Deadlock Guided Testing in CLP Technical Report (including proofs)

ELVIRA ALBERT, MIGUEL GÓMEZ-ZAMALLOA and MIGUEL ISABEL
DSIC, Complutense University of Madrid (UCM), E-28040 Madrid, Spain

Static deadlock analyzers might be able to verify the absence of deadlock. However, they are usually not able to detect its presence. Also, when they detect a potential deadlock cycle, they provide little (or even no) 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. The asynchronous program is first translated into a CLP-version so that the whole combined approach is carried out by relying on the inherent backtracking mechanism and constraint handling of CLP. When the program features a deadlock, our combined use of analysis and testing provides an effective technique to catch deadlock traces. While if the program does not have deadlock, but the analyzer inaccurately spotted it, we might be able to prove deadlock freedom.
KEYWORDS: Testing, Deadlock detection, Symbolic execution, Test case generation, Verification, Deadlock analysis

1 Introduction

A deadlock occurs when a concurrent program reaches a state in which one or more tasks are waiting for each other termination and none of them can make any progress. Deadlocks are one of the most common errors in concurrent programming and, thus, a main goal of verification and testing tools is, respectively, proving deadlock freedom and *deadlock detection*. We consider an *asynchronous* language which allows spawning asynchronous tasks at distributed locations, with no shared memory among them, and with an operation for *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. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks or months after releasing the application. This aspect of static analysis is especially important in security assurance – security attacks try to exercise an application in unpredictable and untested ways. However, due to the use of approximations, most static analyses can only verify the absence of deadlock but not its presence, i.e., they can produce false positives. Moreover, when a potential deadlock is detected, state-of-the-art analysis tools (Flores-Montoya et al. 2013; Giachino E. and C. 2014; Giachino E. and M. 2016) provide little (and often no) 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. In dynamic testing, the application is executed for concrete input values, while in static testing it is executed symbolically (i.e., without any knowledge on the input variables). Since a deadlock can manifest only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings. The primary advantage of *systematic testing* (Christakis et al. 2013; Sen and Agha 2006) for deadlock detection is that it can provide the detailed deadlock trace. There are two shortcomings though: (1) Although recent research tries to avoid redundant exploration as much as possible using Partial Order Reduction (POR) techniques (Abdulla et al. 2014; Albert et al. 2014; Christakis et al. 2013; Flanagan and Godefroid 2005), the search space (even without redundancies) can be huge. This is a threat to the application of systematic testing in concurrent programming. (2) In dynamic testing, one can only guarantee deadlock freedom for finite-state terminating programs (i.e., terminating executions for concrete inputs). In static testing, one needs to assume some termination criteria (e.g., loops can only be executed a fixed number of iterations) and, thus, deadlock freedom can be ensured for the considered termination criterion (e.g., the program is deadlock free provided its loops are unrolled the fixed number of iterations).

This article proposes a seamless combination of static analysis and testing for effective deadlock detection as follows: an existing static deadlock analysis (Flores-Montoya et al. 2013) 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). When the program features a deadlock, our combined use of analysis and testing provides an effective technique to catch deadlock traces. The development of such combined framework requires: (1) the use of a backtracking mechanism to prune those paths that are deadlock free as soon as possible, and keep on exploring the search tree in order to find deadlock paths (if there are), and (2) handling the constraints that describe the potential deadlock cycles as well as the path conditions. Our CLP-based framework benefits from the inherent mechanisms of constraint logic programming for achieving the above requirements as follows: the asynchronous program is translated into a CLP equivalent program so that the whole combined approach is carried out by relying on the inherent backtracking mechanism and constraint handling applied on the CLP-translated program.

1.1 Summary of Contributions

In summary, the main contributions of the article are the following:

1. We extend a standard semantics for asynchronous programs with information about the task interleavings made and the status of tasks.
2. We provide a formal characterization of *deadlock state* which can be checked along the execution and allows us to early detect deadlocks.
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 to guide the testing process towards paths that might lead to a deadlock cycle while discarding deadlock-free paths. Our method can be used both for static and dynamic testing.
4. We introduce several deadlock-based testing criteria to find the first deadlock trace, a representative trace for each deadlock cycle, or all deadlock traces.
5. We implement our methodology in the SYCO/aPET testing system and perform a thorough experimental evaluation on some classical examples.

This article is a revised and extended version of a conference paper that appeared in the proceedings of iFM'16 (Albert et al. 2016a). There are two fundamental contributions w.r.t. (Albert et al. 2016a). The first one is that we extend our approach and our implementation to the *static* testing setting, i.e., we can apply symbolic execution without any knowledge on the input data in order to find deadlock traces. In contrast, (Albert et al. 2016a) is defined for *dynamic* testing which requires the use of particular values for the input data. This has a clear theoretical interest as it generalizes the previous work to a static setting. Besides, the practical impact of such extension is important as now we can use it to prove deadlock freeness, i.e., if we are able to symbolically execute the whole program without finding any deadlock trace, nor termination problem, then the program is proven to be deadlock free. Note that the use of symbolic execution requires the use of termination strategies, since when the program contains loops or recursion, we might need a termination criterion (e.g., one that allows unfolding loops a constant number of times k) to guarantee termination of the process. In such case, our combined approach would prove deadlock freeness up for the selected termination criterion. The second fundamental contribution is that our whole approach is formalized in CLP, while (Albert et al. 2016a) was developed on the imperative language. This gives us the advantages mentioned above, in particular the implicit support for backtracking greatly simplifies the definition of the symbolic testing framework.

1.2 Organization of the Article

The remainder of the article is organized as follows. Section 2 presents the syntax and semantics of the asynchronous programs that we consider. Essentially, it includes three instructions for concurrency: create a new concurrency unit, spawn a task on a concurrency unit, and block the execution of a concurrency unit until a spawned task has finished its execution. Section 3 motivates our work by means of an example that will be used throughout the article to explain the different concepts and techniques.

Section 4 recaps the CLP-based testing framework that is the basis of our combined approach. We will first review the translation of standard imperative programs into CLP-equivalent ones. Next we describe how to perform systematic testing on the concurrent CLP-translated programs. This already requires the use of backtracking to exercise all possible task interleavings in a non-deterministic way. Finally, we leverage the testing framework to the static context in which one does not assume any information on the input data.

Section 5 contains the main contributions of this work. We first introduce the interleavings table in which we store the decisions about task interleavings made during the execution. Using the interleavings table, we provide a technique for the early detection of deadlock states during the execution. Finally, we will present our combined testing and static analysis framework for detecting deadlock traces.

Section 6 describes our implementation and experimental evaluation. Section 7 reviews related work and we conclude in Section 8 concludes and points out several directions for future research.

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 processor, including its own, and synchronize with the termination of tasks. The language uses *future variables*(de Boer et al. 2007) for synchronizing program execution with the completion of asynchronous tasks. A future variable acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete. When the future variable is ready, the result can be retrieved. An asynchronous call $\mathsf{m}(\bar{\mathsf{z}})$ spawned at location x is associated with a future variable f as follows $\mathsf{f} = \mathsf{x} \ ! \ \mathsf{m}(\bar{\mathsf{z}})$. Instruction $\mathsf{r} = \mathsf{f.get}$ allows blocking the execution until the task executing m that is associated to f terminates, and it retrieves the result in r. The declaration of a future variable is as follows $\mathsf{Fut} < \mathsf{T} > \mathsf{f}$, where T is the type of the result r. When a task completes, 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* that can be dynamically created using the instruction **new**. For instance, the instruction b=**new** *Location*(); creates a distributed location of type *Location* which is referenced by b. The program consists of a set of classes that define the types of locations, each of them defines a set of fields and methods of the form $M{::=}T\ m(\bar{T}\ \bar{x})\{s\}$, where statements s take the form $\mathsf{s}{::=}\mathsf{s};\mathsf{s} \mid \mathsf{x}{=}\mathsf{e} \mid$ **if** e then s **else** s $\mid$ **while** e **do** s $\mid$ **return** $\mathsf{x}; \mid \mathsf{b}{=}$**new** $\mid \mathsf{f} = \mathsf{x} \ ! \ \mathsf{m}(\bar{\mathsf{z}}) \mid \mathsf{x} = \mathsf{f.get}$. All

$$(\textsc{mstep})\ \frac{\begin{array}{c} \mathit{selectTask}(S) = \mathit{tk},\ S = \mathit{loc}(o, \bot, h, \mathcal{Q} \cup \{\mathit{tk}\}) \cdot S' \\ \mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tk}\}) \cdot S' \overset{o\cdot \mathit{tk}}{\leadsto^*} S'' \end{array}}{S \xrightarrow{o\cdot \mathit{tk}} S''}$$

$$(\textsc{newloc})\ \frac{\mathit{tk} = \mathit{tsk}(\mathit{tk}, m, l, \mathit{pp} : \mathsf{x} = \mathsf{new\ D}; \mathsf{s}),\ \mathrm{fresh}(o_1),\ h_1 = \mathit{newheap}(D),\ l_1 = l[x \to o_1]}{\mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tk}\}) \leadsto \mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tsk}(\mathit{tk}, m, l_1, \mathsf{s})\}) \cdot \mathit{loc}(o_1, \bot, h_1, \{\})}$$

$$(\textsc{async})\ \frac{\begin{array}{c} \mathit{tk} = \mathit{tsk}(\mathit{tk}, m, l, \mathit{pp} : \mathsf{y} = \mathsf{x}\ !\ \mathsf{m_1}(\bar{\mathsf{z}}); \mathsf{s}),\ l(x){=}o_1,\ \mathrm{fresh}(\mathit{tk}_1),\ \mathrm{fresh}(f_1), \\ l_1{=}\mathit{buildLocals}(\bar{z}, m_1, l),\ l' = l[y \to f_1] \end{array}}{\begin{array}{c} \mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tk}\}) \cdot \mathit{loc}(o_1, \_, \_, \mathcal{Q}_1) \leadsto \mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tsk}(\mathit{tk}, m, l', \mathsf{s})\})\cdot \\ \mathit{loc}(o_1, \_, \_, \mathcal{Q}_1 \cup \{\mathit{tsk}(\mathit{tk}_1, m_1, l_1, \mathit{body}(m_1))\}) \cdot \mathit{fut}(f_1, o_1, \mathit{tk}_1, \mathit{ini}(m_1), \bot) \end{array}}$$

$$(\textsc{return})\ \frac{\mathit{tk} = \mathit{tsk}(\mathit{tk}, m, l, \mathit{pp} : \mathsf{return\ x};\ \mathsf{s}),\ l(x) = v}{\mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tk}\}) \cdot \mathit{fut}(f, \_, \mathit{tk}, \_, \bot) \leadsto \mathit{loc}(o, \bot, h, \mathcal{Q}) \cdot \mathit{fut}(f, \_, \mathit{tk}, \_, v)}$$

$$(\textsc{get1})\ \frac{\mathit{tk} = \mathit{tsk}(\mathit{tk}, m, l, \mathit{pp} : \mathsf{x}\ =\ \mathsf{y.get}; \mathsf{s}),\ l(y) = f,\ \mathit{fut}(f, \_, \_, \_, v) \in \mathtt{Fut},\ l_1 = l[x \to v]}{\mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tk}\}) \leadsto \mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tsk}(\mathit{tk}, m, l_1, \mathsf{s})\})}$$

$$(\textsc{get2})\ \frac{\mathit{tk}{=}\mathit{tsk}(\mathit{tk}, m, l, \mathit{pp}{:}\mathsf{x} = \mathsf{y.get}; \mathsf{s}),\ l(y) = f,\ \mathit{fut}(f, \_, \_, \_, \bot) \in \mathtt{Fut}}{\mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tk}\}) \leadsto \mathit{loc}(o, \mathit{tk}, h, \mathcal{Q} \cup \{\mathit{tk}\})}$$

Fig. 1. Macro-Step Semantics of Asynchronous Programs

methods must return something, hence **void** methods are by-default expressed as **int** methods returning 0. For the sake of generality, the syntax of expressions e and types T is left open.

Figure 1 presents the semantics of the language. A *state* or *configuration* is a set of locations and future variables $\mathit{loc}_0 \cdots \mathit{loc}_n \cdot \mathit{fut}_0 \cdots \mathit{fut}_m$. A *location* is a term $\mathit{loc}(o, \mathit{tk}, h, \mathcal{Q})$ where o is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\bot$ 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 $\mathit{fut}(\mathit{id}, o, \mathit{tk}, m, r)$ where id is a unique future variable identifier, o is the location identifier that executes the task tk awaiting for the future, m is the initial program point of tk and r is the return value of the task tk, or $\bot$ if tk has not finished. A *task* is a term $\mathit{tsk}(\mathit{tk}, m, l, s)$ 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. We assume that the execution starts from a main method without parameters included in a Main class. The initial state is $S{=}\{\mathit{loc}(0, 0, \bot, \{\mathit{tsk}(0, \mathit{main}, l, \mathit{body}(\mathit{main}))\}$ 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 $\bot$ is the empty heap. $\mathit{body}(m)$ 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 (Sen and Agha 2006) (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 non-deterministically an available task from one *active* location in the state (i.e., a location with a non-empty task buffer). The transition $\rightsquigarrow$ 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.

RETURN: When **return** is executed, the lock is released and will never be taken again by that task. Consequently, that task is *finished* and removed from the task buffer. GET2: A y.get instruction waits for the future variable without yielding the lock, i.e., it blocks the execution of the location until the task that is awaiting is finished. Then, when the future is ready, GET1 allows continuing the execution.

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 $\nexists\ S_{n+1} \neq S_n$ such that $S_n \longrightarrow S_{n+1}$. 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 complete derivations starting at S. 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 $\rightsquigarrow$). The systematic exploration of $exec(S)$ thus corresponds to the standard systematic testing setting in which all possible explorations are performed without eliminating any redundancy.

3 Motivating Example

Our running example simulates a simple communication protocol among a database and n workers. Our implementation in Figure 2 has three classes, a Main class which includes the main method, and classes Worker and DB implementing the workers and the database, respectively. The main method just calls method simulate with the number of workers to create in its parameter (in this case 1). Method simulate creates the database and the n workers, and invokes methods register and work on each of them, respectively. The work method of a 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 registers the provided worker reference adding it to its clients list field. In case checkOn is true, before adding the worker, it makes sure that the worker is online. This is done by invoking asynchronously method ping with a concrete value and blocking until it gets the result with the same value. Method getData of the database returns its data field if the caller worker is registered, otherwise it returns **null**.

```
 1 class Main{
 2   main(){
 3     this!simulate(1);
 4     return 0;
 5   }
 6   simulate(int n){
 7     DB db = new DB();
 8     while (n > 0){
 9       Worker w = new Worker();
10       db!register(w);
11       w!work(db);
12       n = n−1;
13     }
14     return 0;
15   }
16 }// end of class Main
17
18 class DB{
19   Data data = ...;
20   List<Worker> clients;// Empty list
21   Bool checkOn = true;
22   int register(Worker w){
23     if (checkOn){
24       Fut⟨int⟩ f = w!ping(5);
25       if (f.get == 5) add(clients,w);
26     } else add(clients,w);
27     return 0;
28   }
29   Data getData(Worker w){
30     if (contains(w,clients)) return data;
31     else return null;
32   }
33 }// end of class DB
34 class Worker{
35   Data data;
36   int work(DB db){
37     Fut⟨Data⟩ f = db!getData(this);
38     data = f.get;
39     return 0;
40   }
41   int ping(int n){return n;}
42 }// end of class Worker
```

Fig. 2. Working example: Communication protocol among a DB and n workers.

Depending on the sequence of interleavings, the execution of this program can finish:

- (I) As one would expect, 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 L25) before getD is executed. (II) happens when getD is executed before the assignment at L25. A deadlock is produced if both register and work start executing before getD and ping.

Figure 3 shows the derivation tree computed by a systematic testing of the main method. Derivations that contain a dotted node are not deadlock, while those with a black 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. All derivations which can be found in this program start by executing methods main and simulate, one after the other. The 1st and 2nd derivations finish with the expected output state (scenario I above), the 3rd and 4th branches are deadlocks (scenario III), and finally, the 5th and 6th derivations correspond to scenario II. Let us see two selected branches in detail. In the first derivation, the third macro-step executes register on location db and then ping on w. It is clear that location db will not deadlock in this derivation, since the get at line 25 will succeed and location w will also be able to complete its tasks, namely, the get at line 38 will succeed because the task in getData will eventually finish as its location is not blocked.

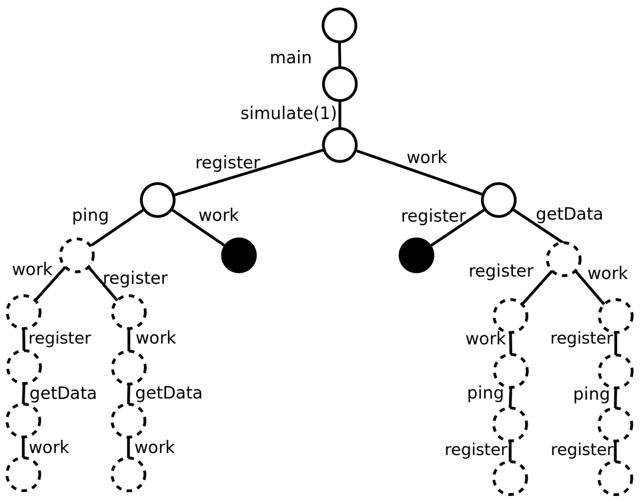

Fig. 3. Systematic testing tree of working example

However, in the third branch, we first select register (and block database waiting for the termination of ping), and then we select work (blocking w waiting for the termination of getData). The get at line 38 will never succeed since it is awaiting for the termination of a task of a blocked location. Hence, we have a deadlock. Let us outline five states of this derivation:

$S_0 \equiv loc(0,0,..,\{tsk(0,main,..)\}) \cdot fut(f_0,0,0,2,\bot) \xrightarrow{0,0}$
$S_1 \equiv loc(0,0,..,\{tsk(1,simulate,..)\}) \cdot fut(f_0,0,0,2,0) \cdot fut(f_1,0,1,6,\bot) \xrightarrow{0,1}$
$S_2 \equiv loc(0,0,...) \cdot loc(db,\bot,h_{db},\{tsk(2,reg,..)\}) \cdot loc(w,\bot,h_w,\{tsk(3,wo,..)\}) \cdot$
$\quad \cdot fut(f_1,0,1,6,0) \cdot fut(f_2,db,2,22,\bot) \cdot fut(f_3,w,3,36,\bot) \xrightarrow{db,2}$
$S_3 \equiv loc(db,2,..,\{tsk(2,reg,f_4.\mathsf{get})\}) \cdot loc(w,\bot,..,\{tsk(4,ping,..),..\}) \cdot fut(f_4,w,4,41,\bot) \xrightarrow{w,3}$
$S_4 \equiv loc(w,3,..,\{tsk(3,wo,f_5.\mathsf{get})\}) \cdot loc(db,2,..,\{tsk(5,getData,..),..\}) \cdot fut(f_5,db,5,29,\bot)$

The third state S_2 is obtained after executing methods main and simulate. It contains the initial location 0, the two locations created at lines 7 and 9 resp., and the two tasks created at lines 10 and 11 resp. in the corresponding 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 register has been selected and fully executed (we have shortened the name of the methods, e.g., reg for register). Let us observe the addition of the future variable created at L24 at S_3. In S_4 we have executed task work in the worker and added a new future variable. This state is already deadlock as we will see in Section 5.2. In the rest of the paper, we use the location and task names instead of numeric identifiers for clarity.

4 A CLP-based Testing Framework

In this section we review the CLP-based testing framework for concurrent asynchronous programs developed in previous work (namely it comprises the following three articles: (Gómez-Zamalloa et al. 2009; Gómez-Zamalloa et al. 2010; Albert et al. 2012)). The framework consists in two main phases. (1) The source asynchronous program is transformed into an equivalent CLP program, which includes calls to specific operations to handle non-native CLP features that include concurrency builtins for asynchronous calls, task scheduling and synchronization. The *CLP-transformed* program can be executed in CLP as long as such operations are encoded in CLP. (2) Systematic and symbolic execution is performed using CLP's execution mechanism over the CLP-transformed program. The use of CLP as the enabling technology for the testing framework has the following important advantages that will become apparent throughtout this section:

- CLP is well known to be a very appropriate paradigm to do meta-programming.
- CLP's native backtracking mechanism allows systematically executing the program in order to try out all non-deterministic task interleavings.
- CLP's backtracking and constraint solving mechanisms allow performing symbolic execution and test case generation (Gómez-Zamalloa et al. 2010).

4.1 CLP-translated Programs

The translation of imperative, object-oriented and concurrent programs into equivalent CLP-translated programs has been subject of previous work (see, e.g.,(Gómez-Zamalloa et al. 2009; Albert et al. 2007; Albert et al. 2012)). Therefore, we only state the features of the translated programs without going into deep details of how the translation is done:

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

1. In and Out are, respectively, the (possibly empty) list of input and output arguments.
2. S_{in} and S_{out} are, respectively, the input and (possibly modified) output states.
3. If m_i is defined by more than one rule, then g is the constraint that guards the execution of each rule, i.e., it must hold for the execution of the rule to proceed.
4. $b_1, \ldots, b_j$ is a sequence of instructions including arithmetic operations, calls to other predicates, field accesses and concurrent operations as defined in Figure 4. As usual, an SSA transformation is performed (Cytron et al. 1991).

Specifically, CLP-translated programs adhere to the grammar in Figure 4. 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*, *PP*, *MSig*, *FSig* and *C* denote, respectively, the set of predicate names, numbers, variables, source program-points, method signatures, field signatures and class names. A clause indistinguishably defines either a method which appears in the original source program (*MSig*), or an additional predicate which corresponds to an intermediate block in the control flow graph of original program (*Block*). A field signature *FSig* contains the field name. 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). Object references are written as terms of the form $r(Ref)$ or $null$. As expected, the operation `new_object`$(S_{in}, C, Ref, S_{out})$ creates a new object of class C in state S_{in} and returns its assigned reference Ref and the updated state S_{out}; `get_field`$(S, Ref, FSig, V)$ retrieves in variable V the value of field $FSig$ of the object referenced by Ref in the state S; and, `set_field`$(S_{in}, Ref, FSig, V, S_{out})$ sets the field $FSig$ of the object referenced by Ref in S_{in} to V and returns the modified state S_{out}. Implementations of these operations in CLP can be found in (Gómez-Zamalloa et al. 2010). Operations `async`/6, `get`/6 and `return`/3 simulate in CLP the asynchronous concurrency and are explained later.

$$\begin{array}{rl}
Clause ::= & Pred(Args_{in},Args_{out},S_{in},S_{out})\ \text{:-}\ [G,]B_1,B_2,\ldots,B_n. \\
G ::= & Num^*\ ROp\ Num^* \mid Ref_1^*\ \backslash{==}\ Ref_2^* \\
B ::= & Var\ \#{=}\ Num^*\ AOp\ Num^* \\
 & \mid Pred(Args_{in},Args_{out},S_{in},S_{out}) \\
 & \mid \texttt{new\_object}(S_{in},C^*,Ref^*,S_{out}) \mid \texttt{async}(S_{in},Ref^*,Call,Var,PP,S_{out}) \\
 & \mid \texttt{return}(S_{in},Var,S_{out}) \mid \texttt{get}(S_{in},Var,Var,PP,Call,S_{out}) \\
 & \mid \texttt{get\_field}(S_{in},Ref^*,FSig,Var) \mid \texttt{set\_field}(S_{in},Ref^*,FSig,Data^*,S_{out})
\end{array}$$

$$\begin{array}{rlrl}
Call ::= & Pred(Args_{in},Args_{out}) & Pred ::= & Block \mid MSig \\
Args ::= & [\,] \mid [Data^* \mid Args] & ROp ::= & \#{>} \mid \#{<} \mid \#{>=} \mid \#{=<} \mid \#{=} \mid \#\backslash{=} \\
Data ::= & Num \mid Ref & AOp ::= & + \mid - \mid * \mid / \mid mod \\
Ref ::= & null \mid r(Var) & S ::= & Var
\end{array}$$

Fig. 4. Syntax of CLP-translated programs

Figure 5 shows (a simplified and pretty-printed version of) the CLP-translated program of our working example. Let us observe the following:

- Conditional statements and loops in the source program are transformed into guarded rules and recursion in the CLP program, respectively. E.g. the **if** of the getData method is encoded in CLP by means of predicate `if3/4`, whereas the **while** of the main method is encoded by the recursive rule `while/4`. 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.
- A global state, which includes the locations (with their fields and task queues) and the future variables, is explicitly handled. Observe that each rule includes as arguments an input and an output state (3rd and 4th arguments respectively). The state is carried along the execution being used and transformed by the corresponding operations as a black box, therefore it is always a variable in the CLP program. The operations that modify the state are explained later.

```
main([This],0,S0,S2) :-
  async(S0,This,
        simulate([This,1],_),_,6,S1),
  return(S1,0,S2).

simulate([This,N],R,S0,S2) :-
  new_object(S0,'DB',DB,S1),
  while([N,DB],R,S1,S2).
while([N,_],0,S0,S1) :-
  N #=< 0, return(S0,0,S1).
while([N,DB],R,S0,S4) :-
  N #> 0,
  new_object(S0,'Worker',W,S1),
  async(S1,DB,
       register([DB,W],_),_,22,S2),
  async(S2,W,work([W,DB],_),_,36,S3),
  N2 #= N - 1,
  while([N2,DB],R,S3,S4).

register([This,W],R,S0,S1) :-
  get_field(S0,This,checkOn,Ch),
  if1([This,W,Ch],R,S0,S1).
if1([_,_,false],R,S0,S1) :-
  cont1([],R,S0,S1).
if1([This,W,true],R,S0,S2) :-
  async(S0,W,ping([W,5],_),F,41,S1),
  get(S1,F,FV,25,if2([This,W,FV],R),S2).
if2([_,_,FV],R,S0,S1) :-
  FV #\= 5,
  cont1([],R,S0,S1).
if2([This,W,5],R,S0,S1) :-
  get_field(S0,This,clients,Cls),
  add([Cls,W],Cls2,_,_),
  set_field(S0,This,clients,Cls2,S1),
  cont1([],R,S1,S2).
cont1([],0,S0,S1) :- return(S0,0,S1).

getData([This,W],R,S0,S1) :-
  get_field(S0,This,clients,Cls),
  contains([Cls,W],RC,_,_),
  if3([This,RC],R,S0,S1).
if3([_,false],null,S,S1) :-
  return(S0,null,S1).
if3([This,true],Data,S0,S1) :-
  get_field(S0,This,data,Data),
  return(S0,Data,S1).

work([This,DB],R,S0,S2) :-
  async(S0,DB,getData([DB],_),F,29,S1),
  get(S1,F,FV,38,cont2([FV],R),S2).
cont2([FV],FV,S0,S1) :- return(S0,FV,S1).

ping([_,N],N,S0,S1) :- return(S0,N,S1).
```

Fig. 5. CLP-translated program for the working example

- An additional predicate is produced for the continuation after a get statement. The call to such continuation predicate is included within the arguments (5th argument) of the `get`/6 operation (see e.g. the second rule of `if1/4`). This allows implementing in CLP the concurrent behavior of asynchronous programs. Namely, a task can suspend its execution due to a blocking synchronization, and afterwards, when the awaiting task has finished, the task has to resume its execution at this precise point.
- Exceptional behavior is ignored for simplicity but can be easily handled by means of additional guarded rules and an exception flag in an additional parameter (see (Gómez-Zamalloa et al. 2010) for details).
- Predicates `contains/4` and `add/4` are the CLP-translated versions of the library operations contains and add. They are basically equivalent, resp., to the classical Prolog predicates `member/2` (returning true/false instead of succeeding/failing) and `append/3` (with a unitary list in the second argument with the element to be added).The last two arguments (input and output states) are ignored.

4.2 Systematic (Concrete) Execution

The standard CLP execution mechanism suffices to execute the CLP-translated programs as long as we provide a suitable implementation of all operations that manipulate the state. Also, just by providing a CLP non-deterministic task selection function, CLP's native backtracking mechanism will allow systematically executing the asynchronous program in order to try out all non-deterministic task interleavings. In the following we define the global state and the operations to manage it.

4.2.1 The Global State

The global state carried along by the CLP-translated program is analagous to that of the language semantics of Section 2:

$$\begin{array}{rcl}
S & ::= & (\mathit{Locs}, \mathit{Futs}) \\
\mathit{Locs} & ::= & [\,] \mid [\mathit{loc}(\mathit{Num}, \mathit{Lock}, \mathit{Fields}, Q) | \mathit{Locs}] \\
\mathit{Futs} & ::= & [\,] \mid [\mathit{fut}(\mathit{Num}, \mathit{Num}, \mathit{Num}, \mathit{PP}, \mathit{RetVal}) | \mathit{Futs}] \\
\mathit{Fields} & ::= & [\,] \mid [\mathit{field}(\mathit{FSig}, \mathit{Data}) | \mathit{Fields}] \\
Q & ::= & [\,] \mid [\mathit{tsk}(\mathit{Num}, \mathit{MSig}, \mathit{Call}, \mathit{PP}, \mathit{TskInfo}) | Q] \\
\mathit{Lock} & ::= & \mathit{bot} \mid \mathit{active}(\mathit{Num}) \mid \mathit{blocked}(\mathit{Num}) \\
\mathit{RetVal} & ::= & \mathit{bot} \mid \mathit{Data} \\
\mathit{TskInfo} & ::= & \mathit{call} \mid \mathit{get}(\mathit{Var}, \mathit{Num})
\end{array}$$

The state is represented as a pair with two lists, the list of locations and the list of futures. Locations and future variables are represented as $loc/4$ and $fut/5$ terms respectively, including the same parameters as in the semantics of Section 2. Namely, locations include the location identifier and lock, the list of fields and the queue of pending tasks; and, future variables include the future identifier, the location, task identifier, initial program point and return value of the task for which the future is awaiting. The return value is *bot* if the task has not finished yet. Fields are represented as $field/2$ terms containing the field signature and its associated data. The location's lock is *bot* if the lock is free, *active*(*TkId*) if this is the active location which is executing task *TkId* (there is only one active location in the state), or *blocked*(*TkId*) if task *TkId* is holding the location's lock. Finally, the queue of pending tasks is represented as a list of $tsk/5$ terms, which includes the identifier, method signature, CLP call and initial program point of the pending task. The last parameter *TskInfo* indicates whether the task is an asynchronous call (*call*/0 term) or a resumption after a `get` statement (*get*/2 term, including the future variable identifier and CLP variable to store its value). Our $tsk/5$ terms differ from the tsk terms of the semantics of Section 2 in the following: (1) in CLP we do not need the local variables mapping (it is managed by the CLP engine); (2) the sequence of instructions is represented as a CLP call (which is a continuation predicate in the case of resumption tasks); and, (3) in CLP we add two additional parameters (the last two ones), the initial program point and the *TskInfo* term.

Example 1
Let us consider the systematic execution of the `main` method of the working example. After executing line 55, we obtain the following state: `S1 = (Locs1,Futs1)` where

```
Locs1 = [loc(0,active(1),[],[tsk(1,simulate,simulate([r(0),1],Ret),6,call)]),
         loc(1,bot,[field(data,..),field(clients,null),field(checkOn,true)],[]),
         loc(2,bot,[field(data,null)],[])], and
Futs1 = [fut(0,0,0,2,0),fut(1,0,1,6,bot)]
```

As expected, two new locations appear in the state. Neither of them is executing, so their locks are bound to *bot*; their fields have the initial values and their queues are empty since we have not performed any asynchronous call yet. Location 0 has finished task main, hence its related future variable (fut 0) has the return value 0. However, the return value of future variable 1 is *bot* since task simulate has not finished yet.

4.2.2 Distribution, Concurrency and Synchronization

Figure 6 shows the relevant part of the CLP implementation of the operations that simulate in CLP the distribution, concurrency and synchronization of our asynchronous programs. Essentially, they correspond to the rules with the same names in the semantics of Figure 1. Specifically:

- Predicate `async`/6, given a state S, builds and adds a call task with *Call* and its initial program point *PP* to the queue of location *LocId* (by means of predicate `add_task/7`); and builds and adds a $fut/5$ term linked to the new task to the list of futures (by means of `add_future/6`), resulting in state $S3$. The call to `fresh_fut/1` in L105 produces a fresh integer future identifier which is stored in the corresponding $fut/5$ term and bound to the local variable *Fut* in L106.
- Predicate `return`/3, given a state S, sets the value of the future variable corresponding to the task that has just finished with the value of local variable *Ret*, and then calls `mstep/3` to switch context to the following macro-step (if possible), resulting in the final state $S3$.
- Predicate `get`/6, given a state S, if the future variable *Fut* is ready, i.e., if it is not *bot*, (first rule) then the value of the future is bound to *FutVal* and the execution proceeds using the continuation predicate *Cont* with initial state S and final state $S2$ by means of `run_task/3`. Predicate `run_task/3` simply builds a full call putting together *Cont* and the states S and $S2$, and performs a meta-call with it.
- Predicate `mstep/3` either finishes the execution (first rule), in case there are no more tasks to be executed (`select_task/2` fails), or selects (non-deterministically) an enabled task *Task* from a non-blocked location (second rule) and continues the execution calling *Task* with initial state $S2$ and final state $S3$ (predicate `run_task/3`). Predicate `update_locks/4` sets accordingly the locks of both the location which was previously executing and the one which is about to start. The *Status* term, which is set in the calls to `mstep/3` from predicates `return`/3 (line 110) and `get`/6 (line 120), allows knowing whether the previous macro-step had ended in a **return** or in a blocked get, so that *bot* or *blocked*(*Id*) respectively is set on the previously active location's lock.

The systematic execution of an asynchronous program in our CLP-based framework is then perfomed just by launching in CLP the goal

```
:- main([r(0)],Ret,SIn,SOut).
```

```
mstep(S,_,S) :-
  \+ select_task(S,_).
mstep(S,Status,S3) :-
  select_task(S,Task),
  update_locks(S,Task,Status,S2),
  run_task(Task,S2,S3).

async(S,LocId,Call,Fut,PP,S3) :-
  add_task(S,LocId,call,Call,PP,TaskId,S2),
  fresh_fut(FutId),
  Fut = FutId,
  add_future(S2,FutId,LocId,TaskId,PP,S3).
```

```
return(S,Ret,S3) :-
  set_fut_value(S,Ret,S2),
  mstep(S2,return,S3).

get(S,Fut,FutVal,_PP,Cont,S2) :-
  ready(S,Fut), !,
  get_fut_value(S,Fut,FutVal),
  run_task(Cont,S,S2).
get(S,Fut,FutVal,PP,Cont,S3) :-
  get_this_id(S,ThisId),
  add_task(S,ThisId,get(Fut,FutVal),
          Cont,PP,_,S2),
  mstep(S2,PP:Fut:get,S3).
```

Fig. 6. CLP operations for distribution, concurrency and synchronization

with initial state `SIn = ([L0],[F0])`, being `L0 = loc(0,active(0),[],[tsk(0,main, main([],Ret),1,call)])` and `F0 = fut(0,0,0,1,bot)`. This will compute, for each derivation, the return value and final state in variables `Ret` and `SOut` respectively.

Example 2
Let us continue the systematic execution of our working example from the state in Example 1. After the call to async/6 in line 56), we obtain the state `(Locs2,Futs2)` where:

```
Locs2 = [loc(0,active(1),[],[tsk(1,simulate,simulate([r(0),1],Ret),6,call)]),
       loc(1,bot,[field(data,..),field(clients,null),field(checkOn,true)],
                          [tsk(2,register,register([r(1),r(2)],Ret2),22,call)]),
       loc(2,bot,[field(data,null)],[])], and,
Futs2 = [fut(0,0,0,2,0),fut(1,0,1,6,bot),fut(2,1,2,22,bot)].
```

We can observe how asynchronous calls do not transfer control from the caller, i.e., they are not executed when they occur, but rather added as pending tasks on the receiver objects that will eventually schedule them for execution. We now continue the execution until the end of method simulate (line 46). The **return** operation sets the value of the future and calls mstep/3 to switch context and select non-determinitically the task to be executed next. Let us assume task work is chosen and starts to execute. The call to the get at line 92 first checks whether the future variable of the call to getData is already available. Since it is not the case (i.e, *Fut* is bound to *bot*) the execution of the current task cannot proceed, therefore the corresponding get task is added to the current object (so that it can be resumed later on) and mstep/3 is called again. The current state is `(Locs3,Futs3)` where:

```
Locs3 = [loc(0,active(1),[],[]),
        loc(1,bot,[field(data,..), field(clients,null),field(checkOn,true)],
                          [tsk(2,register,register([r(1),r(2)],Ret2),22,call)
                           tsk(4,getData,getData([r(1),r(2)],Ret4),29,call)]),
        loc(2,bot,[field(data,null)],[tsk(3,work,cont2([FV],Ret3),38,get)])]) and,
Futs3 = [fut(2,1,2,22,bot),fut(3,2,3,36,bot),fut(4,1,4,29,bot),...].
```

The full derivation tree contains 6 successful branches that correspond to those of Figure 3.

4.3 Symbolic Execution

It is shown in (Gómez-Zamalloa et al. 2010) that in the context of a sequential source language without support for dynamic memory, the symbolic execution of a method using the CLP-translated program is attained by simply using the standard CLP execution mechanism just calling the corresponding predicate with all arguments being free variables. However, in the case of heap-manipulating programs, the straighforward definition of the heap-related operations one could imagine fall short to generate arbitrary heap-allocated data structures correctly handling aliasing during symbolic execution. In (Gómez-Zamalloa et al. 2010), we can find the definition of heap-related operations that allow performing symbolic execution and that can be used directly in our CLP-based framework for asynchronous programs. Given the heap-related operations and the CLP operations for concurrency, the inherent constraint solving and backtracking mechanisms of CLP provide the support for keeping 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. Thus, the symbolic execution of a method `m` in our source asynchronous program is performed in our CLP-based framework just by launching in CLP the goal

```
:- m(In,Ret,SIn,SOut).
```

with initial state `SIn = ([L0|RL],[F0])` being `L0 = loc(0,active(0),Fields,[tsk(0, m,m(In,Ret),PP,call)])`, `F0 = fut(0,0,0,PP,`*bot*`)`, input arguments `In = [r(0)|Args]`, `PP` the initial program point of method `m`, and `Args`, `Fields`, `Ret`, `RL` and `SOut` free variables. Variable `RL` represents the rest of the locations in the input state, which is unknown, and will be bound during the symbolic execution with (partially symbolic) locations. As a result, we will obtain, for each feasible execution path, the constraints on the inputs and outputs for the path, i.e., the so-called *path conditions*.

Example 3
Let us perform a symbolic execution of method getData with a completely unknown input. Specifically, the input arguments are `In = [r(0),r(W)]`, and the initial state is `SIn = ([L0|RL],[F0])` being `L0 = loc(0,active(0),[field(data,D),field(clients, Cls),...],[tsk(0,getData,getData(In,Ret),PP,call)])` and `F0 = fut(0,0,0,PP,`*bot*`)`. The following path conditions are obtained for the first three explored derivations:

1. {`Cls = [], Ret = null`}, i.e., if the clients list is empty it returns **null**,
2. {`Cls = [r(X)], X` $\neq$ `W, Ret = null`}, i.e., if the worker is not in the list it returs **null**,
3. {`Cls = [r(W)|_], Ret = D`}, i.e., if the worker is in the head of the list it returns `D`.

Due to the execution of the contains/4 predicate with a variable list, the exploration continues infinitely producing lists of increasing length with and without `r(W)` at different positions.

In general, in symbolic execution, as soon as the source program has loops or recursion whose termination depends on unknown data, the derivation tree can be infinite. It is therefore necessary to establish a termination criterion, which guarantees that the number of paths traversed remains finite, while at the same time the obtained coverage on the source program is meaningful. One of the standard termination/coverage criteria is the *loop-k* criterion, which limits to a certain threshold the allowed number of iterations on each loop (or recursive calls). E.g., in the above example, if we set $k = 1$, the exploration finishes with the three successful derivation above. Unfortunately, in the context of concurrent programs, the application of this criterion to all tasks of a state does not guarantee termination of the whole process. As studied in (Albert et al. 2015) it is required to take into account more factors that threaten termination. Namely, in symbolic execution, (1) we can switch from one task to another one an infinite number of times, and, (2) we can create an unbounded number of actors. Termination/coverage criteria taking into account those factors are defined in (Albert et al. 2015) and simply adopted in our symbolic execution framework.

Example 4
Let us now perform a symbolic execution of method `simulate` for an unknown input `n`. If we set *loop-k* with $k = 1$, we obtain seven derivations. The first one produces the condition `N #=< 0` and finishes with no workers in the output state and no executed tasks in the database. The other six derivations include the path-condition `N = 1` and correspond to those of Example 2 (see also Figure 3). In case the loop limit is $k = 2$, we obtain a total of 1321 derivations, out of which 50 of them are deadlock.

5 Deadlock-Guided Testing

As already mentioned, systematically exploring all different task interleavings of a concurrent program, even applying the most advanced techniques to eliminate redundancies (see Section 7), presents scalability problems. In the case of symbolic execution, the problem is exacerbated by the intrinsic non-determinism of symbolic execution produced in branching statements involving partially unknown data. In this section we present two complementary techniques to reduce the state space exploration when the goal is to find deadlocks, both in concrete and symbolic execution. First, Section 5.1 proposes an extension of the program state to store information about the task interleavings made and the status of tasks. Section 5.2 provides a formal characterization of deadlock states using the above extension, which can be checked during the execution allowing us to early detect deadlocks. Finally, Section 5.3 presents a new methodology to guide the exploration towards paths that might lead to deadlock (discarding deadlock-free paths), by relying on the information inferred by a static deadlock analysis.

5.1 The Interleavings Table

The interleavings table 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. Specifically, the interleavings table $t \in IT$ is defined as a mapping with entries of the form $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$, where:

- $t_{o,tk,pp}$ is a *macro-step identifier*, or *time identifier*, that includes: the identifiers of the location o and task tk that have been selected in the macro-step, and the program point pp of the first instruction that will be executed;
- n is an integer representing the clock-time when the macro-step starts executing;
- ρ is the status of the task after the macro-step and it can take the following values: (1) pp:f.get when the macro-step ended on a get on a (non-ready) future variable f at program point pp; or (2) **return** when the task finished.

As notation, we define the relation $t \in IT$ if there exists an entry $t \mapsto \langle n, \rho \rangle \in IT$, and the function $status(t, IT)$ which returns the status ρ_t such that $t \mapsto \langle n, \rho_t \rangle \in IT$. The following extensions are done to our CLP semantics:

- We extend the program state with two new arguments: an *interleavings table* and a *clock*. The clock is a natural number. Its value in the initial state is 0, and, it is incremented at each new macro-step.
- The initial state starts with an empty interleavings table and a value 0 for the clock.
- The second clause of rule mstep/3 is extended as follows:

```
mstep(S,Status,S4) :-
    add_entry_it(S,Status,S2),
    select_task(S2,Task),
    update_locks(S2,Task,Status,S3),
    run_task(Task,S3,S4).
```

  Predicate add_entry_it/3 adds a new entry $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$ to the interleavings table of state S for the macro-step that has just finished, resulting in state S2. The o, tk, pp and n are obtained from S (resp. from the active location, active task, initial program point of the active task and states's clock). The ρ is the Status parameter which gets instantiated in the calls to mstep/3 from rules return/2 and get/6. In S2 the clock is also incremented for the next macro-step.

Example 5
The interleavings table below is computed for the derivation in Section 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.

5.2 Early Deadlock Detection

Deadlocks can be easily detected in our CLP framework just by adding the following check to function *selectTask*: "if no task can be selected and there is at least a

S_1	$t_{0,main,2} \mapsto \langle 0, return\rangle$	$\emptyset$
S_2	$t_{0,simulate,6} \mapsto \langle 1, return\rangle$	$\emptyset$
S_3	$t_{db,reg,22} \mapsto \langle 2, 25{:}f_4.\mathsf{get}\rangle$	$fut(f_4, w, ping, 41, \bot)$
S_4	$t_{w,work,36} \mapsto \langle 3, 38{:}f_5.\mathsf{get}\rangle$	$fut(f_5, db, getData, 29, \bot)$

location with a non-empty task buffer then there is a deadlock". However, deadlocks can be detected earlier. To this aim, 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's 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 the above check. The early detection of deadlocks therefore allows reducing state exploration.

We first introduce the auxiliary notion of *blocking 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 get 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 Section 5.1. t_{resume} will also be written as $next(t_{stop})$. When the task stops at t_{stop} due to a get 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 more than one get, the execution may choose another actor several times). For this reason, we define the set of successor macro-steps of the same task from a macro-step: $suc(t_{o,tk,pp_0}, IT) = \{t_{o,tk,pp_i} : t_{o,tk,pp_i} \in IT, t_{o,tk,pp_i} \geq t_{o,tk,pp_0}\}$.

Definition 2 (*Blocking Interval*)
Let $S = s(\_, Futs, IT, \_)$ be a state, $I = (t_{stop}, t_{async}, t_{resume})$ is a *blocking interval* of S, written as $I \in S$, iff:

1. $\exists\ t_{stop} = t_{o,tk_0,pp_0} \in IT,\ \ status(t_{stop}) = pp_1{:}x.\mathsf{get}$,
2. $t_{resume} \equiv t_{o,tk_0,pp_1},\ \ fut(x, o_x, tk_x, pp(M)) \in Futs$,
3. $t_{async} \equiv t_{o_x,tk_x,pp(M)}, \nexists\ t \in suc(t_{async}, IT)$ with $status(t) =$**return**.

In condition 3, we can see that if the task starting at t_{async} has finished, then it is not a blocking interval. This is known by checking that this task has not reached return, i.e., $\nexists\ t \in suc(t_{async}, IT)$ such that $status(t) =$**return**. In condition 1, we see that in ρ_{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 get, 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 6
Let us consider again the derivation in Section 3. We have the following blocking interval $(t_{db,reg,22}, t_{w,ping,41}, t_{db,reg,25}) \in S_3$ with $S_3 \equiv (S_3, IT_3)$, since $t_{db,reg,22} \in IT_3$, $status(t_{db,reg,22}, IT_3) = [25{:}f.\mathsf{get}]$, and $fut(f, w, ping, 41, \perp) \in S_3$. This blocking interval captures the fact that the task at $t_{db,reg,22}$ is blocked waiting for task $ping$ to terminate. Similarly, we have the following interval in S_4: $(t_{w,work,36}, t_{db,getD,29}, t_{w,work,38})$.

The following notion of *deadlock chain* relies on the blocking intervals of Definition 2 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 3 (*Deadlock Chain*)
Let $S = s(\_, \_, IT, \_)$ be a state. A chain of time identifiers $t_0, ..., t_n$ is a *deadlock chain* in S, written as $dc(t_0, ..., t_n)$ iff $\forall t_i \in \{t_0, ..., t_{n-1}\}$ s.t. $(t_i, t'_{i+1}, next(t_i)) \in S$ and one of the following conditions holds:

1. $t_{i+1} \in suc(t'_{i+1}, IT)$, or
2. $loc(t'_{i+1}) = loc(t_{i+1})$ *and* $(t_{i+1}, \_, next(t_{i+1}))$ 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 blocking interval contains the initial time t'_{i+1} in which the task will be selected. However, we look for any blocking 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. 6, this is because such task may have started its execution and then suspended due to a subsequent get 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 for 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 the circularity of the chain, since we require that $t_{n+1} \equiv t_0$.

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

This early deadlock detection is integrated into our CLP semantics just by adding the following clause as the new first clause of rule mstep/3

```
    mstep(S,_,S) :- deadlock(S), !.
```

where the deadlock/1 predicate implements Definition 1. We prove that our definition of deadlock is equivalent to the standard definition of deadlock of Flores-Montoya et al. (2013) (proof can be found at Appendix A).

Example 7
Following Example 6, S_4 is a deadlock state since there exists a *deadlock chain* $dc(t_{db,reg,22}, t_{w,work,36})$. For the first element $t_{db,reg,22}$, condition 2 holds since we have that $(t_{db,reg,22}, t_{w,ping,41}, t_{db,reg,25}) \in S_4$, and $(t_{w,work,36}, t_{db,getD,29}, t_{w,work,38})$ is blocking. Condition 2 holds analogously for $t_{w,work,36}$.

5.3 Guiding Testing using Static Deadlock Analysis

This section proposes a deadlock detection methodology that combines static analysis and systematic testing as follows. First, a state-of-the-art deadlock analysis is run, in particular that of (Flores-Montoya et al. 2013), 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 systematically test the program using a novel technique to guide the exploration towards paths that might lead to deadlock cycles. The goals of this process are: (1) finding concrete deadlock traces associated to the feasible cycles, and, (2) discarding unfeasible 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. As our experiments show in Section 6, our technique allows reducing significantly the search space compared to the full systematic exploration.

5.3.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of (Flores-Montoya et al. 2013) returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points, $tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions. Each arrow $e \xrightarrow{p:tk} e'$ should be interpreted like "location or task e is waiting for the termination of location or task e' due to the synchronization instruction at program point p of task tk". 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. Points-to analysis is used as the basis to infer such abstractions. The analysis is object-sensitive, i.e., it distinguishes the actions performed by the different location abstractions. Both the analysis and the semantics can be made *object-sensitive* by keeping the k ancestor abstract locations (where k is a parameter of the analysis and any $k \geq 0$ can be used). For the sake of simplicity of the presentation, we assume $k = 0$ in the formalization (our implementation uses $k = 1$) (e.g., an abstract task is of the form $o_{pp}.m$ where o_{pp} is the abstract location that executes it).

Example 8
In our working example there are two abstract locations, o_7, correspoding to location database created at line 7 and o_9, corresponding to the n locations worker, created inside the loop at line 9; and four abstract tasks, *register*, *getD*, *work* and *ping*. The following cycle is inferred by the deadlock analysis: $o_7 \xrightarrow{25:register} ping \xrightarrow{41:ping} o_9 \xrightarrow{38:work} getD \xrightarrow{29:getD} o_7$. The first arrow captures that the location created at Line 7 is blocked waiting for the termination of task ping because of the synchronization at L25 of task register. 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., ping) and a location (e.g., o_9) 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 observed, it is complex to figure out from them why these dependencies arise, and more importantly the interleavings scheduled to lead to this situation.

5.3.2 Guiding Testing towards Deadlock Cycles

Given an abstract deadlock cycle, we now present a novel technique to guide the systematic 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 used to allow representing incomplete information.

Definition 4 (*Deadlock-cycle constraints*)
Given a state $S = s(\_, Futs, IT, \_)$, a deadlock-cycle constraint takes one of the following two forms:

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

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

Definition 5 (*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 L_i, Tk_i, ϕ is defined as $\phi(e_i \xrightarrow{p_i:tk_i} e_j \xrightarrow{p_j:tk_j} \dots, L_i, Tk_i) =$

$$\begin{cases} \{\exists t_{L_i,Tk_i,\_} \mapsto \langle \_, p_i{:}F_i.\mathsf{get}\rangle, \exists fut(F_i,L_j,Tk_j,p_j,\bot)\} \cup \phi(e_j \xrightarrow{p_j:tk_j} \dots, L_j, Tk_j) & \text{if } e_j = tk_j \\ \phi(e_j \xrightarrow{p_j:tk_j} \dots, L_i, Tk_j) & \text{if } e_j = \ell \end{cases}$$

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 of function ϕ. (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 9
The following deadlock-cycle constraints are computed for the cycle in Example 8:

$$\{\exists t_{L_1,Tk_1,\_} \mapsto \langle \_, 25{:}F_1.\mathsf{get}\rangle,\ \exists\, \mathit{fut}(F_1, L_2, Tk_2, 41, \bot),\ \exists t_{L_2,Tk_2,\_}{\mapsto}\langle \_, 38{:}F_2.\mathsf{get}\rangle,$$
$$\exists \mathit{fut}(F_2, L_3, Tk_3, 29, \bot)\}$$

They are shown in the order in which they are computed by ϕ. The first two constraints require IT to contain a concrete time in which *some* database gets blocked while registering at line 25 for a *certain* worker to receive the result of executing task ping at Line 41. The worker has not got it because of the value $\bot$ in the future. Furthermore, the last two constraints require a concrete time in which *this* worker waits at Line 38 to get the data stored by *some* database at Line 29 and the data is never returned. Note that, in order to preserve completeness, we are not binding the first and the second databases. If the example is generalized with several databases, there could be a deadlock in which a database waits for a worker which waits for another database and worker, so that the last one waits to get the data stored by the first database. This deadlock would not be found if the two databases 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 $\mathsf{check}_{\mathfrak{C}}$ checks the satisfiability of the constraints at a given state.

Definition 6
Given a set of deadlock-cycle constraints $\mathfrak{C}$, and a state $S = s(Locs, Futs, IT, \_)$, check holds, written $\mathsf{check}_{\mathfrak{C}}(S)$, if:

$$\forall t_{L_i,Tk_i,PP} \mapsto \langle N, \mathsf{sync}(p_i, F_i)\rangle \in \mathfrak{C}, \mathit{fut}(F_i, L_j, Tk_j, p_j, \bot) \in \mathfrak{C}$$

one of the following conditions holds:

1. $\mathsf{reachable}(t_{L_i,Tk_i,p_i}, Locs)$

2. $\exists t_{o_i,tk_i,pp} \mapsto \langle n, \mathsf{sync}(p_i, f_i)\rangle \in IT \wedge \mathit{fut}(f_i, o_j, tk_j, p_j, \bot) \in \mathit{Futs}$

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. 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 fut constraint which is violated, i.e., there is an associated future variable in the state where the associated task has finished (the return value is not equal to $\bot$. 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 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 cycle, hence the derivation can be stopped.

Definition 7 (*Deadlock-cycle guided-testing (DCGT)*)
Consider an abstract deadlock cycle c, and an initial state S_0. Let $\mathfrak{C} = \phi(c, L_{init}, Tk_{init})$ where L_{init}, Tk_{init} are fresh variables. We define DCGT, written $exec_c(S_0)$, as the set $\{d : d \in exec(S_0), \mathsf{deadlock}(S_n)\}$, where S_n is the last state in d.

Example 10
Let us consider the DCGT of our working example with the deadlock-cycle of Example 8, and hence with the constraints $\mathfrak{C}$ of Example 9. The interleavings table in the fifth state of the first derivation contains the entries $t_{0,main,2}{\mapsto}\langle 0, return\rangle$, $t_{0,simulate,6}{\mapsto}\langle 1, return\rangle$, $t_{db,register,22}{\mapsto}\langle 2, 25{:}f_0.block\rangle$ and $t_{w,ping,41}{\mapsto}\langle 3, return\rangle\}$.

Constraints $\{\exists t_{L_1,Tk_1,\_}{\mapsto}\langle \_, 25{:}F_1.block\rangle, \exists \mathit{fut}(F_1, L_2, Tk_2, 41), \mathsf{pending}(Tk_2)\}$ are not satisfiable, (task *ping* has already finished at this point, as we can see in the interleavings table). $\mathsf{check}_{\mathfrak{C}}$ does not hold since $t_{L_1,Tk_1,25}$ is not reachable anymore and thus, the derivation is pruned because there is no deadlock state reachable from this state. Similarly, the rightmost derivation is stopped at its fifth state. Also, the 3rd and 4th derivations are stopped by function deadlock of Theorem 1 (*early detection*). Since there are no more deadlock cycles, the search for deadlock detection finishes applying DCGT. Our methodology therefore explores 9 states instead of the 25 explored by the full systematic execution.

Theorem 2 (*Soundness*)
Given a program P, a set of abstract cycles C in P and an initial state S_0, $\forall d \in exec(S_0)$ if d is a derivation whose last state is deadlock, then $\exists c \in C$ s.t $d \in exec_c(S_0)$.

The proof can be found in Appendix A.

5.3.3 Deadlock-based Testing Criteria

In the application of testing for deadlock detection, and in a general setting where there could occur different potential deadlock cycles, the following practical questions arise: are we 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 testing 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 implemented concrete testing schemes for each of the above criteria by using 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 prototype tool SYCO/aPET (Albert et al. 2013; Albert et al. 2016b), a dynamic/static testing tool for the ABS *concurrent objects* language (Johnsen et al. 2012) which includes the POR techniques to detect and avoid redundant executions described in (Albert et al. 2014) and (Albert et al. 2015). The tool is available for online use through a user-friendly web interface at `http://costa.ls.fi.upm.es/syco`, where most of the benchmarks below can also be found. ABS concurrent objects communicate via *asynchronous* method calls and use await and get, resp., as instructions for non-blocking and blocking synchronization. Handling non-blocking synchronization in our framework do not pose any technical complication and has not been included in our formalization for the sake of simplicity. All the components of our implementation however include support for it.

This section summarizes our experimental results which aim at demonstrating the applicability, effectiveness and impact of the proposed techniques. The benchmarks we have used include: (i) classical concurrency patterns containing deadlocks, namely, *DBW* implements a communication protocol between a database and several workers, *F* is a distributed factorial, *PP* is the pairing problem, *UF* is a loop that creates asynchronous tasks and locations, *HB* the hungry birds problem, *SB* is an extension of the sleeping barber; and, (ii) deadlock free versions of some of the above, named *fP* for the *P* program, for which deadlock analyzers give false positives. All of these programs contain a class *Main* which implements a method main with several integer parameters.

Table 1 compares the results of our deadlock guided testing (DGT) methodology both for the deadlock-per-cycle and all-deadlocks criteria, against those obtained using the standard symbolic execution. In all cases, the default POR techniques included in our testing framework are used. Each benchmark is executed with 2 different limits on loop iterations (column k).The selected limits for each benchmark are different and have been chosen according to the complexity of the benchmark. For the symbolic execution and the DGT with the all-deadlock criterion settings we measure: the number of solutions or complete derivations (columns Ans) and the total time taken (columns T). For the DGT with the deadlock-per-cycle criterion, besides the time (column T), we measure the "number of deadlock executions"/"number of unfeasible cycles"/"number of abstract cycles inferred by the deadlock analysis" (column $D/U/C$), and, since the DCGTs for each cycle are independent and can be performed in parallel, we also show the maximum time measured among the different DCGTs (column T_{max}). For instance, for *HB* with $k = 4$, whereas the standard symbolic execution blows-up, our DGT has been able to find all its 1145 deadlock executions in 2847 ms. Also, our DGT for the deadlock-per-cycle criterion is telling us that the program has five different abstract deadlock cycles (found by the deadlock analysis), but it only found a feasible deadlock execution for two of them (therefore 3 of them were spurious), being $6912ms$ the total time of the process, and $3237ms$ the time of the longest DCGT (including the time of the deadlock analysis), and, hence the total time assuming an ideal parallel setting with 5 processors.

Columns in the group **Speedup** show the gains in time of DGT both for deadlock-per-cycle (columns T_{gain} and T^{max}_{gain}) and all-deadlocks (column T^{all}_{gain}) over the standard symbolic execution. In the case of the deadlock-per-cycle criterion we provide the gains both assuming a sequential setting, hence considering value T of DGT (column T_{gain}), and an ideal parallel setting, therefore considering T_{max} (column T^{max}_{gain}). The gains are computed as X/Y, X being the measure of standard symbolic execution and Y that of the corresponding DGT. 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 180s is used. When the timeout is reached, we write $>X$ to indicate that for the corresponding measure we have got X units in the timeout. In the case of the speedups, $>X$ indicates that the speedup would be X if the process finishes right in the timeout, and hence it is guaranteed to be greater than X.

Our experiments confirm our claim that systematic testing complements deadlock analysis. In the case of programs with deadlock, we have been able to provide concrete test cases (including full traces and scheduling decisions) for feasible deadlock cycles and to discard unfeasible cycles. For deadlock-free programs, we have been able to discard all potential cycles and therefore prove deadlock freedom (modulo the termination limit used). More importantly, the experiments demonstrate that

		Symb. Exec.		**DGT (deadlock-per-cycle)**			**DGT (all)**		**Speedup**		
Bmks.	**k**	**Ans**	**T**	**D/U/C**	**T**	T_{max}	**Ans**	**T**	T_{gain}	T^{max}_{gain}	T^{all}_{gain}
DBW	2	1k	1k	1/0/1	22	3	50	99	50.8	374.9	10.8
	3	196k	∞	1/0/1	21	3	3k	6k	>9k	>60k	>32.3
F	3	11k	7k	3/0/3	17	7	1	30	426.1	1k	241.5
	4	269k	∞	3/0/3	33	5	1	72	>5k	>36k	>3k
PP	2	16	15	2/0/2	12	3	2	9	1.3	5.0	1.7
	3	310	224	2/0/2	10	3	8	16	22.5	74.7	14.0
HB	3	16k	10k	2/3/5	325	325	1k	120	31.4	31.4	80.8
	4	206k	∞	2/3/5	7k	7k	2k	3k	>27	>27	>64
UF	2	1	∞	1/0/1	31	4	36	100	>58k	>450k	>18k
	3	167k	∞	1/0/1	66	4	72	261	>3k	>45k	>690
SB	1	29	31	1/0/1	20	4	3	20	1.6	7.8	1.66
	2	217k	∞	1/0/1	19	4	70	177	>94k	>450k	>10k
fUF	2	201k	∞	0/1/1	430	430	0	427	>4k	>4k	>4k
	3	147k	∞	0/1/1	17k	17k	0	18k	>11	>11	>10
fF	3	5k	4k	0/1/1	34	34	0	34	131.0	131.0	131.0
	4	207k	∞	0/1/1	90	90	0	111	>2k	>2k	>1k
fPP	3	7k	4373	0/2/2	29	29	0	30	150.8	150.8	145.8
	6	341k	∞	0/2/2	3k	3k	0	3k	>535	>534.3	>535.1

Table 1. Experimental evaluation

our DGT methodology is effective and that it can achieve a notable reduction of the search space over standard (symbolic) systematic testing. The gains of DGT both in time and in number of explored states are enormous (more than three orders of magnitude in many cases). It can be observed that the gains are much larger in the examples in which the deadlock analysis does not give false positives (namely, in DBW(3), F(4), F(10), PP(9), UF(3)). An explanation for this is that, in general, the generated constraints for unfeasible cycles are often not able to guide the exploration effectively (e.g. in HB(4)). Indeed, if we consider HB(5) we cannot find a representative of one of the abstract cycles but neither we are able to prove it is a false positive. Even in these cases, DGT outperforms symbolic execution in terms of scalability and flexibility. Let us also observe that the gains are less notable in deadlock-free examples. That is because, on one hand, all cycles are unfeasible in this case, and, on the other, each DCGT cannot stop until all potential deadlock paths have been considered. As expected, when we consider a parallel setting, the gains are much larger.

All in all, we argue that our experiments show that our methodology complements deadlock analysis, finding deadlock traces for the potential deadlock cycles and discarding unfeasible ones, with a significant reduction. It is very effective for programs that contain deadlock, and it is also able to prove deadlock freedom for some cases in which a static analysis reports false positives.

7 Related Work

Since our method uses in conjunction static and dynamic analyses, and the individual methods can be used for multiple purposes, we need to relate it to a wide spectrum of existing techniques that we classify as follows.

7.1 Deadlock Analysis

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 (Flores-Montoya et al. 2013; Giachino et al. 2013) and thread-based programs (Masticola and Ryder 1991; Savage et al. 1997), is based on static analysis techniques. Static analysis can ensure the absence of errors, however it works on approximations (especially for 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 deadlock freedom. Using our method, we try to find a deadlock by exploring the paths (possibly infinite) given by a deadlock detection algorithm that relies on the static information. Although we have used the output given by the deadlock analyzer of (Flores-Montoya et al. 2013), our combined approach could use the output of other static analyzers (e.g., (Giachino et al. 2013)) without requiring any conceptual change to the combined framework.

7.2 Symbolic Execution, Verification, Model Checking, Testing

The core of our CLP-based framework is the symbolic execution engine presented in Section 4. By relying only on this component, one can clearly do (non-guided) deadlock detection already, and besides other types of errors can also be captured (e.g., find critical states that can cause the system to crash). This is the approach taken in model checking and other verification techniques which are based on symbolic execution to automatically verify correctness properties. Indeed, deadlock detection has been intensively studied in the context of model checking (see, e.g., (Rabinovitz and Grumberg 2005)).

Both static and dynamic testing aim at finding bugs, among them deadlocks (see, e.g., (Christakis et al. 2013; Joshi et al. 2009; Kheradmand et al. 2013; Havelund 2000)). Indeed, symbolic execution is at the core of static testing systems and our symbolic execution engine is the basis for the aPET testing system (Albert et al. 2012).

Current research on testing for concurrent systems has focused on avoiding the generation of redundant executions which result from interleaving independent processes (e.g., processes that operate on disjoint areas of the memory). Dynamic Partial Order Reduction (DPOR) (Flanagan and Godefroid 2005; Tasharofi et al. 2012; Albert et al. 2014) is a successful technique to avoid such redundancies. Our work is orthogonal to such line of research, in the sense that we can use DPOR techniques within our framework and the combined approach is still valid. Indeed, our implementation uses the techniques described in (Albert et al. 2014) to eliminate redundancies, as we have mentioned in the experiments section.

7.3 Hybrid Approaches

We now relate our work to hybrid approaches that use static information during testing for deadlock detection, namely (Joshi et al. 2010) and (Agarwal et al. 2006).

As regards (Joshi et al. 2010), it 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), the 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. Besides, our work is based on static testing that generalizes dynamic testing to allow any input data.

As regards (Agarwal et al. 2006), 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 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.

8 Conclusions and Future Work

It is known that testing of concurrent systems suffers from the state explotion problem that results from considering all interleavings of processes. We have proposed a hybrid approach that uses the information yield by a static deadlock analyzer in order to guide the execution of a testing tool towards potential deadlock paths and discard paths that are guaranteed to be deadlock free. As our experiments show, our hybrid approach is more scalable than not-guided testing for deadlock detection. Besides, we can combine it with state-of-the-art DPOR techniques that eliminate redundancies in order to be even more effective.

Indeed, we are currently working on improving existing techniques that detect redundancies during systematic testing in order to prune the search tree even further. Our idea is to work on a more refined notion of independence that will allow us to avoid certain interleavings between independent processes which are not captured by existing techniques. The important point to note is that the achieved improvements will be directly applicable to our hybrid framework for deadlock detection. We are also studying the possibility of guiding the search towards other properties of interest for the actors concurrency model. These are lines for future research.

On the other hand, although in static testing, the system under test is generally analyzed without any knowledge on its inputs, in the case of concurrent and distributed programs, an initial distributed configuration (set of locations and their connections) is usually provided. E.g., this is the case of the simulate method on our motivating example. Our symbolic execution framework is however able to symbolically execute a system from a completely unknown distributed context. This would allow automatically and systematically generating distributed contexts in order to detect those in which a deadlock can occur. In the case of our motivating example, this would mean generating for instance a context with a database location and some worker location with a scheduled work task and a register task on the database for it. However, this would cause a further combinatorial explosion on the different possible distributed contexts that can be generated, most of which will not be relevant for deadlock detection. Interestingly, the deadlock analysis provides the possibly conflicting task interactions that can lead to deadlock. This information could be used to help our framework discard initial distributed configurations that cannot lead to deadlock. This is also a line for future research.

References

Abdulla, P., Aronis, S., Jonsson, B., and Sagonas, K. F. 2014. Optimal dynamic partial order reduction. In *Proc. of POPL'14*. ACM, 373–384.

Agarwal, R., Wang, L., and Stoller, S. D. 2006. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Conf. on Hardware and Software Verification and Testing*. Lecture Notes in Computer Science, vol. 3875. Springer, 191–207.

Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., and an d Guillermo Román-Díez, G. P. 2015. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability 25,* 3, 218–271.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2007. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*. LNCS, vol. 4421. Springer, 157–172.

Albert, E., Arenas, P., and Gómez-Zamalloa, M. 2012. Symbolic Execution of Concurrent Objects in CLP. In *Proc. of PADL'12*. LNCS, vol. 7149. Springer, 123–137.

Albert, E., Arenas, P., and Gómez-Zamalloa, M. 2014. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *Proc. of FORTE'14*. LNCS, vol. 8461. Springer, 49–65.

Albert, E., Arenas, P., and Gómez-Zamalloa, M. 2015. Test Case Generation of Actor Systems. In *Proc. of ATVA'15*. LNCS, vol. 9364. Springer, 259–275.

Albert, E., Arenas, P., Gómez-Zamalloa, M., and Wong, P. Y. 2013. aPET: A Test Case Generation Tool for Concurrent Objects. In *Proc. ESEC/FSE'13, pp. 595–598.* ACM. Available at http://costa.ls.fi.upm.es/apet.

Albert, E., Gómez-Zamalloa, M., and Isabel, M. 2016a. Combining Static Analysis and Testing for Deadlock Detection. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*. Lecture Notes in Computer Science, vol. 9681. Springer, 409–424.

Albert, E., Gómez-Zamalloa, M., and Isabel, M. 2016b. SYCO: A Systematic Testing Tool for Concurrent Objects. In *Proc. of CC'16*. ACM, 269–270.

Christakis, M., Gotovos, A., and Sagonas, K. F. 2013. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 154–163.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst. 13,* 4, 451–490.

de Boer, F. S., Clarke, D., and Johnsen, E. B. 2007. A Complete Guide to the Future. In *Programming Languages and Systems, 16th European Symposium on Programming,* *ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, R. de Nicola, Ed. Lecture Notes in Computer Science, vol. 4421. Springer, 316–330.

Flanagan, C. and Godefroid, P. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds. ACM, 110–121.

Flores-Montoya, A., Albert, E., and Genaim, S. 2013. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*. LNCS 7892. Springer, 273–288.

Giachino, E., Grazia, C., Laneve, C., Lienhardt, M., and Wong, P. 2013. Deadlock Analysis of Concurrent Objects – Theory and Practice.

Giachino E., K. N. and C., L. 2014. Deadlock analysis of unbounded process networks. In *CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*. 63–77.

Giachino E., L. C. and M., L. 2016. A framework for deadlock detection in core ABS. *Software and System Modeling 15,* 4, 1013–1048.

Gómez-Zamalloa, M., Albert, E., and Puebla, G. 2009. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology 51,* 10 (October), 1409–1427.

Gómez-Zamalloa, M., Albert, E., and Puebla, G. 2010. 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.

Havelund, K. 2000. Using runtime analysis to guide model checking of java programs. In *SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*. 245–264.

Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. 2012. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds. Lecture Notes in Computer Science, vol. 6957. Springer, 142–164.

Joshi, P., Naik, M., Sen, K., and D, G. 2010. An effective dynamic analysis for detecting generalized deadlocks. In *Proc. of FSE'10*. ACM, 327–336.

Joshi, P., Park, C., Sen, K., and Naik, M. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proc. of PLDI'09*. ACM, 110–120.

Kheradmand, A., Kasikci, B., and Candea, G. 2013. Lockout: Efficient Testing for Deadlock Bugs. Tech. rep. Available at `http://dslab.epfl.ch/pubs/lockout.pdf`.

Masticola, S. P. and Ryder, B. G. 1991. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *Parallel and Distributed Debugging*. ACM, 97–107.

Rabinovitz, I. and Grumberg, O. 2005. Bounded model checking of concurrent programs. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer, 82–97.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15,* 4, 391–411.

Sen, K. and Agha, G. 2006. Automated Systematic Testing of Open Distributed Programs. In *Proc. of FASE'06*. LNCS, vol. 3922. Springer, 339–356.

Tasharofi, S., Karmani, R. K., Lauterburg, S., Legay, A., Marinov, D., and Agha, G. 2012. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, H. Giese and G. Rosu, Eds. Lecture Notes in Computer Science, vol. 7273. Springer, 219–234.

9 Appendix

Proof of Theorem 1
Given a program state $\mathcal{S} = (S, IT)$, its *dependency graph* G_S and its *abstract dependency graph* $\mathcal{G}$ are formalized in (Flores-Montoya et al. 2013). Let us define function γ which transforms a *sequence of times* each of them fulfilling (1) or (2) in Definition 3 into a path in G_S.

Definition 8 (γ)
Given a state $\mathcal{S}{=}(S, IT)$ and a sequence of times $\{t_0, ..., t_n\}$ *in* S, satisfying (1) or (2) in Definition 3, the *one-to-one* function $\gamma(\{t_0, ...t_n\}){=}e_1{\to}e_2 {\to}\cdots{\to}e_n$ in G_S is defined as follows:

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

where γ_{tk} is the following auxiliary function:

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

We need to distinguish between functions γ and γ_{tk}, as in (Flores-Montoya et al. 2013), 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 γ (γ_{tk}) is: given a *sequence of times* $\{t_0, ..., t_n\} \in S$, we define a path whose edges are obtained as follows: $\forall t_i \in \{t_0, .., t_n\}$ such that $(t_i, t'_{i+1}, next(t_i)) \in S$. If (1) holds, 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) hold, 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 $tsk(t'_{i+1})$).

Theorem 3 (*(Flores-Montoya et al. 2013)*)
Let S be a reachable state and G^{tt}_S the dependency graph taking only task-task dependencies. If future variables cannot be stored in fields, G^{tt}_S is acyclic.

Theorem 4 (*equivalence*)
Let S be a program state,

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

Proof
$\Rightarrow$. Let $dc(\{t_0, ..., t_n\})$ be a deadlock chain, then we could apply function γ, as $\forall t_i \in \{t_0, ..., t_n\}$, t_i satisfies (1) or (2). So, we obtain a path in G_S and using the last condition in Definition 3, both $\gamma(\{t_n\})$ and $\gamma_{tk}(\{t_n\})$ add the edge $tk(t'_0) \to loc(t_0)$ causing the path to become a cycle.
$\Leftarrow$. Given a cycle in G_S, by theorem 3, this one contains at least one object node, which is required by function γ. Now, this case is analogous to the previous one. $\square$

□

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 two disjunct partial maps:

Definition 9 (*l+a*)
Let l and a be two partial maps such that $dom(l) \cap dom(a) = \emptyset$:

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

Definition 10 (*points-to soundness (Albert et al. 2015)*)
Soundness of the points-to analysis amounts to requiring the existence of a partial map α, that maps location and task identifiers to the corresponding abstract ones, such that for any task $tsk(tk, m, o, l, s)$, where o is the object identifier that executes the task tk, and location $loc(o, tk, h, \mathcal{Q})$ in any reachable state S, we have that:

1. $\alpha(tk) = \alpha(o).m$
2. Let x be an location variable $x \in dom(l+h)$, if $\alpha((l+h)(x)) = ob$ then $ob \in \mathcal{A}(\alpha(o), pp(s), x)$.
3. Let x be a future variable, $x{\in}dom(l{+}h)$, $(l{+}h)(x){=}tk_2$ and $tsk(tk_2, m_2, o_2, l_2, \epsilon(v)){\in}\mathtt{T}$ (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 a future variable, $x \in dom(l{+}h)$, $(l{+}h)(x) = tk_2$, $tsk(tk_2, m_2, o_2, l_2, s_2) \in \mathtt{T}$ 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 $\overline{\alpha}$ be the extension of α over the paths in G_s that applies the function α in every node contained by the path.

Definition 11 (*deadlock soundness (Flores-Montoya et al. 2013)*)
Let S be a reachable state. If there is a cycle $\gamma = e_1 \to e_2 \to \cdots \to e_1$ in G_S, then $\overline{\alpha}(\gamma) = \alpha(e_1) \xrightarrow{p_1:tk_1} \alpha(e_2) \xrightarrow{p_2:tk_2} \cdots \xrightarrow{p_n:tk_n} \alpha(e_1)$ is an abstract cycle of $\mathcal{G}$.

Theorem 5
Given an initial state S_0 and an abstract cycle c, $\forall d \in exec(S_0)$, $d \equiv S_0 \longrightarrow^* S_n$, if $\exists\ dc(\{t_0, ..., t_n\}) \in S_n$ such that $\overline{\alpha} \circ \gamma(\{t_0, ..., t_n\}) \in c$, then $d \in exec_c(S_0)$.

Proof
By contradiction, let us suppose that $\exists d \in exec(S_0)$ and $d \notin exec_c(S_0)$. Hence, $\exists S_i \in d$ such that $\mathsf{check}_{\mathfrak{C}}(S_i)$ returns false and, consequently, the derivation $S_0 \longrightarrow^* S_i$ stops, where $\mathfrak{C} = \phi(c, L, Tk)$ and L, Tk are fresh variables.
Therefore, at S_i $\exists\{t_{L_i,Tk_i,PP} \mapsto \langle N, \mathsf{sync}(p_i, F_i)\rangle, fut(F_i, L_j, Tk_j, p_j)\} \subset \mathfrak{C}$ does not hold and neither (1) nor (2) in Definition 6. However, this cannot happen, as $\mathfrak{C}$ imposes necessary constraints for the existence of some representative of c and S_n contains a cycle that is a representative of c, then (1) or (2) must be fulfilled in every state of d. As a result, we get a contradiction. □

Proof of Theorem 2

If the last state is deadlock, then $\exists dc(\{t_0, ..., t_n\}) {\in} S_n$, by Th. 1. Using the soundness of deadlock analysis over the cycle $\gamma(\{t_0, ..., t_n\})$, the existence of c is ensured. Now, by theorem 5, we obtain the result. $\square$