SYCO & aPET **Tutorial**

http://abs-models.org/syco-apet-tutorial/

Elvira Albert
Puri Arenas
Miguel Gómez-Zamalloa
Miguel Isabel

http://www.envisage-project.eu/

Introduction

Writing correct concurrent programs is harder than writing sequential ones, because with concurrency comes 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. Thanks to the non-preemptive scheduling and the absence of shared memory among different objects we have in `ABS`, it suffices to consider such non-determinism only at *release* points, in order not to lose any behavior of the program. However, a naïve systematic exploration of all possible choices still does not scale.

Two different families of techniques can help in mitigating such a state explosion problem: (i) *Partial-order reduction* (POR) [9, 6] is a general theory that allows characterizing redundant 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. (ii) 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 is *deadlock-guided testing*, where the execution is driven towards potentially deadlock paths (while other paths are pruned).

The `SYCO` (resp. `aPET`) tool is a dynamic (resp. static) systematic testing tool for ABS concurrent objects which includes state-of-the-art POR and deadlock-guided testing techniques.

Contents

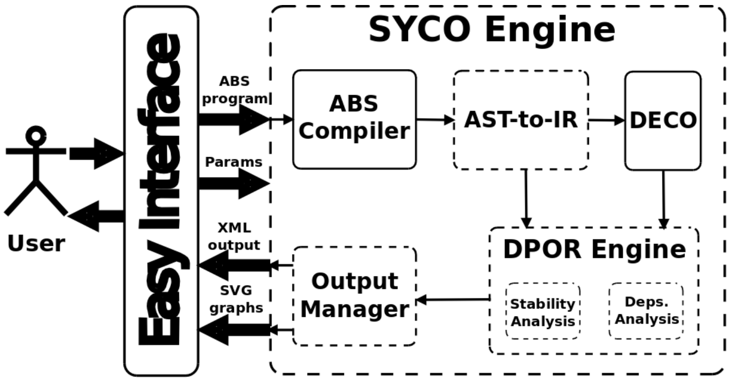


Figure 1: SYCO architecture

1 General Overview

1.1 The SYCO Tool

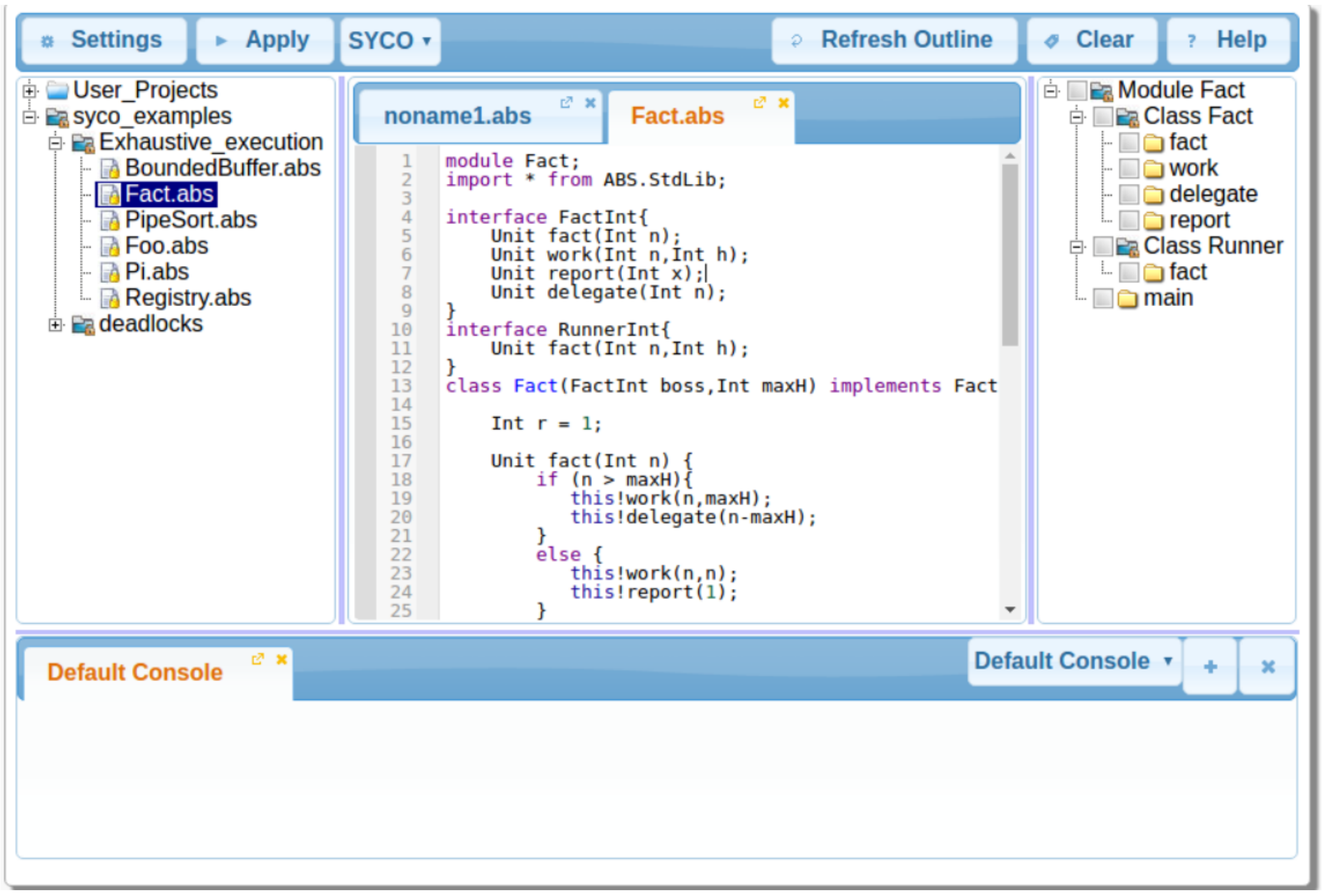
SYCO is a systematic tester for ABS concurrent objects. Figure 1 shows its main architecture. Boxes with dash lines are internal components of SYCO whereas boxes with regular lines are external components. The user interacts with SYCO through its web interface which is integrated within the ABS collaboratory and is hence provided by *EasyInterface* [8]. The 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 [1] and the *stability* and *dependencies* analyses of [1]. 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* [7] 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) [?].

Section 2 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 2 for details.

1.2 The aPET Tool

aPET is a static testing tool and test case generator for ABS concurrent objects based on *symbolic execution* [5]. In symbolic execution 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. 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 above-mentioned problem of the non-deterministic interleavings of processes, 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, [2] extends the POR techniques of [1] 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



```
module Fact;
import * from ABS.StdLib;

interface FactInt{
    Unit fact(Int n);
    Unit work(Int n,Int h);
    Unit report(Int x);
    Unit delegate(Int n);
}
interface RunnerInt{
    Unit fact(Int n,Int h);
}
class Fact(FactInt boss,Int maxH) implements Fact

    Int r = 1;

    Unit fact(Int n) {
        if (n > maxH){
           this!work(n,maxH);
           this!delegate(n-maxH);
        }
        else {
           this!work(n,n);
           this!report(1);
        }
```

Figure 2: Collaboratory web Interface

meaningful set of test-cases, aPET includes the coverage/termination criteria for concurrent objects proposed in [2]. Essentially it consists on limiting: the number of iterations of loops at the level of tasks, the task switches allowed in each concurrent object, and, the concurrency units originated during symbolic execution per program point.

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 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 details how to use aPET with screenshots and provides information about the different parameters which can be set.

2 SYCO: Step by Step

In order to use SYCO we select *Systematic testing (SYCO)* from the pull-down menu of available tools as shown in Figure 2. Our running example is shown in Figure 3 and available at the collaboratory here. 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 $n*(n-1)*\ldots*(n-o.maxH+1)$. 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.

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

Figure 3: Running Example

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 Section 2.3). 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.

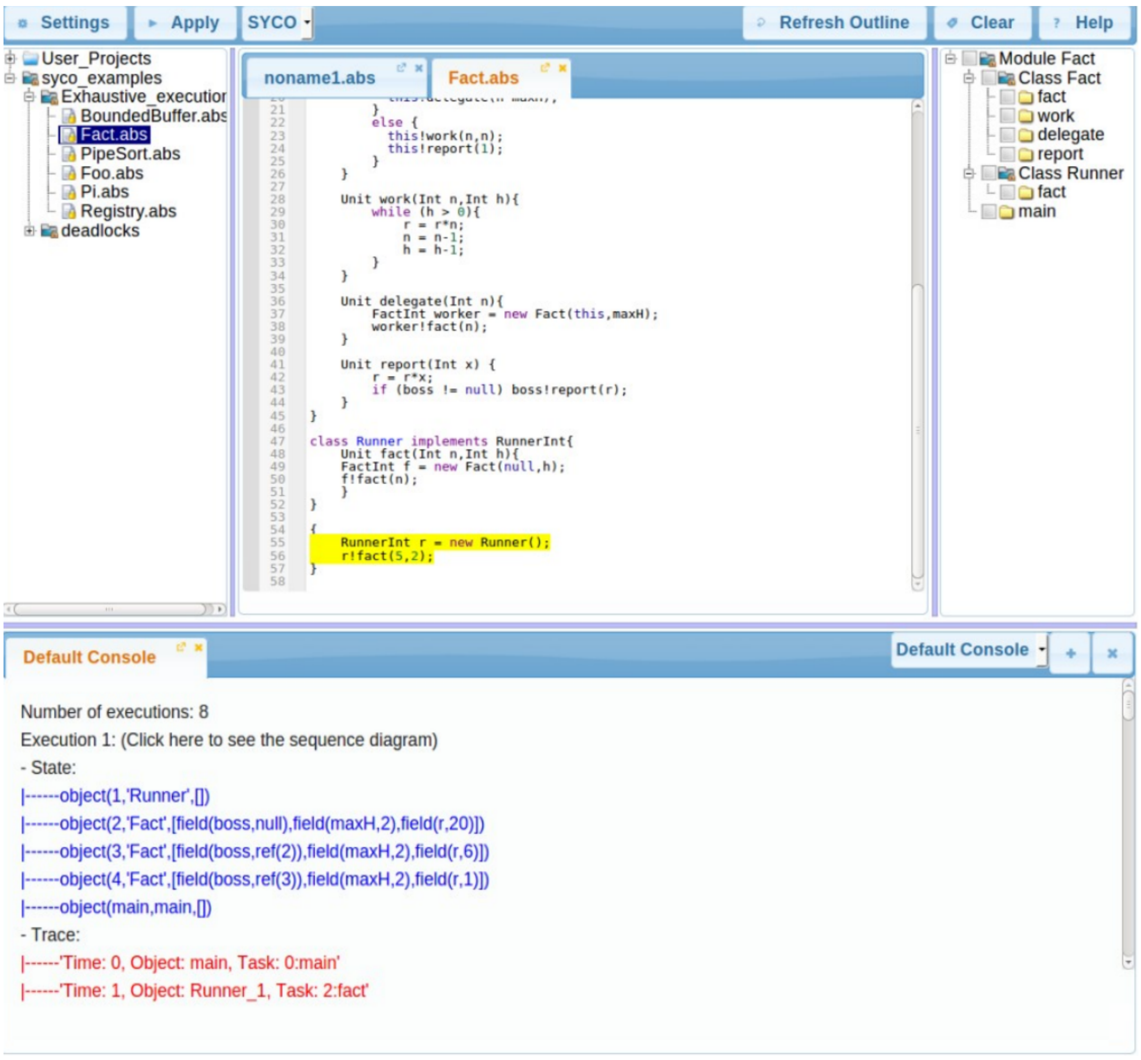
2.1 Using SYCO with default parameters

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.

`SYCO` first prints the number of complete executions explored (in this case eight executions). Note that, by default, an aggressive POR is applied. As we will see later, the number of executions without POR is 280. Also, the most recent POR technique included in `SYCO` is able to obtain just two executions. `SYCO` then 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`



```
            }
            else {
              this!work(n,n);
              this!report(1);
            }
        }

        Unit work(Int n,Int h){
            while (h > 0){
                r = r*n;
                n = n-1;
                h = h-1;
            }
        }

        Unit delegate(Int n){
            FactInt worker = new Fact(this,maxH);
            worker!fact(n);
        }

        Unit report(Int x) {
            r = r*x;
            if (boss != null) boss!report(r);
        }
}

class Runner implements RunnerInt{
        Unit fact(Int n,Int h){
        FactInt f = new Fact(null,h);
        f!fact(n);
        }
}

{
        RunnerInt r = new Runner();
        r!fact(5,2);
}
```

```
Number of executions: 8
Execution 1: (Click here to see the sequence diagram)
- State:
|------object(1,'Runner',[])
|------object(2,'Fact',[field(boss,null),field(maxH,2),field(r,20)])
|------object(3,'Fact',[field(boss,ref(2)),field(maxH,2),field(r,6)])
|------object(4,'Fact',[field(boss,ref(3)),field(maxH,2),field(r,1)])
|------object(main,main,[])
- Trace:
|------'Time: 0, Object: main, Task: 0:main'
|------'Time: 1, Object: Runner_1, Task: 2:fact'
```

Figure 4: Execution of `SYCO` with default parameters

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 4 where the first time (`|--'Time: 0, Object: main, Task: 0:main'`) of the trace has been clicked.

2.2 How to understand the sequence diagrams

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 5 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 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.

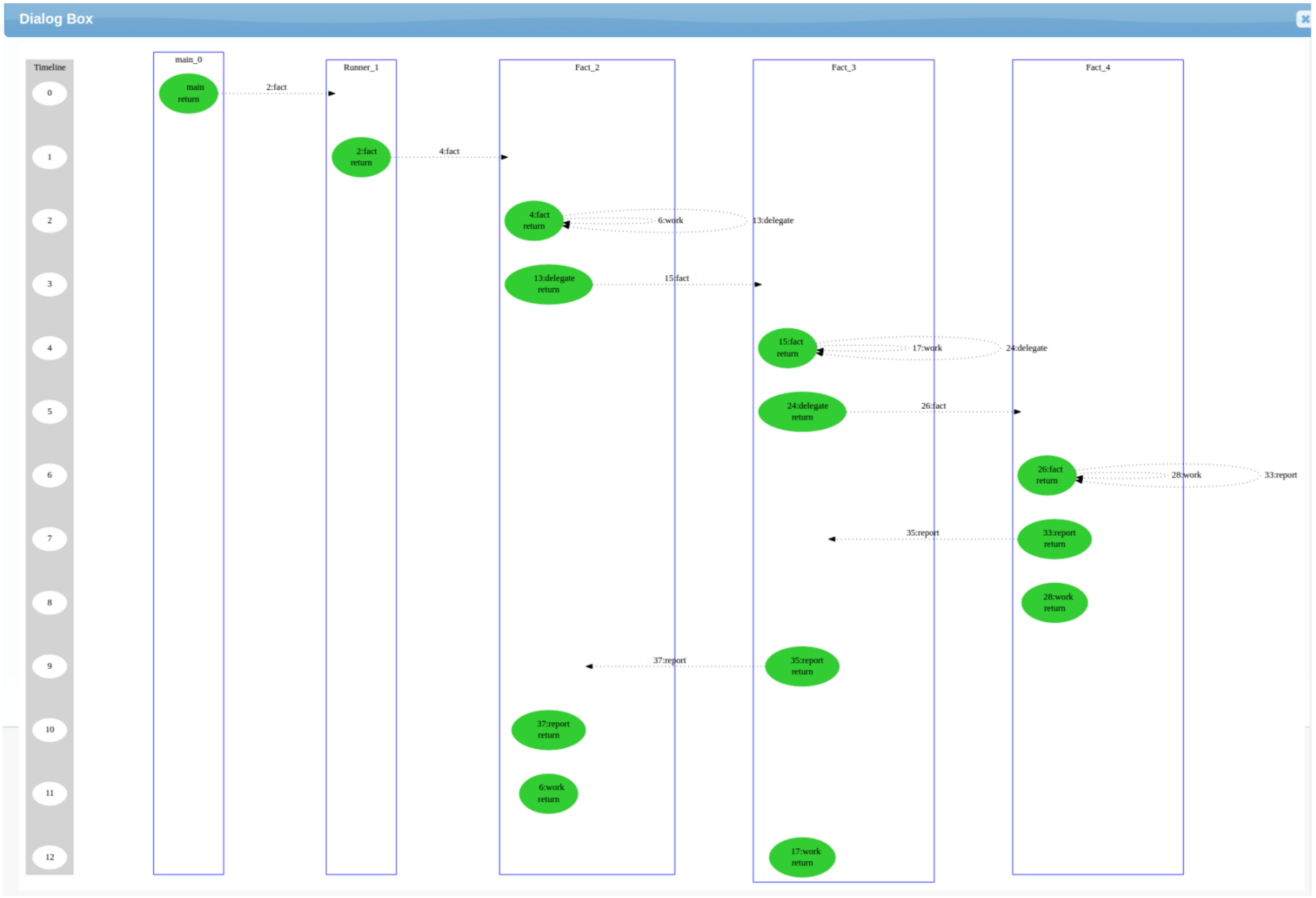
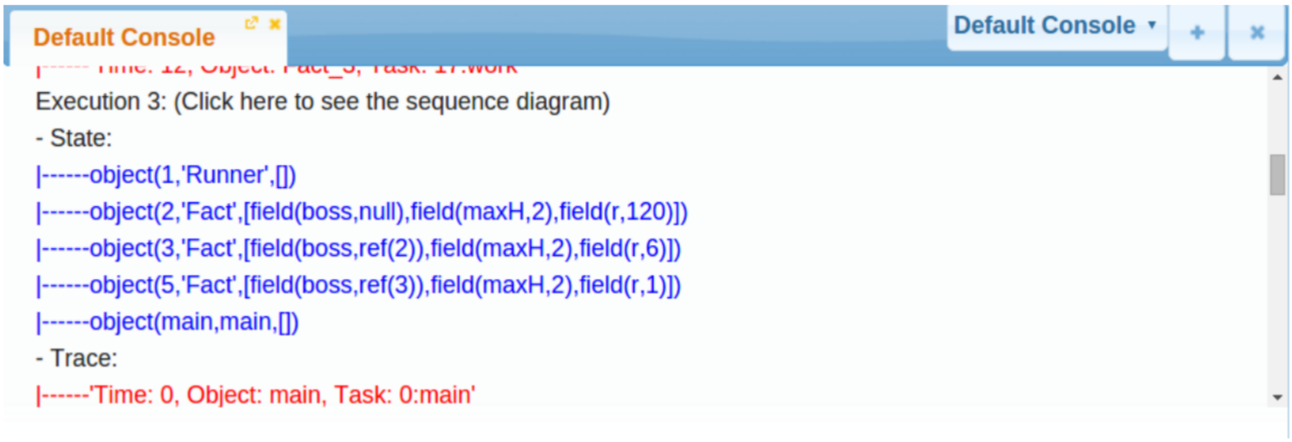


Figure 5: A buggy execution trace for the running example



```
|------ Time: 12, Object: Fact_3, Task: 17:work
Execution 3: (Click here to see the sequence diagram)
- State:
|------object(1,'Runner',[])
|------object(2,'Fact',[field(boss,null),field(maxH,2),field(r,120)])
|------object(3,'Fact',[field(boss,ref(2)),field(maxH,2),field(r,6)])
|------object(5,'Fact',[field(boss,ref(3)),field(maxH,2),field(r,1)])
|------object(main,main,[])
- Trace:
|------'Time: 0, Object: main, Task: 0:main'
```

Figure 6: A correct execution for the running example

In our running example, the trace corresponding to execution 1 is shown in Figure 5. Let us briefly explain the diagram and the relations among the diagram, the code of the program (Figure 3) and the final state computed for execution 1 (Figure 4 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 6 and 7 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).

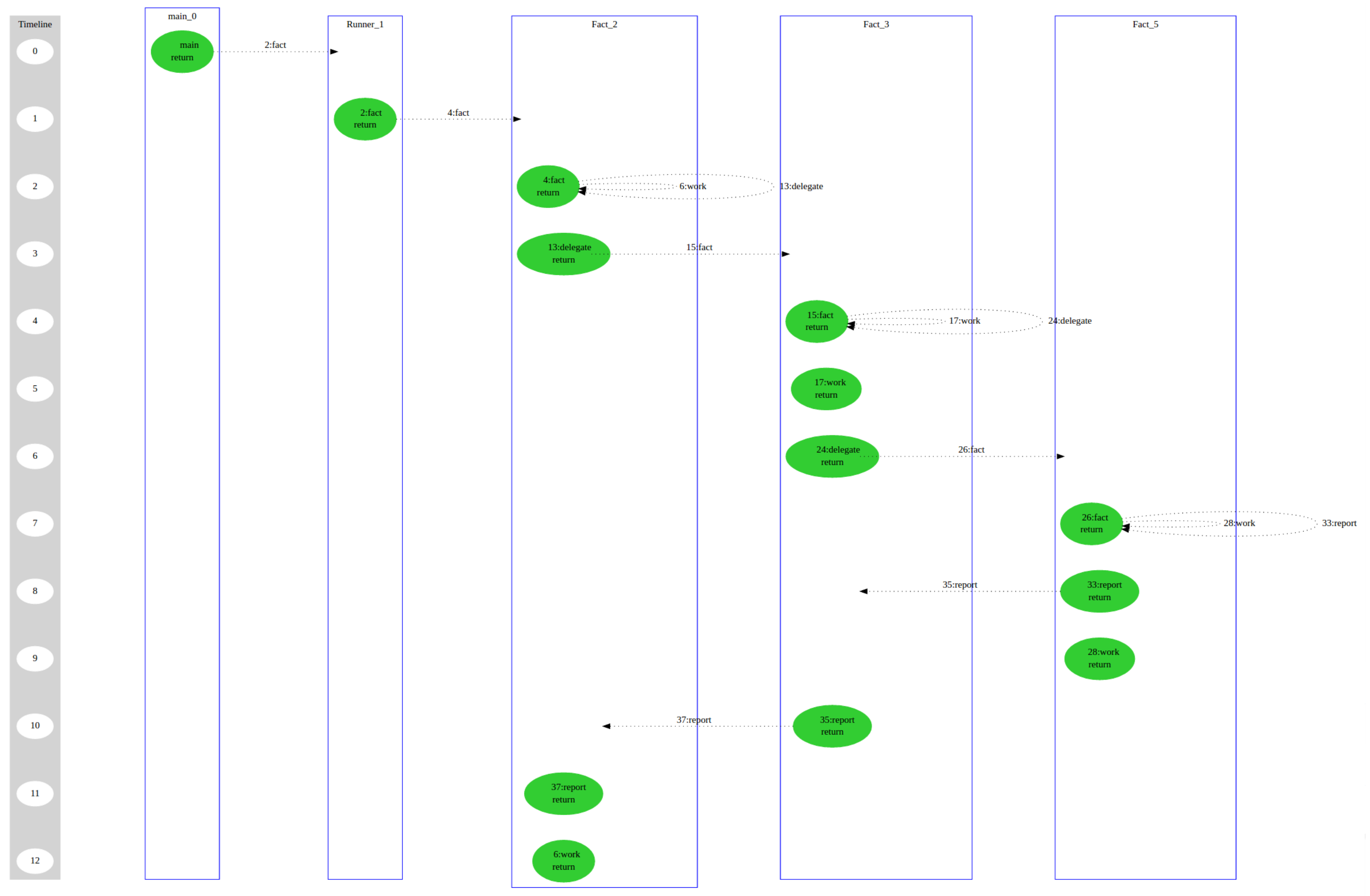
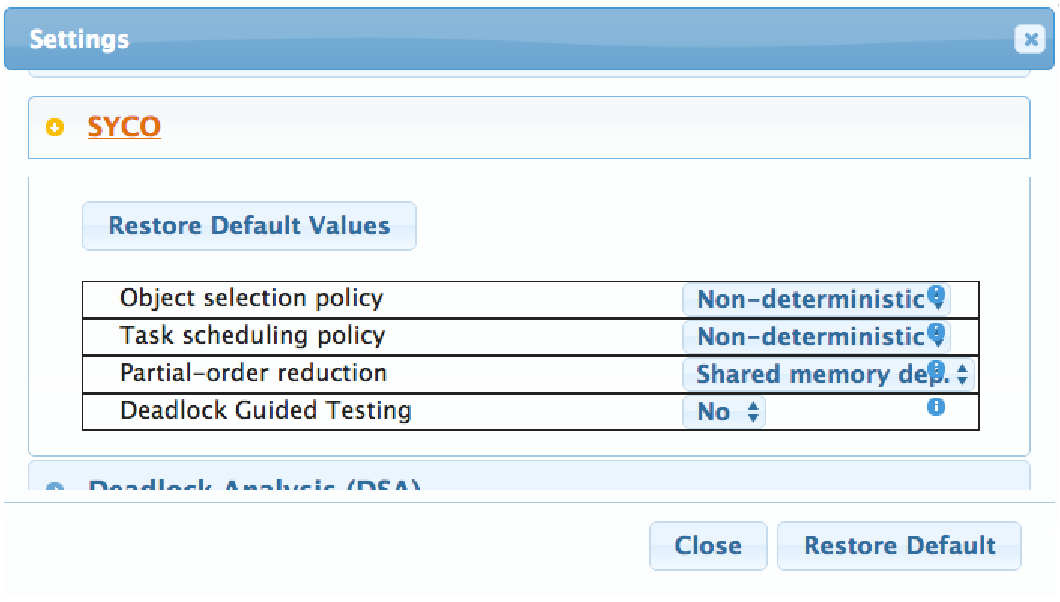


Figure 7: Diagram of correct execution for the running example



Object selection policy	Non-deterministic
Task scheduling policy	Non-deterministic
Partial-order reduction	Shared memory dep.
Deadlock Guided Testing	No

Figure 8: The SYCO parameters

2.3 Parameters of SYCO

Up to now we have executed SYCO with default parameters. Pressing button Settings at the toolbar shows the parameters window, which allows to configure the available parameters for each application. Figure 8 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 [1]). 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 three levels of precision, Naive dep. approx., Shared memory dep. (by default) and Exact dep.. Option Naive dep. approx. only applies the POR object selection in [1] based on stability, whereas option Shared memory dep. over-approximates the dependencies based on shared-memory accesses of [1]. Finally, Exact dep. applies a recent and yet experimental DPOR technique which detects dynamically context-sensitive and exact dependencies. In the example, 8 executions are obtained with POR based on shared-memory dependencies, whereas 18 are obtained with the Naive POR and 280 if POR is disabled. Using our new technique to detect exact dependencies we just get 2 executions. 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 2.4 above.

2.4 Deadlock-guided testing with SYCO

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

```
{\\main block
  DB db = new DBImp(DataSomething);
  Worker w = new WorkerImp();
  db!register(w);
  w!work(db);
}
class DBImp(Data dt) implements DB{
  Worker cl = null;
  void register(Worker w){
    Fut⟨Int⟩ f = w!ping(5);
    if (f.get == 5) cl = w;
  }
  Data getData(Worker w){
    if (cl == w) return data;
    else return DataNull;
  }
}// end class DBImp
class WorkerImp() implements Worker{
  Data data;
  Unit work(DB db){
    Fut⟨Data⟩ f = db!getData(this);
    data = f.get;
  }
  Int ping(Int n){return n;}
}// end of class WorkerImp
```

Figure 9: An example with deadlock

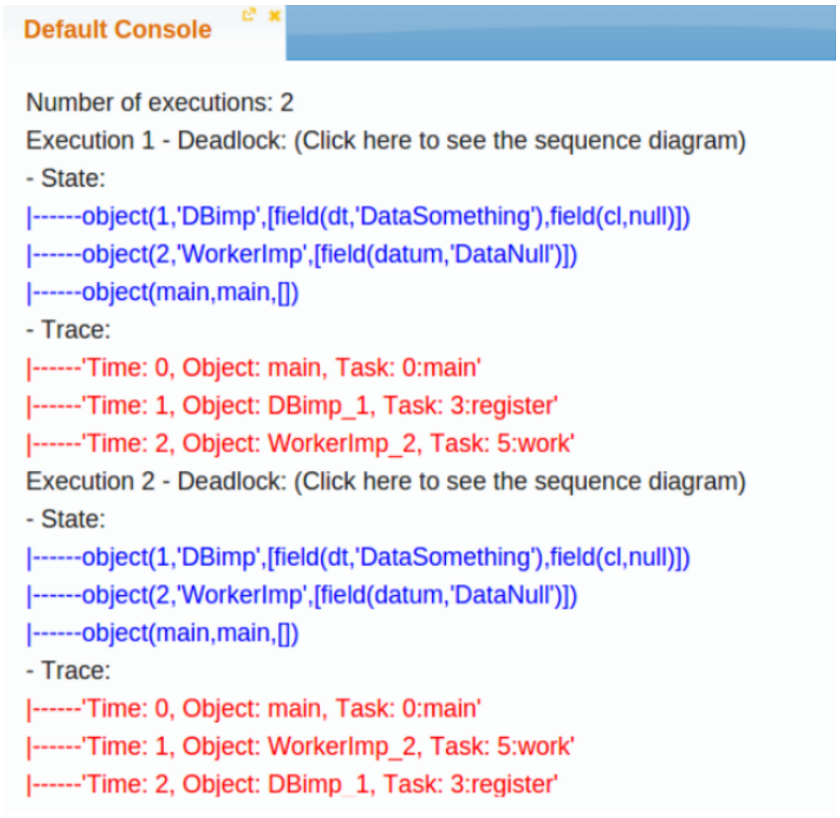
`Guided Testing` for our running example, we get printed `Number of executions: 0` as result in the console, which means that the program is deadlock-free.

Let us consider the program in Figure 9 (available here) 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 `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 `getData` of the database returns its `data` 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.data` having the same value as `db.data`, (ii) with `w.data = 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 9, 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 10. Looking at the sequence diagram of the first execution (Figure 11 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 11 down) is similar but changing the execution order between tasks `3:register` and `5:work`.

3 aPET: Step by Step

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,



```
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 10: Deadlock-guided testing on the Database example

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 12.

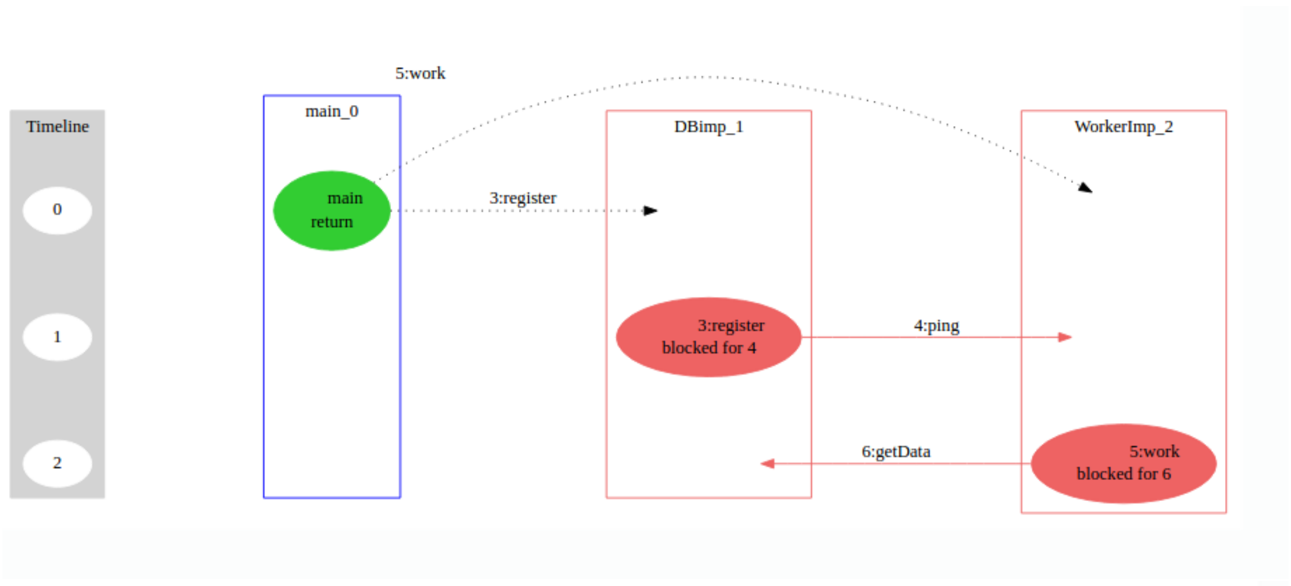
- In the `Input` section, `Args` stands for the value of the input arguments; in this case `ref(A)`, `4` and `1` are the initial values computed for the input parameters this, `n` and `h`. `State` shows the input state. It contains only one object (the caller object) of class `Runner` identified by `A`.
- 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 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.

3.1 Parameters of aPET

The parameters available for aPET are shown in Figure 13, 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



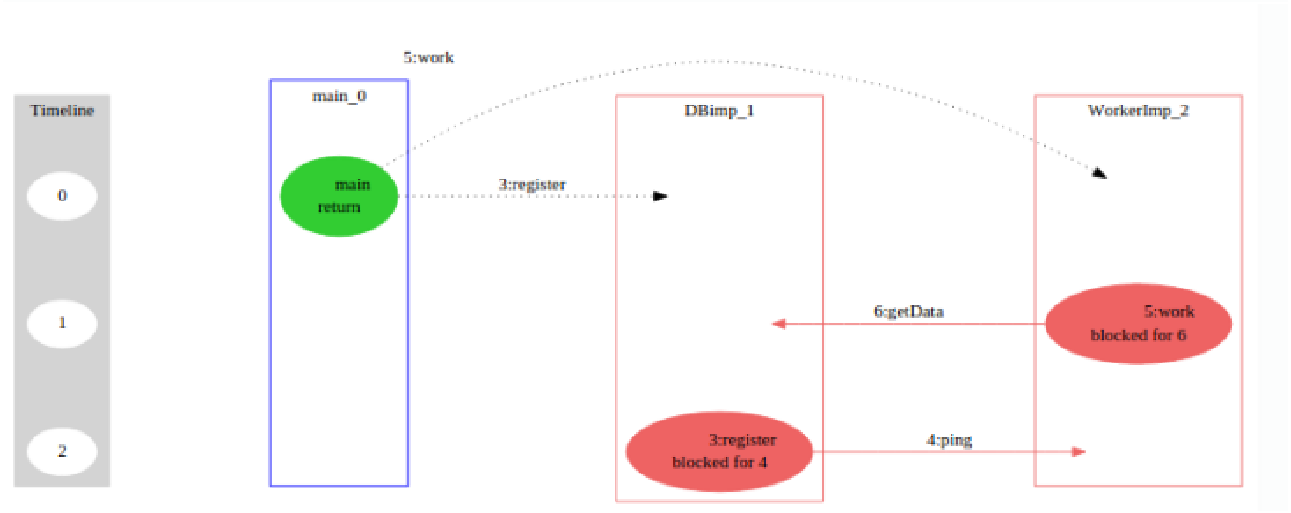
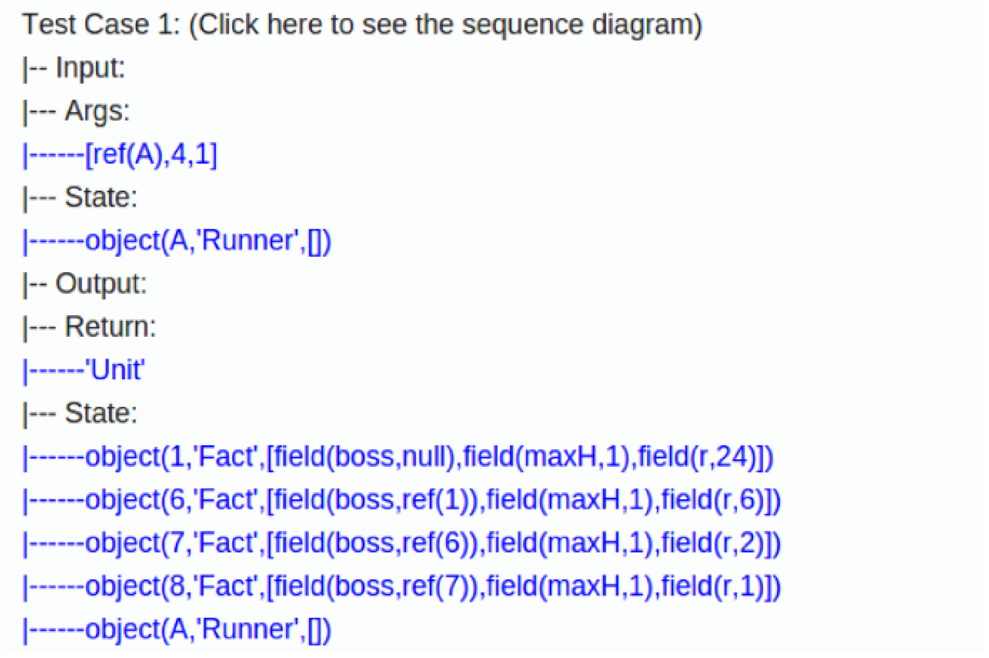


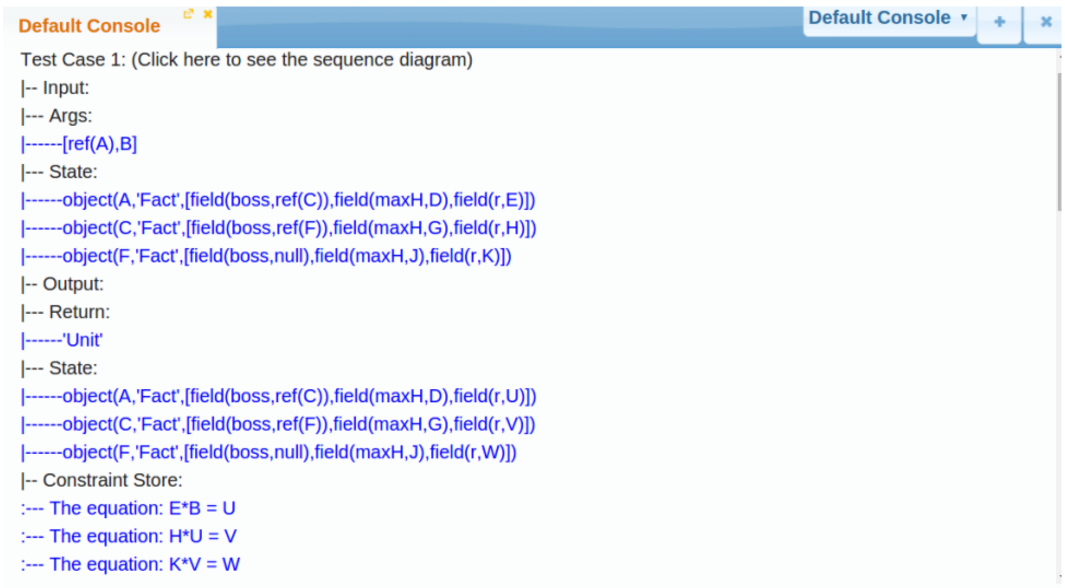
Figure 11: Sequence diagrams of deadlock executions



```
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 12: TCG with aPET for method fact

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:



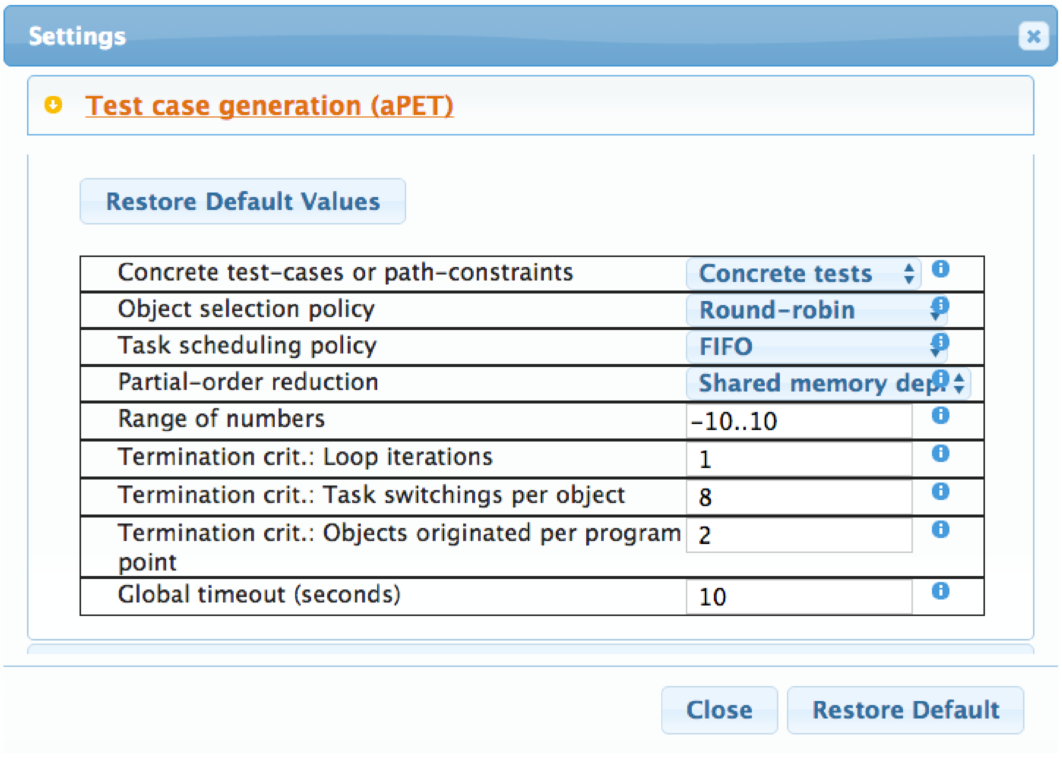
```
Test Case 1: (Click here to see the sequence diagram)
|-- Input:
|--- Args:
|------[ref(A),B]
|--- State:
|------object(A,'Fact',[field(boss,ref(C)),field(maxH,D),field(r,E)])
|------object(C,'Fact',[field(boss,ref(F)),field(maxH,G),field(r,H)])
|------object(F,'Fact',[field(boss,null),field(maxH,J),field(r,K)])
|-- Output:
|--- Return:
|------'Unit'
|--- State:
|------object(A,'Fact',[field(boss,ref(C)),field(maxH,D),field(r,U)])
|------object(C,'Fact',[field(boss,ref(F)),field(maxH,G),field(r,V)])
|------object(F,'Fact',[field(boss,null),field(maxH,J),field(r,W)])
|-- Constraint Store:
:--- The equation: E*B = U
:--- The equation: H*U = V
:--- The equation: K*V = W
```

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:

$$\begin{array}{l} r^A_f \ \ r^A_i * B \\ r^C_f \ \ r^C_i * r^A_i * B \\ r^F_f \ \ r^F_i * r^C_i * r^A_i * B \end{array}$$

where r^o_f (resp. r^o_i) 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



Concrete test-cases or path-constraints	Concrete tests
Object selection policy	Round-robin
Task scheduling policy	FIFO
Partial-order reduction	Shared memory dep.
Range of numbers	-10..10
Termination crit.: Loop iterations	1
Termination crit.: Task switchings per object	8
Termination crit.: Objects originated per program point	2
Global timeout (seconds)	10

Figure 13: The aPET parameters

it is given in the format *Min..Max*. This option is only applicable when that concrete test-cases are 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 2.3). 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 2. E.g., in the first test case (see Figure 14), 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 5.

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

4 Further Reading

The technical details which describe the dynamic and static methods used by `SYCO` and `aPET` can be found in the following papers, which have been published along the duration of the ENVISAGE project:

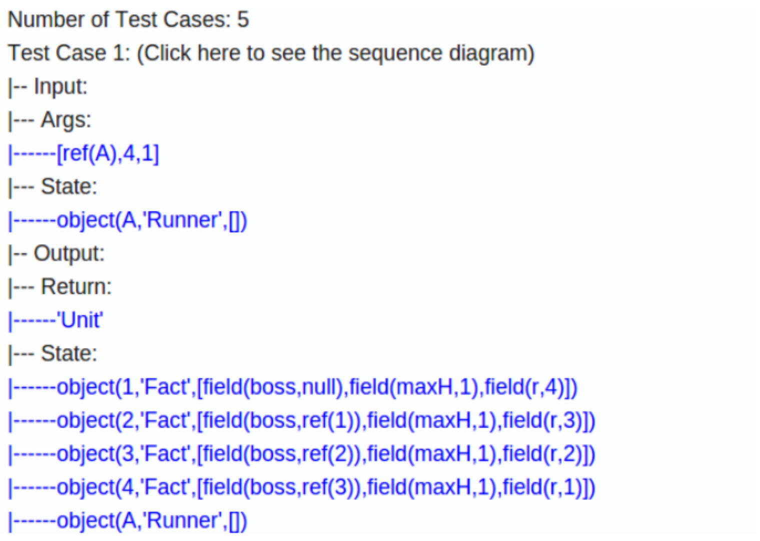

```
Number of Test Cases: 5
Test Case 1: (Click here to see the sequence diagram)
|-- Input:
|--- Args:
|------[ref(A),4,1]
|--- State:
|------object(A,'Runner',[])
|-- Output:
|--- Return:
|------'Unit'
|--- State:
|------object(1,'Fact',[field(boss,null),field(maxH,1),field(r,4)])
|------object(2,'Fact',[field(boss,ref(1)),field(maxH,1),field(r,3)])
|------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 14: First test case obtained with LIFO scheduling

- **SFM'14** [3]: 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** [1]: This work presents novel POR mechanisms and strategies for effectively testing `ABS` models.
- **ATVA'15** [2]: This paper extends the approach for dynamic testing of [1] to the context of static testing and test case generation.
- **STTT'15** [10]: 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** [4]: This tool demonstration paper overviews the `SYCO` tool for testing systematically `ABS` models.
- **iFM'16** [?]: 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.

References

[1] 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.

[2] 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.

[3] 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-Verlag, 2014.

[4] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A systematic testing tool for concurrent objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *25th International Conference on Compiler Construction (CC'16)*, pages 269–270. ACM, 2016.

[5] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[6] Javier Esparza. Model checking using net unfoldings. *Sci. Comput. Program.*, 23(2-3):151–195, 1994.

[7] 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.

[8] S. Genaim and J. Doménech. The EasyInterface Framework, 2015. http://github.com/abstools/easyinterface.

[9] 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.

[10] 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.