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.

Categories and Subject Descriptors D1.3 [*Programming Techniques*]: Concurrent Programming; D2.5 [*Testing and Debugging*]: [testing tools, systematic execution]

Keywords systematic testing, concurrency, concurrent objects, software testing, partial-order reduction

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]. 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. Task scheduling is non-preemptive, i.e., 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 = ob!m() creates an asynchronous task to execute method m on object 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. It not, the object lock is released and the task suspends until the value of f is ready. In contrast, the instruction v = f.**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
```

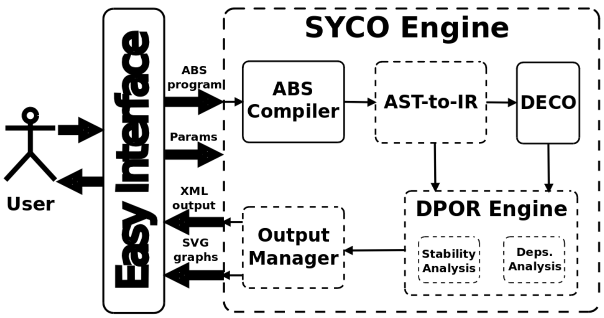
The main method creates the two objects and invokes methods register and work resp. The work method of the worker simply accesses the database (invoking asynchronously method getD) and then blocks until it gets the result, which is assigned to its data field. The register method of the database, first 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.data = db.data , (ii) with w.data = **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.

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 *EasyInter-*

face [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* 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, whose output is used by 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`. 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 [10] 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.

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. 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 main→register→work and main→work→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, 10], the remaining 4 executions can be grouped in two 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 *CHESS* [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.

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] 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.

[4] E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. Technical report, 2015.

[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. http://github.com/abstools/easyinterface.

[8] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMCO'10*, LNCS 6957, pp. 142-164. Springer, 2012.

[9] M. Musuvathi and S. Qadeer. Concurrency Unit Testing with CHESS. Tech. Report MSR-TR-2008-04, Microsoft Research, January 2008.

[10] D. Marinov G. Agha S. Lauterburg, R. K. Karmani. Basset: A Tool for Systematic Testing of Actor Programs. In *Proceedings of FSE 2010*, pages 363–364. ACM, 2010.