Combining Static Analysis and Testing for Deadlock Detection ⋆

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. 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. 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 prove deadlock freedom.

1 Introduction

In concurrent programs, *deadlocks* are one of the most common programming errors 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 which 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. 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 deadlock is found, state-of-the-art analysis tools

⋆ This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

[6, 7, 12] 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 of executing the application for concrete input values. 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* [4, 14] 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 [1, 3–5], the search space of systematic testing (even without redundancies) can be huge. This is a threat to the application of testing in concurrent programming. (2) There is only guarantee of deadlock freedom for finite-state terminating programs (terminating executions with concrete inputs).

This paper proposes a seamless combination of static analysis and testing for effective deadlock detection as follows: an existing static deadlock analysis [6] 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). In summary, the main contributions of this paper are:

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.
4. We have implemented our methodology in the SYCO system (see Sect. 6) and performed a thorough experimental evaluation on some classical examples.

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* 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

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

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

$$\frac{\textsc{(async)}\ tk = tsk(tk, m, l, pp{:}y{=}x!m_1(\bar{z}); s), l(x){=}\ell_1,\ \text{fresh}(tk_1),\ l_1{=}buildLocals(\bar{z}, m_1, l)}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot loc(\ell_1, _, _, \mathcal{Q}') \diamond \boldsymbol{\rho_0} \leadsto loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\})\cdot \\ loc(\ell_1, _, _, \mathcal{Q}' \cup \{tsk(tk_1, m_1, l_1, body(m_1))\}) \cdot fut(y, o_1, tk_1, ini(m_1)) \diamond \boldsymbol{\rho_0}\end{array}}$$

$$\textsc{(return)}\ \frac{tk = tsk(tk, m, l, pp{:}\mathsf{return}; s), \boldsymbol{\rho_1} = \mathsf{return}}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \boldsymbol{\rho_0} \leadsto loc(\ell, \bot, h, \mathcal{Q} \cup \{tsk(tk, m, l, \epsilon)\}) \diamond \boldsymbol{\rho_1}}$$

$$\textsc{(await1)}\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{await}; s), tsk(tk_1, _, _, s_1) \in \mathsf{Loc}, s_1 = \epsilon}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_0} \leadsto \\ loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_0}\end{array}}$$

$$\textsc{(await2)}\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{await}; s), tsk(tk_1, _, _, s_1) \in \mathsf{Loc}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp{:}y.\mathsf{await}}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_0} \leadsto \\ loc(\ell, \bot, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_1}\end{array}}$$

$$\textsc{(block1)}\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{block}; s), tsk(tk_1, _, _, s_1) \in \mathsf{Loc}, s_1 = \epsilon}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_0} \leadsto \\ loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_0}\end{array}}$$

$$\frac{\textsc{(block2)}\ \ tk{=}tsk(tk, m, l, pp{:}y.\mathsf{block}; s), tsk(tk_1, _, _, s_1) \in \mathsf{Loc}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp{:}y.\mathsf{block}}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_0} \leadsto loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, _, tk_1, _) \diamond \boldsymbol{\rho_1}}$$

Fig. 1. Macro-Step Semantics of Asynchronous Programs

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.await | f.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 for now. A *state* or *configuration* is a set of locations and future variables $loc_0 \cdots loc_n \cdot fut_0 \cdots fut_m$. A *location* is a term $loc(\ell, tk, h, \mathcal{Q})$ where ℓ 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 $fut(id, \ell, tk, m)$ where id is a unique future variable identifier, ℓ 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 [14] (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 $selectTask(\ell)$ to select non-deterministically one task of ℓ's queue. 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 $\ell \neq \ell_1$, but the case $\ell = \ell_1$ is analogous, the new task tk_1 is added to $\mathcal{Q}$ of ℓ. 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 $\ell \cdot tk$, the name of the location ℓ and task tk selected (in rule MSTEP) or evaluated in the step (in the transition $\rightsquigarrow$). The systematic exploration of $exec(St)$ thus corresponds to the standard systematic testing setting with no reduction of any kind.

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 in Fig. 2 has a main method shown on 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

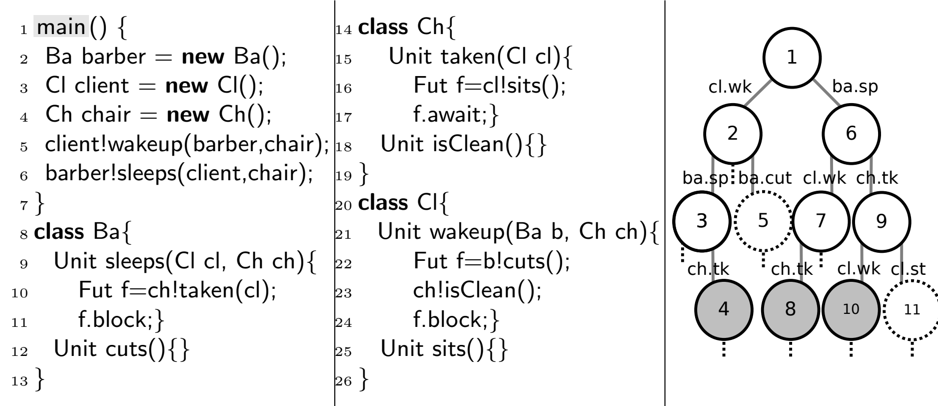

Fig. 2. Classical Sleeping Barber Problem (left) and Execution Tree (right)

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 line 11 (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 systematic testing tree of the main method 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 ba.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 have a deadlock. Let us outline five states of this derivation:

$$St_1 \equiv loc(ini,..)\cdot loc(cl,..,\{tsk(1,wk,..)\})\cdot loc(ba,..,\{tsk(2,sp,..)\})\cdot loc(ch,..) \xrightarrow{cl,1}$$
$$St_2 \equiv loc(cl,..,\{tsk(1,wk,f_0.block)\})\cdot loc(ba,..,\{tsk(3,cut,..),..\})\cdot fut(f_0,ba,3,12)\cdot.. \xrightarrow{ba,2}$$
$$St_3 \equiv loc(ba,..,\{tsk(2,sp,f_1.block)\})\cdot loc(ch,..,\{tsk(5,tk,..),..\})\cdot fut(f_1,ch,5,15)\cdot.. \xrightarrow{ch,5}$$
$$St_4 \equiv loc(ch,..,\{tsk(5,tk,f_2.await),..\})\cdot loc(cl,..,\{tsk(6,st,..),..\})\cdot fut(f_2,cl,6,25)\cdot..$$
$$\xrightarrow{ch,4} St'_4 \equiv loc(ch,..\{tsk(4,isClean,\epsilon),..\})\cdot..$$

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

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 L2, L3 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_2 the addition of the future variable created at L22. In St_3 we have executed task sleeps in the barber and added a new future term. In St_4 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 generating St'_4. 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 systematic testing. This is done by enhancing our standard semantics 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 $t_{\ell,tk,pp} \mapsto \langle n, \rho \rangle$, where:

- $t_{\ell,tk,pp}$ is a *macro-step identifier*, or *time identifier*, that includes: the identifiers of the location ℓ 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 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 $t_{0,0,1} \mapsto \langle 0, \rho_0 \rangle$, 0 being 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 for 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 $t_{\ell,tk,pp} \mapsto \langle n, \rho \rangle$ is added to $table$ in every macrostep if there has been progress in the execution, i.e., $S' \neq S$, n being 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_1	$t_{ini,main,1} \mapsto \langle 0, return \rangle$	$\emptyset$
St_2	$t_{cl,wakeup,21} \mapsto \langle 1, 24{:}f_0.block \rangle$	$fut(f_0, ba, cuts, 12)$
St_3	$t_{ba,sleeps,9} \mapsto \langle 2, 11{:}f_1.block \rangle$	$fut(f_1, ch, taken, 15)$
St_4	$t_{ch,taken,15} \mapsto \langle 3, 17{:}f_2.await \rangle$	$fut(f_2, cl, sits, 25)$

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'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 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: $suc(t_{\ell,tk,pp_0}, table) = \{t_{\ell,tk,pp_i} : t_{\ell,tk,pp_i} \in table, t_{\ell,tk,pp_i} \geq t_{\ell,tk,pp_0}\}$.

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. $\exists\, t_{stop} = t_{\ell,tk_0,pp_0} \in table,\ \rho_{stop} = status(t_{stop}) \in \{pp_1 : x.\mathsf{await}, pp_1{:}x.\mathsf{block}\}$,
2. $t_{resume} \equiv t_{\ell,tk_0,pp_1},\ fut(x, \ell_x, tk_x, pp(M)) \in S$,
3. $t_{async} \equiv t_{\ell_x,tk_x,pp(M)}, \nexists\, t \in suc(t_{async}, table)$ *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 ℓ 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_{ℓ,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 $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_2$ with $St_2 \equiv (S_2, table_2)$, since $t_{cl,wakeup,21} \in table_2$, $status(t_{cl,wakeup,21}, table_2) = [24{:}f.\mathsf{block}]$, $(f, ba, cuts, 12) \in St_2$ and $t_{ba,cuts,12} \notin table_2$. 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 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. We prove that our definition of deadlock is equivalent to the standard definition of deadlock in [6] (proof can be found in [16]).

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 systematic testing as follows. First, a state-of-the-art deadlock analysis is run, in particular that of [6], 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.

5.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [6] 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 ℓ by the program point at which it is created ℓ_{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* 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, ℓ_2, ℓ_3 and ℓ_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: $\ell_2 \xrightarrow{11:sleeps} taken \xrightarrow{17:taken} sits \xrightarrow{25:sits} \ell_3 \xrightarrow{24:wakeup} cuts \xrightarrow{12:cuts} \ell_2$. 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., ℓ_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 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 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. $\exists t_{L,T,PP} \mapsto \langle N, \rho\rangle$*, which means that there exists or will exist an entry of this form in table (time constraint)*
2. $\exists fut(F, L, Tk, p)$*, 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* $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, *and two fresh variables* L_i, Tk_i, ϕ *is defined as* $\phi(e_i \xrightarrow{p_i:tk_i} e_j \xrightarrow{p_j:tk_j} \ldots, L_i, Tk_i) =$

$$\begin{cases} \{\exists t_{L_i,Tk_i,_} \mapsto \langle _, \mathsf{sync}(p_i,F_i)\rangle, \exists fut(F_i,L_j,Tk_j,p_j)\} \cup \phi(e_j \xrightarrow{p_j:tk_j} \ldots, L_j, Tk_j) & \textit{if}\ \ e_j = tk_j \\ \{\mathsf{pending}(Tk_i)\} \ \cup\ \phi(e_j \xrightarrow{p_j:tk_j} \ldots, L_i, Tk_j) & \textit{if}\ \ e_j = \ell \end{cases}$$

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: $\{\ \exists t_{L_1,Tk_1,_} \mapsto \langle _, 11{:}F_1.block\rangle,\ \exists\ fut(F_1, L_2, Tk_2, 15),\ \exists t_{L_2,Tk_2,_} \mapsto \langle _, 17{:}F_2.await\rangle, \exists\ fut(F_2, L_3, Tk_3, 25),\ \mathsf{pending}(Tk_3),\ \exists t_{L_3,Tk_4,_} \mapsto \langle _, 24{:}F_3.block\rangle, \exists fut(F_3, L_4, Tk_5, 12), \mathsf{pending}(Tk_5)\}$. 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* $\forall t_{L_i,Tk_i,PP} \mapsto \langle N, \mathsf{sync}(p_i, F_i)\rangle \in \mathfrak{C}$*,* $fut(F_i, L_j, Tk_j, p_j) \in \mathfrak{C}$*, one of the following conditions holds:*

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

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 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 6 (Deadlock-cycle guided-testing (DCGT)). *Consider an abstract deadlock cycle c, and an initial state* St_0*. Let* $\mathfrak{C} = \phi(c, L_{init}, Tk_{init})$ *with* L_{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 $t_{ini,main,1} \mapsto \langle 0, return\rangle$, $t_{cl,wakeup,21} \mapsto \langle 1, 24{:}f_0.block\rangle$ and $t_{ba,cuts,12} \mapsto \langle 2, return\rangle\}$. $\mathsf{check}_{\mathfrak{C}}$ does not hold since $t_{L_1,Tk_1,24}$ 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. Since there are no more deadlock cycles, the search for deadlock detection finishes with this DCGT. Our methodology therefore explores 19 states instead of the 181 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* St_0*,* $\forall d \in exec(St_0)$ *if d is a derivation whose last state is deadlock, then* $\exists c \in C$ *s.t* $d \in exec_c(St_0)$*. (The proof can be found in App. A)*

6 Experimental Evaluation

We have implemented our approach within the SYCO tool, a testing tool for *concurrent objects* which is available at http://costa.ls.fi.upm.es/syco, where most

of the benchmarks below 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. This section summarizes our experimental results which aim at demonstrating the effectiveness and impact of the proposed techniques. The benchmarks we have used include: (i) classical concurrency patterns containing deadlocks, namely, *SB* is an extension of the sleeping barber, *UL* is a loop that creates asynchronous tasks and locations, *PA* is the pairing problem, *FA* is a distributed factorial, *WM* is the water molecule making problem, *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 also include here a peer-to-peer system *P2P*.

Table 6 shows, for each benchmark, the results of our deadlock guided testing (DGT) methodology for finding a representative trace for each deadlock compared to those of the standard systematic testing. Partial-order reduction techniques are not applied since they are orthogonal. This way we focus on the reductions obtained due to our technique per-se. For the systematic testing setting we measure: the number of solutions or complete derivations (column Ans), the total time taken (column T) and the number of states generated (column S). For the DGT setting, besides the time and number of states (columns T and S), 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 show the maximum time and maximum number of states measured among the different DCGTs (columns T_{max} and S_{max}). For instance, in the DGT for *HB* the analysis has found five abstract cycles, we only found a deadlock execution for two of them (therefore 3 of them were unfeasible), $44s$ being the total time of the process, and $15s$ 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 of DGT over systematic testing both assuming a sequential setting, hence considering values T and S of DGT (column T_{gain} for time and S_{gain} for number of states), and an ideal parallel setting, therefore considering T_{max} and S_{max} (columns T^{max}_{gain} and S^{max}_{gain}). The gains are computed as X/Y, X being the measure of systematic testing and Y that of 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 150s 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. Also, we write X^* when DGT times out.

Our experiments support our claim that testing complements deadlock analysis. In the case of programs with deadlock, we have been able to provide concrete traces 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. More importantly, the experiments demonstrate that

	Systematic			DGT (deadlock-per-cycle)					Speedup			
Bm.	**Ans**	T	S	**D/U/C**	T	T_{max}	S	S_{max}	T_{gain}	S_{gain}	T^{max}_{gain}	S^{max}_{gain}
HB	35k	32k	114k	2/3/5	44k	15k	103k	34k	0.73	0.9	2.15	3.33
FA	11k	11k	41k	2/1/3	2k	759	3k	2k	5.5	13.7	15.1	22.2
UL	>90k	>150k	>489k	1/0/1	133	133	5	5	>1.1k	>2.5k	>2.5k	>98k
SB	>103k	>150k	>584k	1/0/1	59	59	23	23	>2.5k	>25k	>2.5k	>25k
PA	>121k	>150k	>329k	2/0/2	42	4	12	6	>3.6k	>27k	>38k	>55k
WM	>82k	>150k	>380k	1/0/2	>150k	>150k	>258k	>258k	1*	1.47*	1*	1.47*
fFA	5k	7k	25k	0/1/1	5k	5k	11k	11k	1.61	2.35	1.61	2.35
fP2P	25k	66k	118k	0/1/1	34k	34k	52k	52k	1.96	2.28	1.96	2.28
fPA	7k	7k	30k	0/2/2	4k	2k	9k	4k	1.75	3.33	3.73	6.98
fUL	>102k	>150k	>527k	0/1/1	410	410	236	236	>1k	>2k	>1k	>2k

Table 1. Experimental results: Deadlock-guided testing vs. systematic testing

our DGT methodology achieves a notable reduction of the search space over systematic testing in most cases. Except for benchmarks HB and WM which are explained below, the gains of DGT both in time and number of 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 SB, UL and PA). In general, the generated constraints for unfeasible cycles are often not able to guide the exploration effectively (e.g. in HB and WM). Even in these cases, DGT outperforms systematic testing in terms of scalability and flexibility. Let us also observe that the gains are less notable in deadlock-free examples. That is because, 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.

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 [6, 7] and thread-based programs [11, 13], 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 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 [4, 9, 10, 15], where sometimes has been combined with static information [2, 8]. As regards combined approaches, the approach in [8] 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. 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 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.

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 *H*VC, LNCS 3875. Springer, 2006.
3. E. Albert, P. Arenas and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *F*ORTE'14, Springer.
4. M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In ICST'13, pages 154–163. IEEE, 2013.
5. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *P*roc. POPL'05, pp. 110-121. ACM, 2005.
6. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *F*ORTE'13, LNCS 7892. 2013.
7. E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.
8. 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.
9. P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *P*roc. of PLDI'09. ACM, 2009.
10. A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report, 2013.
11. S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *P*arallel and Distributed Debugging. ACM, 1991.
12. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *P*roc. of ICSE, pages 386–396. IEEE, 2009.
13. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM* TCS, 1997.
14. K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *P*roc. FASE'06, LNCS 3922, pp. 339-356. Springer, 2006.
15. K. Havelund, Using Runtime Analysis to Guide Model Checking of Java Programs, Proceedings of the 7th International SPIN Workshop, Springer-Verlag, 2000.
16. E. Albert, M. Gómez-Zamalloa, et.al. Combining Static Analysis and Testing for Deadlock Detection. http://costa.ls.fi.upm.es/papers/costa/AlbertGI15.pdf.