in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to an **await** or **return** instruction. In this case, we apply rule MSTEP to select an available task from a location, namely we apply the function  $selectLoc(S)$  to select non-deterministically one *active* location in the state (i.e., a location with a non-empty queue) and  $selectTask(o)$  to select non-deterministically one task of  $o$ ’s queue. The transition  $\leadsto$  defines the evaluation within a given location. NEWLOC creates a new location without tasks, with a fresh identifier and heap. ASYNC spawns a new task (the initial state is created by  $buildLocals$ ) with a fresh task identifier  $tk_1$ , and it adds a new future to the state.  $ini(m)$  refers to the first program point of method  $m$ . We assume  $o \neq o_1$ , but the case  $o = o_1$  is analogous, the new task  $tk_1$  is added to  $\mathcal{Q}$  of  $o$ . The rules for sequential execution are standard and are thus omitted. AWAIT1: If the future variable we are awaiting for points to a finished task, the await can be completed. The finished task  $t_1$  is only looked up but it does not disappear from the state as its status may be needed later on. AWAIT2: Otherwise, the task yields the lock so that any other task of the same location can take it. RETURN: When **return** is executed, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction  $\epsilon$ ). BLOCK2: A *y.block* instruction waits for the future variable but without yielding the lock. Then, when the future is ready, BLOCK1 allows continuing the execution.

In what follows, a *derivation* or *execution*  $E \equiv St_0 \longrightarrow \dots \longrightarrow St_n$  is a sequence of macro-steps (applications of rule MSTEP). The derivation is *complete* if  $St_0$  is the initial state and  $\nexists St_{n+1} \neq St_n$  such that  $St_n \longrightarrow St_{n+1}$ . Since the execution is non-deterministic, multiple derivations are possible from a state. Given a state  $St$ ,  $exec(St)$  denotes the set of all possible derivations starting at  $St$ . We sometimes label transitions with  $o \cdot tk$ , the name of the location  $o$  and task  $tk$  selected (in rule MSTEP) or evaluated in the step (in the transition  $\leadsto$ ).

### 3 Motivating Example

Our running example is a simple version of the classical sleeping barber problem where a barber sleeps until a client arrives and takes a chair, and the client wakes up the barber to get a haircut. Our implementation has a **main** method showed





**Fig. 2.** Classical Sleeping Barber Problem (left) and Execution Tree (right)

to the left and three classes **Ba**, **Ch** and **Cl** implementing the barber, chair and client, respectively. The **main** creates three locations **barber**, **client** and **chair** and spawns two asynchronous tasks to start the **wakeup** task in the client and **sleeps** in the barber, both tasks can run in parallel. The execution of **sleeps** spawns an asynchronous task on the **chair** to represent the fact that the client takes the chair, and then blocks at L11 (L11 for short) until the chair is taken. The task **taken** first adds the task **sits** on the client, and then **awaits** on its termination at L17 without blocking, so that another task on the location **chair** can execute. On the other hand, the execution of **wakeup** in the client spawns an asynchronous task **cuts** on the barber and one on the chair, **isClean**, to check if the chair is clean. The execution of the client blocks until **cuts** has finished. We assume that all methods have an implicit return at the end.

Fig. 2 summarizes the execution tree of the **main** by showing some of the macro-steps taken. Derivations that contain a dotted node are not deadlock, while those with a gray node are deadlock. A main motivation of our work is to detect as early as possible that the dotted derivations will not lead us to deadlock and prune them. Let us see two selected derivations in detail. In the derivation ending at node 5, the first macro-step executes **cl.wakeup** and then **b.cuts**. Now, it is clear that the location **cl** will not deadlock, since the **block** at L24 will succeed and the other two locations will be also able to complete their tasks, namely the **await** at L17 of location **ch** can finish because the client is certainly not blocked, and also the **block** at L11 will succeed because the task in **taken** will eventually finish as its location is not blocked. However, in the branch of node 4, we first select **wakeup** (and block client), then we select **sleeps** (and block barber), and then select **taken** that will remain in the **await** at L17 and will never succeed since it is awaiting for the termination of a task of a blocked location. Thus, we certainly have a deadlock. Let us outline five states of this derivation:

$$\begin{aligned}
St_0 &\equiv loc(ini, ..) \cdot loc(cl, .., \{tsk(1, wk, ..)\}) \cdot loc(ba, .., \{tsk(2, sp, ..)\}) \cdot loc(ch, ..) \xrightarrow{cl, 1} \\
St_1 &\equiv loc(cl, .., \{tsk(1, wk, f_0.block)\}) \cdot loc(ba, .., \{tsk(3, cut, ..), ..\}) \cdot fut(f_0, ba, 3, 12) \cdot .. \xrightarrow{ba, 2} \\
St_2 &\equiv loc(ba, .., \{tsk(2, sp, f_1.block)\}) \cdot loc(ch, .., \{tsk(5, tk, ..), ..\}) \cdot fut(f_1, ch, 5, 15) \cdot .. \xrightarrow{ch, 5} \\
St_3 &\equiv loc(ch, .., \{tsk(5, tk, f_2.await), ..\}) \cdot loc(cl, .., \{tsk(6, st, ..), ..\}) \cdot fut(f_2, cl, 6, 25) \cdot .. \\
&\xrightarrow{ch, 4} St_4 \equiv loc(ch, .., \{tsk(4, isClean, return), ..\}) \cdot ..
\end{aligned}$$

$$\begin{array}{c}
\text{(MSTEP2)} \quad \frac{\begin{array}{c} \text{selectLoc}(S) = \text{loc}(o, \perp, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, \text{selectTask}(o) = \text{tsk}(tk, m, l, pp : s), \\ \text{check}_{\mathfrak{c}}(S, \text{table}), S \diamond \rho_0 \xrightarrow{o \cdot tk}^* S' \diamond \rho, S \neq S', \text{not}(\text{deadlock}(S')) \\ \text{clock}(n), \text{table}' = \text{table} \cup t_{o, tk, pp} \mapsto \langle n, \rho \rangle \end{array}}{(S, \text{table}) \xrightarrow{o \cdot tk} (S', \text{table}')}
\end{array}$$

**Fig. 3.** MSTEP2 rule for combined testing and analysis

The first state is obtained after executing the `main` where we have the initial location `ini`, three locations created at L3, L2 and L4, and two tasks at L5 and L6 added to the queues. Note that each location and task is assigned a unique identifier (we use numbers as identifiers for tasks and short names as identifiers for locations). In the next state, the task `wakeup` has been selected and fully executed (we have shortened the name of the methods, e.g., `wk` for `wakeup`). Observe at  $St_1$  the addition of the future variable created at L22. In  $St_2$  we have executed task `sleeps` in the barber and added a new future term. In  $St_3$  we execute task `taken` in the chair (this state is already deadlock as we will see in Sec. 4.2), however location `chair` can keep on executing an available task `isClean`. From now on, we use the location and task names instead of numeric identifiers for clarity.

## 4 Testing for Deadlock Detection

The goal of this section is to present a framework for early detection of deadlocks during testing. This is done by enhancing the standard semantics for asynchronous programs with information which allows us to easily detect *dependencies* among tasks, i.e., when a task is awaiting for the termination of another one. These dependencies are necessary to detect in a second step *deadlock states*.

### 4.1 An Enhanced Semantics for Deadlock Detection

In the following we define the *interleavings table* whose role is twofold: (1) It stores all decisions about task interleavings made during the execution. This way, at the end of a concrete execution, the exact ordering of the performed macro-steps can be observed. (2) It will be used to detect deadlocks as early as possible, and, also to detect states from which a deadlock cannot occur, therefore allowing to prune the execution tree when we are looking for deadlocks. The interleavings table is a mapping with entries of the form  $t_{id_o, id_t, pp} \mapsto \langle n, \rho \rangle$ , where:

- $t_{id_o, id_t, pp}$  is a *macro-step identifier*, or *time identifier*, that includes: the identifiers of the location  $id_o$  and task  $id_t$  that have been selected in the macro-step, and the program point  $pp$  of the first instruction that will be executed;
- $n$  is a (non-negative) integer representing the time when the macro-step starts executing;
- $\rho$  is the status of the task after the macro-step and it can take three values as it can be seen in Fig. 1: **block** or **await** when executing these instructions on a future variable that is not ready (we also annotate in  $\rho$  the information on the associated future); **return** that allows us to know that the task finished.

We use a function **clock**( $n$ ) to represent a clock that starts at 0, is increased by one in every execution of **clock**, and returns the current value  $n$ . The initial

entry is  $t_{0,0,1} \mapsto \langle 0, \rho_0 \rangle$ , being 0 the identifier for the initial location and task, and 1 the first program point of *main*. The clock also assigns the value 0 as the first element in the tuple and a fresh variable in the second element  $\rho_0$ . The next macro-step will be assigned clock value 1, next 2, and so on. As notation, we define the relation  $t \in \text{table}$  if there exists an entry  $t \mapsto \langle n, \rho \rangle \in \text{table}$ , and the function  $\text{status}(t, \text{table})$  which returns the status  $\rho_t$  such that  $t \mapsto \langle n, \rho_t \rangle \in \text{table}$ . The semantics is extended by changing rule MSTEP as in Fig. 3. The function **deadlock** will be defined in Thm. 1 to stop derivations as soon as deadlock is detected. Function  $\text{check}_{\mathcal{E}}$  should be ignored by now, it will be defined in Sec. 5.2. Essentially, there are two new aspects: (1) The state is extended with the status  $\rho$ , namely all rules include a status  $\rho$  attached to the state using the symbol  $\diamond$ . The status is showed in bold font in Fig. 1 and can get a value in rules **block2**, **await2** and **return**. The initial value  $\rho_0$  is a fresh variable. (2) The state for the macrostep is extended with the interleavings table *table*, and a new entry  $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$  is added to *table* in every macrostep if there has been progress in the execution, i.e.,  $S' \neq S$ , being  $n$  the current clock time.

*Example 1.* The interleavings table below (left) is computed for the derivation in Sec. 3. It has as many entries as macro-steps in the derivation. We can observe that subsequent time values are assigned to each time identifier so that we can then know the order of execution. The right column shows the future variables in the state that store the location and task they are bound to.

$St_0$	$t_{ini,main,1} \mapsto \langle 1, \text{return} \rangle$	$\emptyset$
$St_1$	$t_{cl,wakeup,21} \mapsto \langle 2, 24:f_0.\text{block} \rangle$	$\text{fut}(f_0, ba, cuts, 12)$
$St_2$	$t_{ba,sleeps,9} \mapsto \langle 3, 11:f_1.\text{block} \rangle$	$\text{fut}(f_1, ch, taken, 15)$
$St_3$	$t_{ch,taken,15} \mapsto \langle 4, 17:f_2.\text{await} \rangle$	$\text{fut}(f_2, cl, sits, 25)$

## 4.2 Formal Characterization of Deadlock State

Our semantics can easily be extended to detect deadlock just by redefining function *selectLoc* so that only locations that can proceed are selected. If, at a given state, no location is selected but there is at least a location with a non-empty queue then there is a deadlock. However, deadlocks can be detected earlier. We present the notion of *deadlock state* which characterizes states that contain a *deadlock chain* in which one or more tasks are waiting for each other termination and none of them can make any progress. Note that, from a deadlock state, there might be tasks that keep on progressing until the deadlock is finally made explicit. Even more, if one of those tasks runs into an infinite loop, the deadlock will not be captured using this naive extension. The early detection of deadlocks is crucial to reduce state exploration as our experiments show in Sec. 6.

We first introduce the auxiliary notion of *waiting interval* which captures the period in which a task is waiting for another one to terminate. In particular, it is defined as a tuple  $(t_{stop}, t_{async}, t_{resume})$  where  $t_{stop}$  is the macro-step at which the location stops executing a task due to some block/await instruction,  $t_{async}$  is the macro-step at which the task that is being awaited is selected for execution, and,  $t_{resume}$  is the macro-step at which the task will resume its execution.  $t_{stop}$ ,  $t_{async}$  and  $t_{resume}$  are time identifiers as defined in Sec. 4.1.  $t_{resume}$  will also be written

as  $next(t_{stop})$ . When the task stops at  $t_{stop}$  due to a **block** instruction, we call it *blocking interval*, as the location remains blocked between  $t_{stop}$  and  $next(t_{stop})$  until the awaited task, selected in  $t_{async}$ , has already finished. The execution of a task can have several points at which macro-steps are performed (e.g., if it contains several **await** or **block** the processor may be lost several times). For this reason, we define the set of successor macro-steps of the same task from a macro-step:  $suc(t_{o,tk,pp_0}, table) = \{t_{o,tk,pp_i} : t_{o,tk,pp_i} \in table, t_{o,tk,pp_i} \geq t_{o,tk,pp_0}\}$ .

**Definition 1 (Waiting/Blocking Intervals).** Let  $St = (S, table)$  be a state,  $I = (t_{stop}, t_{async}, t_{resume})$  is a waiting interval of  $St$ , written as  $I \in St$ , iff:

1.  $\exists t_{stop} = t_{o,tk_0,pp_0} \in table, \rho_{stop} = status(t_{stop}) \in \{pp_1 : x.await, pp_1:x.block\}$ ,
2.  $t_{resume} \equiv t_{o,tk_0,pp_1}, fut(x, o_x, tk_x, pp(M)) \in S$ ,
3.  $t_{async} \equiv t_{o_x,tk_x,pp(M)}, \nexists t \in suc(t_{async}, table)$  with  $status(t) = return$ .

If  $\rho_{stop} = x.block$ , then  $I$  is blocking.

In condition 3, we can see that if the task starting at  $t_{async}$  has finished, then it is not a waiting interval. This is known by checking that this task has not reached return, i.e.,  $\nexists t \in suc(t_{async}, table)$  such that  $status(t) = return$ . In condition 1, we see that in  $\rho_{stop}$  we have the name of the future we are awaiting (whose corresponding information is stored in  $fut$ , condition 2). In order to define  $t_{resume}$  in condition 2, we search for the same task  $tk_0$  and same location  $o$  that executes the task starting at program point  $pp_1$  of the **await/block**, since this is the point that the macro-step rule uses to define the macro-step identifier  $t_{o,tk_0,pp_1}$  associated to the resumption of the waiting task.

*Example 2.* Let us consider again the derivation in Sec. 3. We have the following blocking interval  $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_1$  with  $St_1 \equiv (S_1, table_1)$ , since  $t_{cl,wakeup,21} \in table_1$ ,  $status(t_{cl,wakeup,21}, table_1) = [24:f.block]$ ,  $(f, ba, cuts, 12) \in St_1$  and  $t_{ba,cuts,12} \notin table_1$ . This blocking interval captures the fact that the task at  $t_{cl,wakeup,21}$  is blocked waiting for task *cuts* to terminate. Similarly, we have the following two intervals in  $St_4$ :  $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$  and  $(t_{ch,taken,15}, t_{cl,sits,25}, t_{ch,taken,17})$ .

The following notion of *deadlock chain* relies on the waiting/blocking intervals of Def. 1 in order to characterize chains of calls in which intuitively each task is waiting for the next one to terminate until the last one which is waiting on the termination of a task executing on the initial location (that is blocked). Given a time identifier  $t$ , we use  $loc(t)$  to obtain its associated location identifier.

**Definition 2 (Deadlock Chain).** Let  $St = (S, table)$  be a state. A chain of time identifiers  $t_0, \dots, t_n$  is a deadlock chain in  $St$ , written as  $dc(t_0, \dots, t_n)$  iff  $\forall t_i \in \{t_0, \dots, t_{n-1}\}$  s.t.  $(t_i, t'_{i+1}, next(t_i)) \in St$  one of the following conditions holds:

1.  $t_{i+1} \in suc(t'_{i+1}, table)$ , or
2.  $loc(t'_{i+1}) = loc(t_{i+1})$  and  $(t_{i+1}, \neg, next(t_{i+1}))$  is blocking.

and for  $t_n$ , we have that  $t_{n+1} \equiv t_0$ , and condition 2 holds.

Let us explain the two conditions in the above definition: In condition (1), we check that when a task  $t_i$  is waiting for another task to terminate, the waiting interval contains the initial time  $t'_{i+1}$  in which the task will be selected. However, we look for any waiting interval for this task  $t_{i+1}$  (thus we check that  $t_{i+1}$  is a successor of time  $t'_{i+1}$ ). As in Def. 2, this is because such task may have started its execution and then suspended due to a subsequent await/block instruction. Abusing terminology, we use the time identifier to refer to the task executing. In condition (2), we capture deadlock chains which occur when a task  $t_i$  is waiting on the termination of another task  $t'_{i+1}$  which executes on a location  $loc(t'_{i+1})$  which is blocked. The fact that is blocked is captured by checking that there is a blocking interval from a task  $t_{i+1}$  executing on this location. Finally, note that the circularity of the chain, since we require that  $t_{n+1} \equiv t_0$ .

**Theorem 1 (Deadlock state).** *A state  $St$  is deadlock, written  $deadlock(S)$ , if and only if there is a deadlock chain in  $St$ .*

Derivations ending in a deadlock state are considered complete derivations. Correctness proofs can be found in the Appendix. We prove that our definition of deadlock is equivalent to the standard definition of deadlock in [11, 9].

*Example 3.* Following Ex. 1,  $St_4$  is a deadlock state since there exists a *deadlock chain*  $dc(t_{cl,wakeup,21}, t_{ba,sleeps,9}, t_{ch,taken,15})$ . For the second element in the chain  $t_{ba,sleeps,9}$ , condition 1 holds as  $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11}) \in St_4$  and  $t_{ch,taken,15} \in suc(t_{ch,taken,15}, table_4)$ . For the first element  $t_{cl,wakeup,21}$ , condition 2 holds since  $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_4$  and  $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$  is blocking. Condition 2 holds analogously for  $t_{ch,taken,15}$ .

## 5 Combining Static Deadlock Analysis and Testing

This section proposes a deadlock detection methodology that combines static analysis and testing as follows. First, a state-of-the-art deadlock analysis is run, in particular that of [11], which provides a set of abstractions of potential *deadlock cycles*. If the set is empty, then the program is deadlock-free. Otherwise, using the inferred set of deadlock cycles, we test the program using our enhanced semantics with two goals: (1) finding concrete deadlock traces associated to the different cycles, and, (2) discarding deadlock cycles, and in case all cycles are discarded, ensure deadlock freedom for the considered input or, in our case, for the `main` method under test.

### 5.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [11] returns a set of abstract deadlock cycles of the form  $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \dots \xrightarrow{p_n:tk_n} e_1$ , where  $p_1, \dots, p_n$  are program points,  $tk_1, \dots, tk_n$  are *task abstractions*, and nodes  $e_1, \dots, e_n$  are either *location abstractions* or task abstractions. Three kinds of arrows can be distinguished, namely, *task-task* (a task is awaiting for the termination of another one), *task-location* (a task is awaiting for a location to be idle) and *location-task* (the location is blocked due the task). *Location-location* arrows cannot happen. The abstractions for tasks and locations can be performed at different levels of accuracy

during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location  $o$  by the program point at which it is created  $o_{pp}$ , and each task by the method name executing. They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Both the analysis and the semantics can be made *object-sensitive* [3] by keeping the  $k$  ancestor abstract locations (where  $k$  is a parameter of the analysis). For the sake of simplicity of the presentation, we assume  $k = 0$  in the formalization (our implementation uses  $k = 1$ ).

*Example 4.* In our working example there are three abstract locations,  $o_2$ ,  $o_3$  and  $o_4$ , corresponding to locations `barber`, `client` and `chair`, created at lines 2, 3 and 4; and six abstract tasks, *sleeps*, *cuts*, *wakeup*, *sits*, *taken* and *isClean*. The following cycle is inferred by the deadlock analysis:  $o_2 \xrightarrow{11:sleeps} taken \xrightarrow{17:taken} sits \xrightarrow{25:sits} o_3 \xrightarrow{24:wakeup} cuts \xrightarrow{12:cuts} o_2$ . The first arrow captures that the location created at L2 is blocked waiting for the termination of task `taken` because of the synchronization at L11 of task *sleeps*. Observe that cycles contain dependencies also between tasks, like the second arrow, where we capture that `taken` is waiting for *sits*. Also, a dependency between a task (e.g., *sits*) and a location (e.g.,  $o_3$ ) captures that the task is trying to execute on that (possibly) blocked location. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can be observed, it is complex to figure out from them why these dependencies arise, and in particular the interleavings scheduled to lead to this situation.

## 5.2 Guiding Testing towards Deadlock Cycles

Given an abstract deadlock cycle, we now present a novel technique to guide the execution towards paths that might contain a representative of that abstract deadlock cycle, by discarding paths that are guaranteed not to contain such a representative. The main idea is as follows: (1) From the abstract deadlock cycle, we generate *deadlock-cycle constraints*, which must hold in all states of derivations leading to the given deadlock cycle. (2) We extend the execution semantics to support deadlock-cycle constraints, with the aim of stopping derivations as soon as cycle-constraints are not satisfied. Uppercase letters in constraints denote variables to allow representing incomplete information.

**Definition 3 (Deadlock-cycle constraints).** *Given a state  $St = (S, table)$ , a deadlock-cycle constraint takes one of the following three forms:*

1.  $\exists t_{O,T,PP} \mapsto \langle N, \rho \rangle$ , which means that there exists or will exist an entry of this form in table (time constraint)
2.  $\exists fut(F, O, Tk, p)$ , which means that there exists or will exist a future variable of this form in  $S$  (fut constraint)
3.  $pending(Tk)$ , which means that task  $Tk$  has not finished (pending constraint)

The following function  $\phi$  computes the set of deadlock-cycle constraints associated to a given abstract deadlock cycle.

**Definition 4 (Generation of deadlock-cycle constraints).** Given an abstract deadlock cycle  $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \dots \xrightarrow{p_n:tk_n} e_1$ , and two fresh variables  $O_i, Tk_i$ ,  $\phi$  is defined as  $\phi(e_i \xrightarrow{p_i:tk_i} e_j \xrightarrow{p_j:tk_j} \dots, O_i, Tk_i) =$

$$\begin{cases} \{ \exists t_{O_i, Tk_i, -} \mapsto \langle -, \text{sync}(p_i, F_i) \rangle, \exists \text{fut}(F_i, O_j, Tk_j, p_j) \} \cup \phi(e_j \xrightarrow{p_j:tk_j} \dots, O_j, Tk_j) & \text{if } e_j = tk_j \\ \{ \text{pending}(Tk_i) \} \cup \phi(e_j \xrightarrow{p_j:tk_j} \dots, O_i, Tk_j) & \text{if } e_j = o \end{cases}$$

Notation  $\text{sync}(p_i, F_i)$  is a shortcut for  $p_i:F_i.\text{block}$  or  $p_i:F_i.\text{await}$ . Uppercase letters appearing for the first time in the constraints are fresh variables. The first case handles location-task and task-task arrows (since  $e_j$  is a task abstraction), whereas the second case handles task-location arrows ( $e_j$  is an abstract location). Let us observe the following: (1) The abstract location and task identifiers of the abstract cycle are not used to produce the constraints. This is because constraints refer to concrete identifiers. Even if the cycle contains the same identifier on two different nodes or arrows, the corresponding variables in the constraints cannot be bound (i.e., we cannot use the same variables) since they could refer to different concrete identifiers. (2) The program points of the cycle ( $p_i$  and  $p_j$ ) are used in time and fut constraints. (3) Location and task identifier variables of fut constraints and subsequent time or pending constraints are bound (i.e., the same variables are used). This is done using the 2nd and 3rd parameters of function  $\phi$ . (4) In the second case,  $Tk_j$  is a fresh variable since the location executing  $Tk_i$  can be blocked due to a (possibly) different task. Intuitively, deadlock-cycle constraints characterize all possible deadlock chains representing the given cycle.

*Example 5.* The following deadlock-cycle constraints are computed for the cycle in Ex. 4:  $\{ \exists t_{O_1, Tk_1, -} \mapsto \langle -, 11:F_1.\text{block} \rangle, \exists \text{fut}(F_1, O_2, Tk_2, 15), \exists t_{O_2, Tk_2, -} \mapsto \langle -, 17:F_2.\text{await} \rangle, \exists \text{fut}(F_2, O_3, Tk_3, 25), \text{pending}(Tk_3), \exists t_{O_3, Tk_4, -} \mapsto \langle -, 24:F_3.\text{block} \rangle, \exists \text{fut}(F_3, O_4, Tk_5, 12), \text{pending}(Tk_5) \}$ . They are shown in the order in which they are computed by  $\phi$ . The first four constraints require *table* to contain a concrete time in which *some* barber sleeps waiting at L11 for a *certain* chair to be taken at L15 and, during another concrete time, this one waits at L17 for a *certain* client to sit at L25. The client is not allowed to sit by the 5th constraint. Furthermore, the last three constraints require a concrete time in which *this* client waits at L24 to get a haircut by *some* barber at L12 and that haircut is never performed. Note that, in order to preserve completeness, we are not binding the first and the second barber. If the example is generalized with several clients and barbers, there could be a deadlock in which a barber waits for a client which waits for another barber and client, so that the last one waits to get a haircut by the first one. This deadlock would not be found if the two barbers are bound in the constraints (i.e., if we use the same variable name). In other words, we have to account for deadlocks which traverse the abstract cycle more than once.

The idea now is to monitor the execution using the inferred deadlock-cycle constraints for the given cycle, with the aim of stopping derivations at states that do not satisfy the constraints. The following boolean function  $\text{check}_{\mathcal{C}}$  checks the satisfiability of the constraints at a given state.

**Definition 5.** Given a set of deadlock-cycle constraints  $\mathfrak{C}$ , and a state  $St = (S, table)$ , *check holds*, written  $check_{\mathfrak{C}}(St)$ , if  $\forall t_{O_i, Tk_i, PP} \mapsto \langle N, sync(p_i, F_i) \rangle \in \mathfrak{C}, fut(F_i, O_j, Tk_j, p_j) \in \mathfrak{C}$ , one of the following conditions holds:

1.  $reachable(t_{O_i, Tk_i, p_i}, S)$
2.  $\exists t_{o_i, tk_i, pp} \mapsto \langle n, sync(p_i, f_i) \rangle \in table \wedge fut(f_i, o_j, tk_j, p_j) \in S \wedge$   
 $(pending(Tk_j) \in \mathfrak{C} \Rightarrow getTskSeq(tk_j, S) \neq \epsilon)$

Function `reachable` checks whether a given task might arise in subsequent states. We over-approximate it syntactically by computing the transitive call relations from all tasks in the queues of all locations in  $S$ . Precision could be improved using more advanced analyses. Function `getTskSeq` gets from the state the sequence of instructions to be executed by a task (which is  $\epsilon$  if the task has terminated). Intuitively, `check` does not hold if there is at least a time constraint so that: (i) its time identifier is not reachable, and, (ii) in the case that the interleavings table contains entries matching it, for each one, there is an associated future variable in the state and a pending constraint for its associated task which is violated, i.e., the associated task has finished. The first condition (i) implies that there cannot be more representatives of the given abstract cycle in subsequent states, therefore if there are potential deadlock cycles, the associated time identifiers must be in the interleavings table. The second condition (ii) implies that, for each concrete potential cycle in the state, there is no deadlock chain since at least one of the blocking tasks has finished. This means there cannot be derivations from this state leading to the given deadlock cycle, therefore this derivation can be stopped. Function `check $_{\mathfrak{C}}$`  is used in the semantics to prune deadlock-free derivations as showed in Figure 3.

The following definition presents the notion of deadlock-cycle guided testing.

**Definition 6 (Deadlock-cycle guided-testing (DCGT)).** Consider an abstract deadlock cycle  $c$ , and an initial state  $St_0$ . Let  $\mathfrak{C} = \phi(c, O_{init}, Tk_{init})$  with  $O_{init}, Tk_{init}$  fresh variables. We define  $DCGT$ , written  $exec_c(St_0)$ , as the set  $\{d : d \in exec(St_0), \text{deadlock}(St_n)\}$ , where  $St_n$  is the last state in  $d$ .

*Example 6.* Let us consider the DCGT of our working example with the deadlock-cycle of Ex. 4, and hence with the constraints  $\mathfrak{C}$  of Ex. 5. The interleavings table at  $St_5$  contains the entries  $t_{ini, main, 1} \mapsto \langle 1, return \rangle$ ,  $t_{cl, wakeup, 21} \mapsto \langle 2, 24: f_0.block \rangle$  and  $t_{ba, cuts, 12} \mapsto \langle 3, return \rangle$ . `check $_{\mathfrak{C}}$`  does not hold since  $t_{O_1, Tk_1, 24}$  is not reachable from  $St_5$  and constraint `pending( $Tk_5$ )` is violated (task *cuts* has already finished at this point). The derivation is hence pruned. Similarly, the rightmost derivation is stopped at  $St_{11}$ . Also, derivations at  $St_4$ ,  $St_8$  and  $St_{10}$  are stopped by function `deadlock` of Th. 1. Our deadlock guided testing methodology generates 16 states instead of the 181 generated by the standard exhaustive execution.

**Theorem 2 (Soundness).** Given a program  $P$ , a set of abstract cycles  $C$  in  $P$  and an initial state  $St_0$ ,  $\forall d \in exec(St_0)$  if  $d$  is a derivation whose last state is *deadlock*, then  $\exists c \in C$  such that  $d \in exec_c(St_0)$ .



### 5.3 Deadlock-based Testing Criteria

In the application of testing for deadlock detection, and in a general setting where there could arise many potential deadlock cycles, the following practical questions arise: is a user interested in just finding the first deadlock trace? or do we rather need to obtain all deadlock traces? For the purpose of the programmer to identify and fix the sources of the deadlock error(s), it could be more useful to find a deadlock trace per abstract deadlock cycle. This is the kind of questions that test adequacy criteria answer. Using our methodology, we are able to provide the following *deadlock-based adequacy criteria*:

- **first-deadlock**, which requires exercising at least one deadlock execution,
- **all-deadlocks**, which requires exercising all deadlock executions,
- **deadlock-per-cycle**, which, for each abstract deadlock cycle, requires exercising at least one deadlock execution representing the given cycle (if exists)

We have developed concrete testing schemes for each criteria above relying on our DCGT methodology. For **first-deadlock**, DCGT is called for each abstract deadlock cycle until finding the first deadlock. For both **all-deadlocks** and **deadlock-per-cycle**, DCGT is also called for each abstract cycle, but with the difference that the different DCGTs can be run in parallel since they are completely independent. In the case of **deadlock-per-cycle**, each DCGT finishes as soon as a deadlock representing the corresponding cycle is found. It can also be very practical to set a time-limit per DCGT to prevent that the state explosion on a certain DCGT degrades the efficiency of the whole exploration.

## 6 Experimental Evaluation

We have implemented our approach within the tool aPET [5], a test case generator for *concurrent objects* which is available at <http://costa.ls.fi.upm.es/apet>, where the benchmarks in this paper can also be found. Concurrent objects communicate via *asynchronous* method calls and use **await** and **block**, resp., as instructions for non-blocking and blocking synchronization. Therefore, the language in Sec. 2 fully captures their concurrency model. This section summarizes our experimental results which have been performed using as benchmarks: (i) classical concurrency patterns containing deadlocks, namely *SB* is an extension of the sleeping barber with several clients, *UL* is a loop that creates asynchronous tasks and locations, *PA* the pairing problem, *FA* is a distributed factorial, *WM* making a water molecule, *HB* the hungry birds problem, and, (ii) deadlock free versions of some of the above, named *fX* for the *X* problem, for which deadlock analyzers give false positives. We include here a peer-to-peer system *P2P*.

Table 1 shows the results obtained using three different settings: (1) the first set of columns **Exh** corresponds to building the whole search tree, (2) the second to the **first-deadlock** criterion, and (3) the third to the **deadlock-per-cycle** criterion. For each setting *i*, we measure the total time taken (column  $T_i$ ) and the number of states generated (column  $S_i$ ). Column *Ans* contains the solutions obtained by the whole execution tree. Column *D/F/C* in the third setting shows “number of deadlock executions” / “number of unfeasible cycles” / “number of abstract cycles”

	(1) Exh				(2) first-deadlock				(3) deadlock-per-cycle				S-up	
Bm.	Ans	$T_1$	$S_1$	$T_2$	$S_2$	D/F/C	$T_3$	$T_{Max}$	$S_3$	$S_{Max}$	$T_{up}$	$S_{up}$		
SB	103k	$\infty$	>584k	62	23	1/0/1	59	11	23	23	$\infty$	$\infty$		
UL	90k	$\infty$	>489k	150	5	1/0/1	133	3	5	5	$\infty$	$\infty$		
PA	121k	$\infty$	>329k	40	6	2/0/2	42	4	12	6	$\infty$	$\infty$		
WM	82k	$\infty$	>380k	248	15	1/0/2	$\infty$	$\infty$	>258k	>258k	-	-		
HB	35k	32k	114k	82	15	2/3/5	44k	15k	103k	34k	2.15	3.33		
FA	11k	11k	41k	786	1k	2/1/3	2k	759	3k	2k	15.07	22.19		
fFA	5k	7k	25k	5k	11k	0/1/1	5k	5k	11k	11k	1.61	2.35		
fP2P	25k	66k	118k	34k	52k	0/1/1	34k	34k	52k	52k	1.96	2.28		
fUL	102k	$\infty$	>527k	435	236	0/1/1	410	230	236	236	$\infty$	$\infty$		
fPA	7k	7k	30k	4k	9k	0/2/2	4k	2k	9k	4k	3.73	6.98		

Table 1. Experimental evaluation

found by the analysis. For instance, for *HB* we have 2/3/5 that shows that the analysis has found five abstract cycles, but we only found a deadlock execution for two of them, therefore 3 of them were unfeasible. Since the DCGTs in setting 3 can be performed in parallel, columns  $T_{max}$  and  $S_{max}$  show the maximum time and number of states measured among all of them. Columns in **S-up** show the gain of setting 3 w.r.t. 1 computed as  $T_{up} = T_1/T_{max}$  (the gain is  $\infty$  when  $T_1$  is  $\infty$  and  $T_{max}$  is not, or none “-” when  $T_{max}$  is  $\infty$  too), and analogously for states. Times are in milliseconds and are obtained on an Intel(R) Core(TM) i7 CPU at 2.3GHz with 8GB of RAM, running Mac OS X 10.8.5. A timeout of 150.000ms (written 150k) is used. When the timeout is reached we write  $\infty$ .

When comparing setting 2 w.r.t. 1, we see that, if the program features a deadlock, our guided-testing is very effective, e.g., by just exploring 6 states in 40ms the deadlock is found in *PA*. When the program is deadlock free, we need to explore the whole execution also in setting 2. Although the (spurious) information provided by the analysis does not allow much pruning in these cases, still there is a notable gain (e.g., in *fPA* we explore about one third of the states explored in setting 1 and the time is almost halved). Importantly, we are able to prove deadlock freedom in all examples while exhaustive exploration times out in *fUL*. As regards setting 3, we achieve significant gains w.r.t. exhaustive exploration for deadlock-free examples (e.g., by just exploring 23 states in *SB* we found one representative per cycle in 59ms. while setting 1 times out). The gains are much larger in the examples in which the deadlock analysis does not give false positives (namely, in *SB*, *UL*, *PA*). For *WM*, we have failed to find a representative of a potential cycle within the timeout. This is because every abstract cycle produces different constraints, some of them allow important pruning during testing as they impose very restrictive conditions, whereas others can hardly guide because most of derivations fulfill the constraints. When this happens, the number of states explored is slightly smaller than with exhaustive execution. However, when we consider that each DCGT is computed in parallel for each cycle (columns **S-up**), we achieve further gains (in *SB*, *UL*, *HB* and *PA* we decrease the time notably) and in *WP* we perform slightly better than in set-

ting 1. Finally, for the examples that are deadlock free, the number of explored states for settings 2 and 3 is the same. This is because in order to ensure that a deadlock representative cannot be found, it is necessary to make exhaustive exploration with every abstract cycle. All in all, we argue that our experiments show that our methodology is very effective for programs that contain deadlock, and it is able also to prove deadlock freedom for some cases in which a static analysis reports false positives.

## 7 Conclusions and Related Work

There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [11, 12, 9] and thread-based programs [16, 18], is based on static analysis techniques. Static analysis can ensure the absence of errors, however it works on approximations (especially for handling iteration and pointer aliasing) which might lead to a “don’t know” answer. Our work complements static analysis techniques and can be used to look for deadlock paths when static analysis is not able to prove the absence of deadlock. Using our method, if there might be a deadlock, we try to find it by exploring the paths given by our deadlock detection algorithm that relies on the static information.

Deadlock detection has been also studied in the context of dynamic testing and model checking [15, 14, 8, 7], where sometimes has been combined with static information [13, 2]. As regards combined approaches, the approach in [13] first performs a transformation of the program into a trace program that only keeps the instructions that are relevant for deadlock and then dynamic testing is performed on such program. The approach is fundamentally different from ours: in their case, since model checking is performed on the trace program (that over-approximates the deadlock behaviour), this method can detect deadlocks that do not exist in the program, while in our case this is not possible since the testing is performed on the original program and the analysis information is only used to drive the execution. In [2], the information inferred from a type system is used to accelerate the detection of potential cycles. This work shares with our work that information inferred statically is used to improve the performance of the testing tool, however there are important differences: first, their method developed for Java threads captures deadlocks due to the use of locks and cannot handle wait-notify, while our technique is not developed for specific patterns but rather works on a general characterization of deadlock of asynchronous programs; their underlying static analysis is a type inference algorithm which infers deadlock types and the checking algorithm needs to understand these types to take advantage of them, while we base our method on an analysis which infers descriptions of chains of tasks and a formal semantics is enriched to interpret them; additional contributions of our work are the deadlock-based testing criteria.

Finally, although we have presented our technique in the context of dynamic testing, our approach would be applicable also in static testing where the execution is performed on constraints variables rather than on concrete values. This extension will require the use of termination criteria which provide the desired degree of coverage. This remains as subject for future research.

## References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal dynamic partial order reduction. In *Proc. of POPL'14*, pages 373–384. ACM, 2014.
2. R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Conf. on Hardware and Software Verification and Testing, LNCS 3875*, pages 191–207. Springer, 2006.
3. E. Albert, P. Arenas, J. Correias, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.
4. 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.
5. E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In *FSE'13*, pp. 595–598. ACM, 2013.
6. E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. Technical report, 2015. Available at: <http://costa.ls.fi.upm.es/papers/costa/AlbertGI15.pdf>.
7. B. D. Bingham, J. D. Bingham, J. Erickson, and M. R. Greenstreet. Distributed Explicit State Model Checking of Deadlock Freedom. In *Proc. of CAV'13*, volume 8044 of *Lecture Notes in Computer Science*, pages 235–241. Springer, 2013.
8. M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2013 IEEE Sixth International Conf. on Software Testing, Verification and Validation*, pages 154–163. IEEE, 2013.
9. F. S. de Boer, M. Bravetti, I. Grabe, M. David Lee, M. Steffen, and G. Zavattaro. A Petri Net based Analysis of Deadlocks for Active Objects and Futures. In *Proc. of FACS 2012*, 2012.
10. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proc. POPL'05*, pp. 110–121. ACM, 2005.
11. A. Flores, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. *FORTE'13, LNCS*, pp 273–288. Springer, 2013.
12. E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.
13. P. Joshi, M. Naik, K. Sen, and Gay D. An effective dynamic analysis for detecting generalized deadlocks. In *Proc. of FSE'10*, pages 327–336. ACM, 2010.
14. P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI'09*, pages 110–120. ACM, 2009.
15. A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report <http://dslab.epfl.ch/pubs/lockout.pdf>, 2013.
16. S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *PDD'91*, pages 97–107. ACM, 1991.
17. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. of ICSE*, pages 386–396. IEEE, 2009.
18. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
19. K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *Proc. FASE'06, LNCS 3922*, pp. 339–356. Springer, 2006.
20. 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. *FORTE, LNCS 7273*, pages 219–234. Springer, 2012.

## 8 Appendix

*Proof (Proof of Theorem 1).*

Given a program state  $St = (S, table)$ , its *dependency graph*  $G_S$  and its *abstract dependency graph*  $\mathcal{G}$  are formalized in [11]. Let us define the function  $\gamma$  that transforms a *sequence of times* that each of them fulfills (1) or (2) in Def. 2 into a path in  $G_S$ .

**Definition 7** ( $\gamma$ ). *Given a state  $St=(S, table)$  and a sequence of times  $\{t_0, \dots, t_n\}$  in  $St$ , satisfying (1) or (2) in Def. 2. The one-to-one function  $\gamma(\{t_0, \dots, t_n\})=e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n$  in  $G_S$  is defined as follows:*

$$\gamma(\{t_0, \dots, t_n\}) = \begin{cases} \{loc(t_0) \rightarrow tsk(t_1)\} \cup \gamma_{tk}(\{t_1, \dots, t_n\}) & \text{if } t_0 \text{ holds (1)} \\ \{loc(t_0) \rightarrow tsk(t'_1) \rightarrow loc(t'_1)\} \cup \gamma(\{t_1, \dots, t_n\}) & \text{if } t_0 \text{ holds (2)} \wedge \neg(1) \end{cases}$$

where  $\gamma_{tk}$  is the following auxiliary function:

$$\gamma_{tk}(\{t_0, \dots, t_n\}) = \begin{cases} \{tsk(t_0) \rightarrow tsk(t_1)\} \cup \gamma_{tk}(\{t_1, \dots, t_n\}) & \text{if } t_0 \text{ holds (1)} \\ \{tsk(t_0) \rightarrow tsk(t'_1) \rightarrow loc(t'_1)\} \cup \gamma(\{t_1, \dots, t_n\}) & \text{if } t_0 \text{ holds (2)} \wedge \neg(1) \end{cases}$$

We need to distinguish between functions  $\gamma$  and  $\gamma_{tk}$ , as in [11], a location blocked in a task could be represented in  $G_S$  by both the location identifier and the blocked task identifier, depending on the previous context. The intuition of function  $\gamma$  ( $\gamma_{tk}$ ) is: given a *sequence of times*  $\{t_0, \dots, t_n\} \in St$ , we define a path whose edges are obtained as follows:  $\forall t_i \in \{t_0, \dots, t_n\}$  such that  $(t_i, t'_{i+1}, next(t_i)) \in St$ . if (1) is held, then there exists an *edge t-t* between  $tsk(t_i)$  and  $tsk(t_{i+1})$  (an *edge edge o-t* between  $loc(t_i)$  and  $tsk(t_{i+1})$ ), as  $tsk(t'_{i+1}) = tsk(t_{i+1})$  by definition of function *suc*. On the other hand, if 2 and  $\neg 1$  are held, then there exist two edges in  $G_S$ : an *edge t-o* between  $tsk(t'_{i+1})$  and  $loc(t'_{i+1})$ , as this task belongs to a location which is blocked and an *edge t-t (edge o-t)*, between  $tsk(t_i)$  and  $tsk(t'_{i+1})$ , (between  $loc(t_i)$  and  $tsk(t'_{i+1})$ ).

**Lemma 1** ([3]). *Let  $S$  be a reachable state and  $G_S^{tt}$  the dependencies graph taking only task-task dependencies. If future variables cannot be stored in fields,  $G_S^{tt}$  is acyclic.*

**Theorem 3 (equivalence).** *Let  $St$  be a program state,*

$$\exists dc(\{t_0, \dots, t_n\}) \in St \iff \exists \text{ cycle } \gamma(\{t_0, \dots, t_n\}) \in G_S$$

*Proof.*

$\Rightarrow$ . Let  $dc(\{t_0, \dots, t_n\})$  be a deadlock chain, then we could apply the function  $\gamma$ , as  $\forall t_i \in \{t_0, \dots, t_n\}$ ,  $t_i$  satisfies (1) or (2). So, we obtain a path in  $G_S$  and using the last condition in Def. 2, both  $\gamma(\{t_n\})$  and  $\gamma_{tk}(\{t_n\})$  add the edge  $tk(t'_0) \rightarrow loc(t_0)$  causing the path becomes a cycle.

$\Leftarrow$ . Given a cycle in  $G_S$ , by the lemma 1, this one contains at least one object node, which is required by the function  $\gamma$ . Now, This case is analogous to the previous one.

The proof of Theorem 2 relies on the soundness of both the points-to and the deadlock analyses that we state below. We first define an auxiliary operation that performs the union between to disjunct partial maps:

**Definition 8 ( $l+a$ ).** Let  $l$  and  $a$  be two partial maps such that  $\text{dom}(l) \cap \text{dom}(a) = \emptyset$ :

- $(l+a)(x) = l(x)$  iff  $x \in \text{dom}(l)$
- $(l+a)(x) = a(x)$  iff  $x \in \text{dom}(a)$

**Definition 9 (points-to soundness [3]).** Soundness of the points-to analysis amounts to requiring the existence a partial map  $\alpha$ , that maps location and task identifiers to corresponding abstract ones, such that for any task  $\text{tsk}(tk, m, o, l, s)$ , where  $o$  is the object identifier that executes the task  $tk$ , and location  $\text{loc}(o, tkh, \mathcal{Q})$  in any reachable state  $S$ , we have that:

1.  $\alpha(tk) = \alpha(o).m$
2. Let  $x$  be an location variable  $x \in \text{dom}(l+h)$ , if  $\alpha((l+h)(x)) = ob$  then  $ob \in \mathcal{A}(\alpha(o), pp(s), x)$ .
3. Let  $x$  be future variable,  $x \in \text{dom}(l+h)$ ,  $(l+h)(x) = tk_2$  and  $\text{tsk}(tk_2, m_2, o_2, l_2, \epsilon(v)) \in T$  (i.e.,  $x$  is a variable that points to a finished task). Then, given  $\alpha(tk_2) = tk$ , either the task identifier or the ready task identifier belong to the points-to result.  $\{tk, tk_r\} \cap \mathcal{A}(\alpha(o), pp(s), x) \neq \emptyset$ .
4. Let  $x$  be future variable,  $x \in \text{dom}(l+h)$ ,  $(l+h)(x) = tk_2$ ,  $\text{tsk}(tk_2, m_2, o_2, l_2, s_2) \in T$  and  $s_2 \neq \epsilon(v)$  (i.e., the pointed task  $tk_2$  is not finished). Then, given  $\alpha(tk_2) = tk$ , the task identifier belongs to the points-to result,  $tk \in \mathcal{A}(\alpha(o), pp(s), x)$ .

Let  $\bar{\alpha}$  be the extension of  $\alpha$  over the paths in  $G_s$  that applies the function  $\alpha$  in every node contained by the path.

**Definition 10 (deadlock soundness [3]).** Let  $S$  be a reachable state. If there is a cycle  $\gamma = e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_1$  in  $G_S$ , then  $\bar{\alpha}(\gamma) = \alpha(e_1) \xrightarrow{p_1:tk_1} \alpha(e_2) \xrightarrow{p_2:tk_2} \dots \xrightarrow{p_n:tk_n} \alpha(e_1)$  is an abstract cycle of  $\mathcal{G}$ .

**Lemma 2.** Given an initial state  $St_0$  and an abstract cycle  $c$ ,  $\forall d \in \text{exec}(St_0)$ ,  $d \equiv St_0 \rightarrow^* St_n$ , if  $\exists dc(\{t_0, \dots, t_n\}) \in St_n$  such that  $\bar{\alpha} \circ \gamma(\{t_0, \dots, t_n\}) \in c$ , then  $d \in \text{exec}_c(St_0)$ .

*Proof.* By contradiction, let us suppose that  $\exists d \in \text{exec}(St_0)$  and  $d \notin \text{exec}_c(St_0)$ . Hence,  $\exists St_i \in d$  such that  $\text{check}_{\mathcal{C}}(St_i)$  returns false and, consequently, the derivation  $St_0 \rightarrow^* St_i$  stops, where  $\mathcal{C} = \phi(c, O, Tk)$  and  $O, Tk$  are fresh variables. Therefore, at  $St_i \exists \{t_{O_i, Tk_i, PP} \mapsto \langle N, \text{sync}(p_i, F_i) \rangle \in \mathcal{C}, \text{fut}(F_i, O_j, Tk_j, p_j)\} \subset \mathcal{C}$  doesn't hold neither (1) nor (2) in Def. 5. However, this cannot happen, as  $\mathcal{C}$  imposes necessary constraints for the existence of some representative of  $c$  and  $St_n$  contains a cycle that is representative of  $c$ , then (1) or (2) must be fulfilled in every state of  $d$ . As a result, we get a contradiction.

*Proof (Proof of Theorem 2).* If the last state is deadlock, then  $\exists dc(\{t_0, \dots, t_n\}) \in St_n$ , by Th. 1. Using the soundness of deadlock analysis over the cycle  $\gamma(\{t_0, \dots, t_n\})$ , the existence of  $c$  is ensured. Now, by Lemma 2, we obtain the result.