Combining Static Analysis and Testing for Deadlock Detection Technical Report (including Proofs)

Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel

Complutense University of Madrid (UCM), Spain

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

1 Introduction

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

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

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

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

2 Asynchronous Programs: Syntax and Semantics

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

$$\text{(MSTEP)}\ \frac{selectLoc(S) = loc(o, \bot, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, selectTask(o) = tsk(tk, m, l, s),\quad S \diamond \boldsymbol{\rho_\emptyset} \overset{o\cdot tk}{\leadsto^*} S' \diamond \boldsymbol{\rho}}{S \xrightarrow{o\cdot tk} S'}$$

$$\frac{\text{(NEWLOC)}\quad tk = tsk(tk, m, l, x = \mathsf{new}\ D; s), \text{fresh}(o'), h' = newheap(D), l' = l[x \to o']}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \boldsymbol{\rho_0} \leadsto loc(o, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l', s)\}) \cdot loc(o', \bot, h', \{\}) \diamond \boldsymbol{\rho_0}}$$

$$\frac{\text{(ASYNC)}\ tk = tsk(tk, m, l, y{=}x!m_1(\bar{z}); s), l(x){=}o_1,\ \text{fresh}(tk_1),\ l_1{=}buildLocals(\bar{z}, m_1, l)}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot loc(o_1, \_, \_, \mathcal{Q}') \diamond \boldsymbol{\rho_0} \leadsto loc(o, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\})\cdot loc(o_1, \_, \_, \mathcal{Q}' \cup \{tsk(tk_1, m_1, l_1, body(m_1))\}) \cdot fut(y, o_1, tk_1, ini(m_1)) \diamond \boldsymbol{\rho_0}}$$

$$\text{(RETURN)}\ \frac{tk = tsk(tk, m, l, \mathbf{return}; s), \boldsymbol{\rho_1 = return}}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \boldsymbol{\rho_0} \leadsto loc(o, \bot, h, \mathcal{Q} \cup \{tsk(tk, m, l, \epsilon)\}) \diamond \boldsymbol{\rho_1}}$$

$$\text{(AWAIT1)}\ \frac{tk = tsk(tk, m, l, y.\mathsf{await}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Ob}, s_1 = \epsilon}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto loc(o, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0}}$$

$$\text{(AWAIT2)}\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{await}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Ob}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp : y.\mathsf{await}}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto loc(o, \bot, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_1}}$$

$$\text{(BLOCK1)}\ \frac{tk = tsk(tk, m, l, y.\mathsf{block}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Ob}, s_1 = \epsilon}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto loc(o, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0}}$$

$$\frac{\text{(BLOCK2)}\quad tk{=}tsk(tk, m, l, pp{:}y.\mathsf{block}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Ob}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp{:}\boldsymbol{y}.\mathsf{block}}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \leadsto loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_1}}$$

Fig. 1. Semantics of Asynchronous Programs

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

Fig. 1 presents the semantics of the language. The information about $\boldsymbol{\rho}$ in bold font is part of the extensions for testing in Sec. 4 and should be ignored by now. A *state* or *configuration* is a set of locations and future variables $o_0 \cdots o_n \cdot fut_0 \cdots fut_m$. A *location* is a term $loc(o, tk, h, \mathcal{Q})$ where o is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\bot$ 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 $fut(id, o, tk, m)$ where id is a unique future variable identifier, o is the location identifier that executes the task tk awaiting for the future, and m is the initial program point of tk. A *task* is a term $tsk(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed or ϵ if the task has terminated. We assume that the execution starts from a main method without parameters. The initial state is $St=\{loc(0, 0, \perp, \{tsk(0, main, l, body(main))\}$ with an initial location with identifier 0 executing task 0. Here, l maps local variables to their initial values (**null** in case of reference variables) and $\perp$ is the empty heap. $body(m)$ is the sequence of instructions in method m, and we can know the program point pp where an instruction s is in the program as follows $pp{:}s$.

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

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

3 Motivating Example

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

```
1  main() {
2    Ba barber = new Ba();
3    Cl client = new Cl();
4    Ch chair = new Ch();
5    client!wakeup(barber,chair);
6    barber!sleeps(client,chair);
7  }
8  class Ba{
9    Unit sleeps(Cl cl, Ch ch){
10      Fut f=ch!taken(cl);
11      f.block;}
12   Unit cuts(){}
13 }
```

```
14 class Ch{
15    Unit taken(Cl cl){
16      Fut f=cl!sits();
17      f.await;}
18   Unit isClean(){}
19 }
20 class Cl{
21   Unit wakeup(Ba b, Ch ch){
22      Fut f=b!cuts();
23      ch!isClean();
24      f.block;}
25   Unit sits(){}
26 }
```

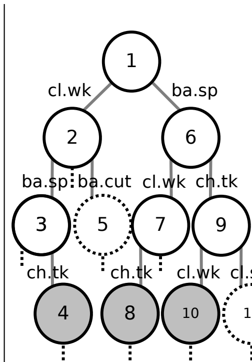

Fig. 2. Classical Sleeping Barber Problem (left) and Execution Tree (right)

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

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

$St_0 \equiv loc(ini,..)\cdot loc(cl,..,\{tsk(1,wk,..)\})\cdot loc(ba,..,\{tsk(2,sp,..)\})\cdot loc(ch,..) \xrightarrow{cl,1}$

$St_1 \equiv loc(cl,..,\{tsk(1,wk,f_0.block)\})\cdot loc(ba,..,\{tsk(3,cut,..),..\})\cdot fut(f_0,ba,3,12)\cdot .. \xrightarrow{ba,2}$

$St_2 \equiv loc(ba,..,\{tsk(2,sp,f_1.block)\})\cdot loc(ch,..,\{tsk(5,tk,..),..\})\cdot fut(f_1,ch,5,15)\cdot .. \xrightarrow{ch,5}$

$St_3 \equiv loc(ch,..,\{tsk(5,tk,f_2.await),..\})\cdot loc(cl,..,\{tsk(6,st,..),..\})\cdot fut(f_2,cl,6,25)\cdot ..$

$\xrightarrow{ch,4} St_4 \equiv loc(ch,..\{tsk(4,isClean,return),..\})\cdot ..$

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

Fig. 3. MSTEP2 rule for combined testing and analysis

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

4 Testing for Deadlock Detection

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

4.1 An Enhanced Semantics for Deadlock Detection

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

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

We use a function $\boldsymbol{clock(n)}$ to represent a clock that starts at 0, is increased by one in every execution of ***clock***, and returns the current value **n**. The initial entry is $t_{0,0,1} \mapsto \langle 0, \rho_0 \rangle$, being 0 the identifier for the initial location and task, and 1 the first program point of *main*. The clock also assigns the value 0 as the first element in the tuple and a fresh variable in the the second element ρ_0. The next macro-step will be assigned clock value 1, next 2, and so on. As notation, we define the relation $t \in table$ if there exists an entry $t \mapsto \langle n, \rho \rangle \in table$, and the function $status(t, table)$ which returns the status ρ_t such that $t \mapsto \langle n, \rho_t \rangle \in table$. The semantics is extended by changing rule MSTEP as in Fig. 3. The function deadlock will be defined in Thm. 1 to stop derivations as soon as deadlock is detected. Function $\mathsf{check}_{\mathfrak{C}}$ should be ignored by now, it will be defined in Sec. 5.2. Essentially, there are two new aspects: (1) The state is extended with the status ρ, namely all rules include a status ρ attached to the state using the symbol $\diamond$. The status is showed in bold font in Fig. 1 and can get a value in rules block2, await2 and return. The initial value ρ_0 is a fresh variable. (2) The state for the macrostep is extended with the interleavings table *table*, and a new entry $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$ is added to *table* in every macrostep if there has been progress in the execution, i.e., $S' \neq S$, being n the current clock time.

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

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

4.2 Formal Characterization of Deadlock State

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

We first introduce the auxiliary notion of *waiting interval* which captures the period in which a task is waiting for another one to terminate. In particular, it is defined as a tuple $(t_{stop}, t_{async}, t_{resume})$ where t_{stop} is the macro-step at which the location stops executing a task due to some block/await instruction, t_{async} is the macro-step at which the task that is being awaited is selected for execution, and, t_{resume} is the macro-step at which the task will resume its execution. t_{stop}, t_{async} and t_{resume} are time identifiers as defined in Sec. 4.1. t_{resume} will also be written as $next(t_{stop})$. When the task stops at t_{stop} due to a block instruction, we call it *blocking interval*, as the location remains blocked between t_{stop} and $next(t_{stop})$ until the awaited task, selected in t_{async}, has already finished. The execution of a task can have several points at which macro-steps are performed (e.g., if it contains several await or block the processor may be lost several times). For this reason, we define the set of successor macro-steps of the same task from a macro-step: $suc(t_{o,tk,pp_0}, table) = \{t_{o,tk,pp_i} : t_{o,tk,pp_i} \in table, t_{o,tk,pp_i} \geq t_{o,tk,pp_0}\}$.

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

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

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

In condition 3, we can see that if the task starting at t_{async} has finished, then it is not a waiting interval. This is known by checking that this task has not reached return, i.e., $\nexists\, t \in suc(t_{async}, table)$ such that $status(t) = return$. In condition 1, we see that in 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 await/block, since this is the point that the macro-step rule uses to define the macro-step identifier t_{o,tk_0,pp_1} associated to the resumption of the waiting task.

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

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

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

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

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

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

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

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

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

5 Combining Static Deadlock Analysis and Testing

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

5.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [11] returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, 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. Three kinds of arrows can be distinguished, namely, *task-task* (a task is awaiting for the termination of another one), *task-location* (a task is awaiting for a location to be idle) and *location-task* (the location is blocked due the task). *Location-location* arrows cannot happen. The abstractions for tasks and locations can be performed at different levels of accuracy during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location o by the program point at which it is created o_{pp}, and each task by the method name executing. They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Both the analysis and the semantics can be made *object-sensitive* [3] by keeping the k ancestor abstract locations (where k is a parameter of the analysis). For the sake of simplicity of the presentation, we assume $k = 0$ in the formalization (our implementation uses $k = 1$).

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

5.2 Guiding Testing towards Deadlock Cycles

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

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

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

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

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

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

Notation $\mathsf{sync}(p_i, F_i)$ is a shortcut for $p_i{:}F_i.\mathsf{block}$ or $p_i{:}F_i.\mathsf{await}$. Uppercase letters appearing for the first time in the constraints are fresh variables. The first case handles location-task and task-task arrows (since e_j is a task abstraction), whereas the second case handles task-location arrows (e_j is an abstract location). Let us observe the following: (1) The abstract location and task identifiers of the abstract cycle are not used to produce the constraints. This is because constraints refer to concrete identifiers. Even if the cycle contains the same identifier on two different nodes or arrows, the corresponding variables in the constraints cannot be bound (i.e., we cannot use the same variables) since they could refer to different concrete identifiers. (2) The program points of the cycle (p_i and p_j) are used in time and fut constraints. (3) Location and task identifier variables of fut constraints and subsequent time or pending constraints are bound (i.e., the same variables are used). This is done using the 2nd and 3rd parameters of function 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 5. The following deadlock-cycle constraints are computed for the cycle in Ex. 4: $\{\, \exists t_{O_1,Tk_1,\_} \mapsto \langle \_, 11{:}F_1.block\rangle,\ \exists\, fut(F_1, O_2, Tk_2, 15),\ \exists t_{O_2,Tk_2,\_} \mapsto \langle \_,$ $17{:}F_2.await\rangle, \exists\, fut(F_2, O_3, Tk_3, 25),\ \mathsf{pending}(Tk_3),\ \exists t_{O_3,Tk_4,\_} \mapsto \langle \_, 24{:}F_3.block\rangle,$ $\exists fut(F_3, O_4, Tk_5, 12), \mathsf{pending}(Tk_5)\}$. They are shown in the order in which they are computed by ϕ. The first four constraints require *table* to contain a concrete time in which *some* barber sleeps waiting at L11 for a *certain* chair to be taken at L15 and, during another concrete time, this one waits at L17 for a *certain* client to sit at L25. The client is not allowed to sit by the 5th constraint. Furthermore, the last three constraints require a concrete time in which *this* client waits at L24 to get a haircut by *some* barber at L12 and that haircut is never performed. Note that, in order to preserve completeness, we are not binding the first and the second barber. If the example is generalized with several clients and barbers, there could be a deadlock in which a barber waits for a client which waits for another barber and client, so that the last one waits to get a haircut by the first one. This deadlock would not be found if the two barbers are bound in the constraints (i.e., if we use the same variable name). In other words, we have to account for deadlocks which traverse the abstract cycle more than once.

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

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

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

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

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

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

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

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

5.3 Deadlock-based Testing Criteria

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

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

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

6 Experimental Evaluation

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

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

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

Table 1. Experimental evaluation

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

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

7 Conclusions and Related Work

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

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

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

References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal dynamic partial order reduction. In *P*roc. of POPL'14, pages 373–384. ACM, 2014.
2. R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *C*onf. on Hardware and Software Verification and Testing, *L*NCS 3875, pages 191–207. Springer, 2006.
3. E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *S*oftware Testing, Verification and Reliability, 25(3):218–271, 2015.
4. E. Albert, P. Arenas, and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *P*roc. FORTE'14, LNCS 8461, pp. 49-65. Springer, 2014.
5. E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In *F*SE'13, pp. 595–598. ACM, 2013.
6. E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. Technical report, 2015. Available at: http://costa.ls.fi.upm.es/papers/costa/AlbertGI15.pdf.
7. B. D. Bingham, J. D. Bingham, J. Erickson, and M. R. Greenstreet. Distributed Explicit State Model Checking of Deadlock Freedom. In *P*roc. of CAV'13, volume 8044 of *L*ecture Notes in Computer Science, pages 235–241. Springer, 2013.
8. M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2*013 IEEE Sixth International Conf. on Software Testing, Verification and Validation, pages 154–163. IEEE, 2013.
9. F. S. de Boer, M. Bravetti, I. Grabe, M. David Lee, M. Steffen, and G. Zavattaro. A Petri Net based Analysis of Deadlocks for Active Objects and Futures. In *P*roc. of FACS 2012, 2012.
10. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *P*roc. POPL'05, pp. 110-121. ACM, 2005.
11. A. Flores, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. *F*ORTE'13, LNCS, pp 273–288. Springer, 2013.
12. E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.
13. P. Joshi, M. Naik, K. Sen, and Gay D. An effective dynamic analysis for detecting generalized deadlocks. In *P*roc. of FSE'10, pages 327–336. ACM, 2010.
14. P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *P*LDI'09, pages 110–120. ACM, 2009.
15. A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report http://dslab.epfl.ch/pubs/lockout.pdf, 2013.
16. S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *P*DD'91, pages 97–107. ACM, 1991.
17. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *P*roc. of ICSE, pages 386–396. IEEE, 2009.
18. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM* Trans. Comput. Syst., 15(4):391–411, 1997.
19. K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *P*roc. FASE'06, LNCS 3922, pp. 339-356. Springer, 2006.
20. S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. *F*ORTE, LNCS 7273, pages 219–234. Springer, 2012.

8 Appendix

Proof (Proof of Theorem 1).

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

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

$$\gamma(\{t_0,...,t_n\}) = \begin{cases} \{loc(t_0) \to tsk(t_1)\} \cup \gamma_{tk}(\{t_1,...,t_n\}) & \textit{if } t_0 \textit{ holds (1)} \\ \{loc(t_0) \to tsk(t_1') \to loc(t_1')\} \cup \gamma(\{t_1,...,t_n\}) & \textit{if } t_0 \textit{ holds } (2) \wedge \neg(1) \end{cases}$$

where γ_{tk} is the following auxiliar function:

$$\gamma_{tk}(\{t_0,...,t_n\}) = \begin{cases} \{tsk(t_0) \to tsk(t_1)\} \cup \gamma_{tk}(\{t_1,...,t_n\}) & \textit{if } t_0 \textit{ holds (1)} \\ \{tsk(t_0) \to tsk(t_1') \to loc(t_1')\} \cup \gamma(\{t_1,...,t_n\}) & \textit{if } t_0 \textit{ holds } (2) \wedge \neg(1) \end{cases}$$

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

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

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

$$\exists\ dc(\{t_0, ..., t_n\}) \in St \Longleftrightarrow \exists\ cycle\ \gamma(\{t_0, ..., t_n\}) \in G_S$$

Proof.
⇒ . Let $dc(\{t_0, ..., t_n\})$ be a deadlock chain, then we could apply the 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 Def. 2, both $\gamma(\{t_n\})$ and $\gamma_{tk}(\{t_n\})$ add the edge $tk(t'_0) \to loc(t_0)$ causing the path becomes a cycle.
⇐. Given a cycle in G_S, by the lemma 1 , this one contains at least one object node, which is required by the function γ. Now, This case is analogous to the previous one.

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

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

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

Definition 9 (points-to soundness [3]). *Soundness of the points-to analysis amounts to requiring the existence a partial map α, that maps location and task identifiers to 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 $loc(o, tkh, \mathcal{Q})$ in any reachable state S, we have that:*

1. $\alpha(tk) = \alpha(o).m$
2. *Let x be an location variable $x \in dom(l+h)$, if $\alpha((l+h)(x)) = ob$ then $ob \in \mathcal{A}(\alpha(o), pp(s), x)$.*
3. *Let x be future variable, $x \in dom(l+h)$, $(l+h)(x) = tk_2$ 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 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 10 (deadlock soundness [3]). *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 $\mathcal{G}$.*

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

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

Proof (Proof of Theorem 2). If the last state is deadlock, then $\exists dc(\{t_0, ..., t_n\}) \in St_n$, 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 Lemma 2, we obtain the result.