

Received February 2, 2021, accepted March 1, 2021, date of publication March 10, 2021, date of current version March 30, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3065421

Deadlock-Guided Testing

MIGUEL GÓMEZ-ZAMALLOA¹ AND MIGUEL ISABEL² 

¹Departamento de Sistemas Informáticos y Computación, Facultad de Informática, Complutense University of Madrid, 28040 Madrid, Spain

²Lenguajes, Sistemas Informáticos e Ingeniería de Software, E.T.S. de Ingenieros Informáticos, Universidad Politécnica de Madrid, 28660 Madrid, Spain

Corresponding author: Miguel Isabel (miguelis@fi.upm.es)

This work was supported in part by the Spanish Ministerio de Ciencia, Innovación y Universidades (MCIU) through the Agencia Estatal de Investigación (AEI) and Fondo Europeo de Desarrollo Regional (FEDER) (EU) Project under Grant RTI2018-094403-B-C31, in part by the Comunidad de Madrid (CM) Projects co-funded by the Fondos Estructurales y de Inversión Europeos (EIE Funds) of the European Union under Grant S2018/TCS-4314 and Grant S2018/TCS-4339, and by the Ministerio de Educación Cultura y Deporte through the Spanish Formación del Profesorado Universitario (FPU) under Grant FPU15/04313.

ABSTRACT Static deadlock analyses might be able to verify the absence of deadlock. However, they are usually not able to detect its presence. Moreover, when a potential deadlock is detected, they provide little (and often no) information that can help the user in finding the source of the anomalous behaviour. This paper proposes a testing methodology that combines static analysis and symbolic execution for effective deadlock detection in asynchronous programs. When the program features a deadlock, our testing methodology provides an effective technique to catch deadlock traces. While if the program does not have deadlock, but the static deadlock analysis inaccurately spotted it, our approach is able to prove deadlock freedom (up to the limit of the performed symbolic exploration).

INDEX TERMS Deadlock analysis, deadlock detection, symbolic execution, testing, test case generation, verification.


I. INTRODUCTION

A deadlock occurs when a concurrent program reaches a state in which a set of processes or tasks are waiting for some event or condition and they are all unable to change the state, hence do not allowing the program to make any progress. Together with data-races, deadlocks are one of the most important sources of bugs in concurrent programming. A main challenge of verification and static analysis tools is thus proving *deadlock freedom*, and, conversely, a main challenge of testing tools is performing *deadlock detection*. We consider a general-purpose concurrent language, with distributed locations, asynchronous communication among them by means of asynchronous method calls which create the corresponding task at the specified location, with no shared memory among the different locations, and with an operation for *blocking* synchronization with the termination of asynchronous calls. This provides the foundations for the concurrency model of languages used in industry, e.g., *Erlang* and *Scala*, and libraries used in mainstream languages, e.g., *Akka*.

Also, we consider a setting where a task cannot be interrupted until its end, except if it explicitly blocks waiting for the termination of another task, i.e., we have *cooperative scheduling*. In this context, when a location finishes the execution of a task, it chooses any of the other available tasks

in the location and starts its execution. Hence, in order not to lose any possible behavior of the program, in particular deadlocks, all possible combinations of task interleavings must be considered.

Static analysis and testing usually follow two different approaches when it comes to reason about deadlocks. The approaches can often complement each other and thus it seems quite natural to combine them. Static analysis usually evaluates an application by examining all possible execution paths and variable values but without executing it. Hence, it might be able to prove deadlock freedom and also to catch deadlocks that could not show up until days or even weeks after releasing the application. This feature is especially relevant in security assurance where the attackers often attempt to exercise an application in unpredictable ways. However, due to the use of approximations, most static analyses are not able to verify the presence of deadlocks, being able to verify only its absence. In other words, most static analyses can produce false positives. Moreover, when a potential deadlock is detected, state-of-the-art analysis tools [1]–[4] provide little (and often no) information on the source of the deadlock. In particular, for complex deadlocks caused by the interactions of possibly many different locations and tasks, it might be very important to know the concrete deadlock trace, i.e., the locations that are involved, and, the exact sequence of task interleavings that cause the deadlock. This can be an essential information to allow the programmer identifying and fixing the problem.

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana .

In contrast, testing consists in executing the application. In dynamic testing, the application is executed for concrete input values, while in static testing it is executed symbolically (i.e., without any knowledge on the concrete inputs). Most errors in concurrent programs, including deadlocks, may manifest only on specific sequences of task interleavings. Hence, in principle, in order not to lose any possible behavior of the program, the testing process must systematically consider all possible ways in which the different tasks can interleave their execution. Applying *systematic testing* [5]–[7] for deadlock detection allows obtaining detailed traces for the deadlocks that are found. This is one of the main advantages of testing techniques w.r.t static deadlock analysis since it can help the user in finding the source of the deadlock. However, there are two main complications: (1) Even applying aggressive Partial Order Reduction (POR) techniques [8]–[11] to avoid the exploration of redundant executions, there is usually a state-explosion problem. This is a general threat to the application of systematic testing on concurrent programs. (2) It is well known that dynamic testing can only verify the presence of errors, but not their absence. Hence, deadlock freedom can only be guaranteed for terminating programs with a concrete set of inputs. Instead, in static testing, deadlock freedom can be ensured for a program with unknown inputs. However, as soon as there is a loop in the program whose termination depends on unknown data, it is required to use a termination criterion in order to guarantee finiteness of the process. E.g, a classical termination criterion consists in limiting the number of iterations of each program loop. Hence, in this context, deadlock freedom can only be ensured for the particular termination criterion used.

This article proposes a testing methodology that combines static analysis and symbolic/systematic execution for effective deadlock detection. Essentially, we firstly invoke an existing static deadlock analysis [1] in order to get *abstract* descriptions of potential deadlock cycles. Such abstract deadlock cycles are used in a second step to steer the symbolic execution in order to find associated deadlock traces (or discard them), avoiding as much as possible the exploration of paths that do not lead to these deadlock cycles. When the program features a deadlock, our combined use of analysis and symbolic/systematic execution provides an effective technique to catch deadlock traces. The development of such combined framework requires: (1) pruning, as soon as possible, those paths that are guaranteed to be deadlock free, while keeping the exploration in order to find deadlock paths (if there are), and (2) handling, together with the path constraints that arise during symbolic execution, the constraints describing the potential deadlock cycles.

This article is a revised and extended version of a conference paper that appeared in the proceedings of iFM'16 [12]. The fundamental contribution w.r.t. [12] is the extension, implementation and experimental evaluation of our deadlock-guided approach to the *static* testing setting with symbolic execution. This allows applying our approach to find deadlock traces to contexts where the input data is unknown.

In contrast, [12] is defined for *dynamic* testing which requires full knowledge of the input data. This has a clear theoretical interest as it generalizes the previous work to a static setting. Besides, the practical impact of such extension is important as now we can use it to prove deadlock freeness, i.e., if we are able to symbolically execute the whole program without finding any deadlock trace, nor termination problem, then the program is proven to be deadlock free. Also, this allows applying our approach for performing test-case generation via symbolic execution [13], [14].

The original research was part of the Master thesis of one of the authors [15] where we explored a proof-of-concept of the approach in the Constraint Logic Programming setting.

II. ASYNCHRONOUS PROGRAMS: SYNTAX AND SEMANTICS

We formalize our approach for the ABS *concurrent objects* language [16]. It is based on a distributed programming model with explicit locations. This concurrency model is in fact quite general since it subsumes both *rendez-vous synchronization* as in Ada [17]; actor-based *asynchronous message passing* [18] (see e.g., Erlang [19] and Akka [20], [21]); *fork/join parallelism* [22]; *futures* [23] in Argus [24] and MultiLisp [25] and in *active-object languages and frameworks* such as ConcurrentSmalltalk [26], ABCL [27], Eiffel Parallel [28], CJava [29], and ProActive [30].

Locations model computation entities each of which contains a call stack, a local storage (or heap) and a buffer of scheduled tasks (initially empty). Locations select non-deterministically tasks from their buffer for execution. A task hence executes on its own location being able to access its own local storage, schedule asynchronously tasks to other locations or on its own location, and synchronize with the completion of other tasks. Locations hence behave as *concurrent objects*. *Future variables* [31] are used for synchronizing program execution with the termination of asynchronous tasks. A future variable acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete. When the future variable is ready, the result can be retrieved. We use the syntax $f = x ! m(\bar{z})$ to denote an asynchronous call of task or method $m(\bar{z})$, which is sent to location x and whose result is associated with future variable f . Instruction $r = f.get$ allows blocking the execution until the task executing m that is associated to f terminates, and it retrieves the result in r . The declaration of a future variable f is written $Fut\langle T \rangle f$, where T is the type of the result r . When the execution of a task in a location finishes, the location selects (non-deterministically) from its buffer another task for execution. The number of distributed locations does not need to be known statically. The instruction **new** allows to dynamically create new locations. For instance, the instruction $b = \mathbf{new} \text{ Location}()$; creates a distributed location of type `Location` which is referenced by b . The program consists of a set of classes that define the types of locations.

$$\text{(MSTEP)} \frac{\text{selectTask}(S) = tk, S = \text{loc}(o, \perp, h, Q \cup \{tk\}) \cdot S'}{\text{loc}(o, tk, h, Q \cup \{tk\}) \cdot S' \xrightarrow{o \cdot tk} S''} S \xrightarrow{o \cdot tk} S''$$

FIGURE 1. Macro-step semantics rule of asynchronous programs.

Each of them defines a set of fields and methods of the form $M ::= T \ m(\bar{X})\{S\}$, where statements s take the form $s ::= s; s \mid x = e \mid \text{if } e \text{ then } s \text{ else } s \mid \text{while } e \text{ do } s \mid \text{return } x; \mid \text{b=new} \mid f = x ! m(\bar{z}) \mid x = f.\text{get}$. All methods must return something, hence **void** methods are by-default expressed as **int** methods returning 0. The syntax of expressions e and types T is deliberately left open for generality.

A. SEMANTICS OF ASYNCHRONOUS PROGRAMS

States of our asynchronous programs are defined as a set of locations and future variables $\text{loc}_0 \cdots \text{loc}_n \cdot \text{fut}_0 \cdots \text{fut}_m$. Locations are represented as terms of the form $\text{loc}(o, tk, h, Q)$ where o is the unique identifier of the location, tk is the unique identifier of the task which is currently executing, which is said to hold the location's lock. We write \perp if the location is currently not executing any task, i.e., the lock of the location is currently free. h denotes the location's local storage (or heap), and Q the location's buffer of scheduled tasks. Future variables are represented as terms of the form $\text{fut}(id, o, tk, m, r)$ where id is a unique identifier, o is the identifier of the location which is executing the task tk whose termination is synchronized via the future variable, m is the program point where tk starts and r is the value which is returned by tk , or \perp if the execution of tk is not completed. Tasks are represented as terms of the form $\text{tsk}(tk, m, l, s)$ where tk is the task identifier, m is the name of the method associated with the task, l maps local variables to their associated current values, and s is the sequence of instructions to be executed. We consider the initial entry of the program is the **main** method which is included in the **Main** class.

We define the semantics of our asynchronous programs via macro-step semantics [7]. This is possible because locations do not share anything of their states and because the execution of tasks can not be interrupted. The transition rule " \longrightarrow " denotes the sequential execution (without interleavings) of all instructions of a task until it reaches a **return** or a **get** statement.

The semantics rule for " \longrightarrow ", referred to as MSTEP, is shown in Figure 1, and selects non-deterministically an available task from one *active* location in the state (i.e., a location with a non-empty task buffer). The *micro-step* transitions \rightsquigarrow define the evaluation of the statements within a given location. The micro-step transition rules for the different statements of the language are standard and are thus omitted. They can be found in [12].

We define a *derivation* or *execution* $E \equiv S_0 \longrightarrow \cdots \longrightarrow S_n$ as a sequence of macro-steps (applications of rule MSTEP). If S_0 is the initial state and $\nexists S_{n+1} \neq S_n$ such that $S_n \longrightarrow S_{n+1}$ we say that the derivation is *complete*. Due to undeterminism, there can be different derivations from a given state. We use

$\text{exec}(S)$ to denote the set of all possible complete derivations starting at a given state S . Transitions are sometimes labeled with $o \cdot tk$, being o and tk the names of the selected location and task in rule MSTEP. The systematic exploration of $\text{exec}(S)$ thus corresponds to the standard systematic testing setting in which all possible explorations are performed without eliminating any redundancy. The semantics will be exemplified in the next two sections.

III. MOTIVATING EXAMPLE

Our running example simulates a simple communication protocol among a database and n workers. Our implementation in Figure 2 has three classes, a **Main** class which includes the **main** method, and classes **Worker** and **DB** implementing the workers and the database, respectively. The **main** method just calls method **simulate** with the number of workers to create in its parameter (in this case 1). Method **simulate** creates the database and the n workers, and spawns tasks **register** and **work** on each of them, respectively. A worker executing task **work** simply requires the access to the data stored by the database (spawning a task **getData**) and thus, it gets blocked awaiting for the result until the **data** are returned by the database and stored in its own **data** field. Method **getData** of the database checks if the caller worker has been previously registered as its client, in which case it returns the value of its **data** field. Otherwise, it returns **null**. The **register** method of the database registers the provided worker reference adding it to its **clients** list field. In case **checkOn** is true, before adding the worker, it makes sure that the worker is online. This is done by spawning a task **ping** with a concrete value and blocking until it gets the result with the same value.

We can observe different scenarios depending on the sequence of interleavings among the different tasks involved in the communication protocol:

- (I) As one would expect, i.e., with $w.\text{data} = \text{db.data}$,
- (II) with $w.\text{data} = \text{null}$, or,
- (III) in a deadlock.

Scenario (I) happens when the database has registered the worker as its own client (assignment at line 25) before it executes task **getData**. Scenario (II) happens when the database executes task **getData** before the assignment at line 25. In scenario (III), we can find a deadlock situation. It happens if both tasks **register** and **work** start executing before tasks **getData** and **ping**.

IV. SYMBOLIC EXECUTION

Symbolic execution [14], [32]–[36] is a well-known static testing technique which is able to execute a program without knowledge of its inputs. It hence allows to statically test fragments of a program independently. The execution is performed using *constraint variables*, that represent symbolic values, rather than concrete values. As a result, symbolic execution produces a set of equivalence classes of inputs, which are often called *path conditions*. A path condition essentially consists of the *constraints* or conditions on the input data that make the program to traverse the corresponding program

```

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

```

FIGURE 2. Communication protocol among a DB and n workers.

path. Path conditions also include relations between the input and output of the program. For instance, let us consider this function

```
int abs(int n){if (n<0) return -n; else return n;}
```

The following two path conditions are obtained: $\{(N < 0, Z = -N), (N \geq 0, Z = N)\}$ where Z denotes the return value of the function. The first equivalence class characterizes program executions taking the **then** branch, with input condition $N < 0$ and with relation $Z = -N$ for the output. The second characterizes executions of the **else** branch with input condition $N \geq 0$ and relation $Z = N$ for the output. Note the use of uppercase letters on constraint variables to distinguish them from ordinary program variables.

Example 1: Consider a context in which there is a location *db* whose buffer contains a task *getData*. The input parameter of the task is a constraint variable *W*, and its local heap is mapped to constraint variables as follows: $\{data \mapsto D, clients \mapsto Cls, checkOn \mapsto CO\}$. We obtain three different derivations with the following path conditions:

- 1) $\langle Cls=[], Ret=null \rangle$, i.e., the task returns **null** if the clients list does not contain any client,
- 2) $\langle Cls=[X], X \neq W, Ret=null \rangle$, i.e., if the worker *W* is not in the list it returns **null**,
- 3) $\langle Cls=W::Cls', Ret=D \rangle$, i.e., the task returns the stored data *D* if worker *W* is in the head of the clients list.

The symbolic execution of function **contains** produces infinite lists of increasing length with and without worker *W* at every possible position of the list. This fact can be observed in the third path condition where the new constraint variable Cls' appears.

Symbolic execution is in general unbounded since there can be infinite derivations as soon as there is a loop in the program whose termination depends on unknown data. It is hence required to use a termination criterion in order to guarantee finiteness of the process. It is also desirable to

use a criterion which is meaningful in terms of the coverage which is achieved by the symbolic execution, and ideally, which is related to well-known test adequacy criteria (branch coverage, path coverage, etc.). Therefore, such a criterion is often refer to as a termination/coverage criterion. A classical criterion to guarantee termination is the *loop-k* criterion, which establishes a limit on the number of loop iterations which are allowed. E.g., in the above example, with $k=1$, the symbolic execution produces the three derivations shown and stops. However, in presence of concurrency, program termination is not guaranteed if we simply rely on this criterion for the symbolic execution of all programs tasks. As shown in [13], in the context of concurrent programs, there are more factors that can threaten termination. Specifically, in symbolic execution of concurrent programs, (1) there can be an unbounded number of task switches, and, (2) there can be an unbounded number of location creations. This is further studied in [13]. In this article the criteria defined in [13] are simply adopted.

Example 2: Figure 3 shows the symbolic/systematic execution tree computed for the *simulate* method with an unknown input parameter *N* and *loop-k* with $k=1$ as termination/coverage criterion. Derivations with dotted nodes and edges are not deadlock, whereas those ending with a squared node are deadlock. We can observe that seven complete derivations are explored. The leftmost branch produces the path-condition $\langle N \leq 0 \rangle$. Its final state does not contain any worker and the database does not execute any task along the derivation. The remaining branches finish with the condition $\langle N = 1 \rangle$ and correspond to the scenarios described in Section III. Namely, the second and third branches finish with the expected output state (scenario I), the fourth and fifth derivations lead to deadlock states (scenario III). Finally, the sixth and seventh branches refer to scenario II. Let us notice the dashed branch is pruned since the loop limit is $k = 1$, then branches with $\langle N > 1 \rangle$ must not be explored.

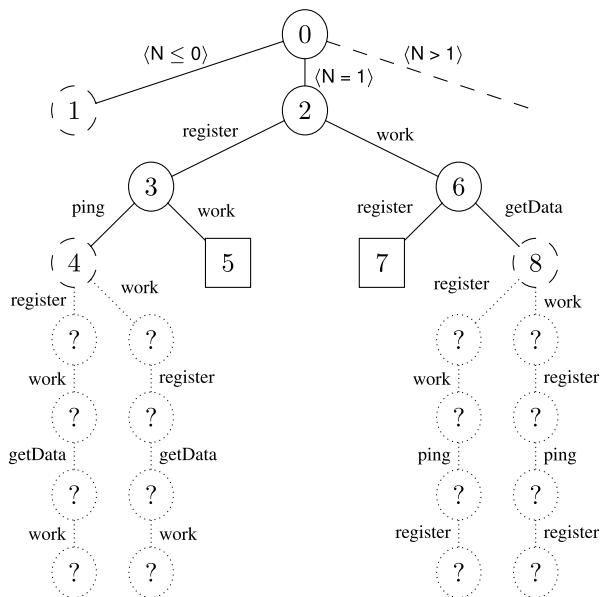


FIGURE 3. Symbolic/systematic execution tree from an initial context with task **simulate(N)**.

Let us explain in detail the explored derivations. Symbolic execution starts with a free constraint variable N . In line 7 a new database location is created. Then, in line 8, since variable N is unknown, symbolic execution branches. In the first branch the path constraint $N \leq 0$ is added and hence the execution does not enter the while loop and finishes with such a path constraint and a database location. In the second branch the path constraint $N > 0$ is added, the execution enters the loop, a worker location is created and tasks **register** and **work** are scheduled on the database and worker locations respectively. It then adds the constraint $N_2 = N - 1$, evaluates the loop guard and branches again. This time, in the first branch we get the constraint $N_2 \leq 0$ (hence implying $N = 1$), the execution exits the loop and reaches the return at line 14. It then proceeds to pick an available task for execution. In the other branch the constraint $N_2 > 0$ is added (hence implying $N > 1$), the execution attempts to enter again the loop but the loop- k limit stops it. Let us continue with the derivation with $N = 1$ assuming that task **register** is selected. In the second macro-step task **register** is hence executed on location db . It then gets blocked awaiting for the termination of **ping**, but only temporarily until task **ping** finishes. Similarly, location w also gets temporarily blocked in the **get** instruction at line 38, but will resume as soon as task **getData** finishes. The third branch is similar to the previous one, the only difference is the order of execution of tasks **work** and **register**, since in the former, there is a state where both locations are temporarily blocked, whereas in the latter, task **register** finishes its execution before location w gets blocked because of task **work**. Indeed, both branches can be considered as equivalent according to the partial order reduction theory. However, in the fourth derivation, task **register** is executed (and location db gets blocked awaiting for the execution of task **ping**), and then task **work** is also executed (blocking

location w waiting for task **getData** to be executed). The future variable waited by the **get** instruction at line 38 will never be ready since it is bound to a task to be executed by a blocked location. This is therefore a deadlock state. Finally, the fifth, the sixth, and the seventh derivations are analogous to the fourth, the second, and the third branches, respectively. Let us outline four states of the fourth derivation:

$$\begin{aligned}
 S_0 &\equiv loc(0, 0, \dots, \{tsk(1, \mathbf{simulate}, \dots)\}) \xrightarrow{0,1} \\
 S_2 &\equiv loc(0, 0, \dots) \cdot loc(db, \perp, h_{db}, \{tsk(2, \mathbf{reg}, \dots)\}) \cdot \\
 &\quad loc(w, \perp, h_w, \{tsk(3, \mathbf{wo}, \dots)\}) \xrightarrow{db,2} \\
 S_3 &\equiv loc(db, 2, \dots, \{tsk(2, \mathbf{reg}, f_4, \mathbf{get})\}) \cdot \\
 &\quad fut(f_4, w, 4, 41, \perp) \dots \\
 &\quad loc(w, \perp, \dots, \{tsk(4, \mathbf{ping}, \dots), \dots\}) \xrightarrow{w,3} \\
 S_5 &\equiv loc(w, 3, \dots, \{tsk(3, \mathbf{work}, f_5, \mathbf{get}), \dots\}) \cdot \\
 &\quad fut(f_5, db, 5, 29, \perp) \dots \\
 &\quad loc(db, 2, \dots, \{tsk(5, \mathbf{getData}, \dots), \dots\})
 \end{aligned}$$

The second state S_2 is obtained after executing task **simulate** when the value of input parameter N is 1. The first macro-step creates two new locations at lines 7 and 9 respectively. It also spawns two tasks at lines 10 and 11 respectively in their corresponding buffers. Observe that each location and task receives a unique identifier (numbers are assigned as task identifiers, whereas locations receive short names, e.g., **reg** for **register**). At state 3, the location db has executed the task **register** and adds the future variable created at line 24 to the new state. At state S_5 , the location w executes task **work** and a new future variable is added to the state. As we will see in Section V, the resulting state is deadlock. For clarity, along the examples of this paper, we use location and task names instead of numeric identifiers.

V. DEADLOCK-GUIDED TESTING

This section presents our testing methodology that combines systematic/symbolic execution and static analysis for effective deadlock detection. As already mentioned, the systematic exploration of all different task interleavings poses scalability problems, even if we apply leading-edge POR technology to detect redundancies (see Section VII). This problem is aggravated in the case of symbolic execution due to its inherent non-determinism produced in branching statements involving unknown data.

In our working example (see Figure 3) POR technology is at best able to detect that the third and fifth derivations are redundant, hence saving the exploration of a total of 8 states. The main goal of our deadlock-guided technique is however to detect and stop, as soon as possible, derivations that will not lead to deadlock. Namely, in our working example, the goal, and the challenge, is to stop the second and third, and also the fourth and fifth derivations at states 4 and 8 respectively, hence saving the exploration of a total of 16 states.

In Section V-A, we first propose an extension of the language semantics and program state to carry out the

scheduling of the current execution and the status of tasks. Then, in Section V-B, we present *deadlock-guided testing*, our new methodology to guide the exploration towards deadlock traces. This methodology relies on the above information, on the data reported by a static deadlock analyzer, and on a symbolic/systematic execution engine.

This way we extend the technique presented in [12] to the context of symbolic execution. This allows applying our approach to find deadlock traces or to ensure deadlock freeness in contexts where the input data is possibly unknown. Instead, in [12] it is required to have full knowledge of the input data (i.e. the execution must start from a *main* method that creates a concrete initial state). As a consequence, the applicability of the methodology presented in [12] is much more limited. Besides, we have re-formalized certain aspects of our methodology in [12], on one hand, improving the overall clarity of the presentation, and also, allowing it to be applied for different deadlock-based testing adequacy criteria (see Section V-C). Finally, we have also added a soundness proof of our methodology (see Section V-B3).

A. THE INTERLEAVINGS TABLE AND ENHANCED SEMANTICS

The interleavings table stores all scheduling decisions among the tasks involved in the execution. When an execution is fully explored, we can use this table to recover the exact ordering of the performed macro-steps. Specifically, the interleavings table IT is a mapping with entries of the form $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$, where:

- $t_{o,tk,pp}$ is a *macro-step identifier*, or *time identifier*. It includes the identifiers of the location o and task tk that have been selected in the macro-step. Finally, pp indicates the program point of the first instruction that the current macro-step executes;
- n is an integer that indicates the clock-time when the current macro-step starts its execution;
- ρ is the status of the current task and the cause of the stop of the macro-step. It can take two different values: (i) **pp:f.get** when the macro-step finished at program point pp on a **get** instruction on a future variable f which is not ready; or (ii) **return** when the task has already finished.

As notation, the relation $t \in IT$ denotes the existence of an entry $t \mapsto \langle n, \rho \rangle \in IT$.

Example 3: In Figure 4, we show the interleavings table computed for the fourth execution in Figure 3 of Section III. This table has as many entries as macro-steps in the execution. Thanks to the time values assigned to each time identifier, we can know the scheduling decisions that lead to such an execution. Finally, we show in the right column the future variables appearing in each state. These variables store not only the location and task they are bound to, but also the return information (\perp in both cases, since their associated tasks have not finished).

1) ENHANCED SEMANTICS FOR DEADLOCK DETECTION

The semantics of our asynchronous language is extended by updating rule MSTEP as in Figure 5. Function $clock(n)$

S_2	$t_{0,simulate,6} \mapsto \langle 0, \text{return} \rangle$	\emptyset
S_3	$t_{db,reg,22} \mapsto \langle 1, 25:f_4.\text{get} \rangle$	$fut(f_4, w, \text{ping}, 41, \perp)$
S_5	$t_{w,work,36} \mapsto \langle 2, 38:f_5.\text{get} \rangle$	$fut(f_5, db, \text{getData}, 29, \perp)$

FIGURE 4. Interleavings table for Example 3.

(MSTEP2)

$$\begin{aligned}
 & \text{selectTask}(S) = tk, \\
 & S = \text{loc}(o, \perp, h, \mathcal{Q} \cup \{tsk(tk, m, l, pp : s)\}) \cdot S', \\
 & \text{loc}(o, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, pp : s)\}) \cdot S' \diamond \rho_0 \xrightarrow{o \cdot tk} S'' \diamond \rho, \\
 & S \neq S'', \text{clock}(n), IT'' = IT \cup \{t_{o,tk,pp} \mapsto \langle n, \rho \rangle\}, \\
 & \text{check}_{\mathcal{E}}(S, IT) \\
 & \hline
 & (S, IT) \xrightarrow{o \cdot tk} (S'', IT'')
 \end{aligned}$$

FIGURE 5. MSTEP2 rule for deadlock-guided testing.

represents a clock, starting at 0, which is increased by one in every call to function $clock$, and returns the current value n .

For now, we should ignore function $\text{check}_{\mathcal{E}}$. It will be defined in Section V-B2. Essentially, there are two new aspects: (1) We extend the state with the status ρ , attached by means of the symbol \diamond . Its initial value, written ρ_{\emptyset} , is a fresh variable that annotates the value **pp:f.get** when the task being executed stops because of a **get** instruction, awaiting for the termination of the task bound to f , or the value **return** when the current task has completely finished. (2) Every program state is extended with the interleavings table IT . In case there has been progress in the execution, i.e., $S'' \neq S$; and we also add a new entry $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$ to IT , where o and tk are the current location and task identifiers, respectively, chosen by function selectTask ; pp is the first instruction executed by the current macrostep; n is the current clock time. Finally, ρ stores the cause of the end of the current macrostep.

B. GUIDING TESTING USING STATIC DEADLOCK ANALYSIS

In this section we present our new methodology, a combination of static analysis and systematic/symbolic execution for effective deadlock detection that works as follows. First, we run a state-of-the-art deadlock analyzer [1]. This analyzer reports a set of abstract *deadlock cycles*. If this set does not contain any cycle, then the analyzer is able to prove deadlock freedom. Otherwise, we use these abstract cycles to perform a systematic/symbolic execution which is steered towards executions that might lead to deadlock, discarding as soon as possible executions leading to a deadlock-free final state. This process has the following two goals: (1) finding concrete deadlock traces related to the abstract cycles, and, (2) discarding spurious abstract cycles, produced by the loss of precision of the analysis. In case we are able to discard all cycles, we prove deadlock freedom up to the limits imposed on the symbolic execution to ensure termination (e.g., number of the loop iterations). The experimental evaluation in Section VI shows that this methodology is able to significantly reduce

the search space compared to the full exploration performed by the standard systematic/symbolic execution.

1) DEADLOCK ANALYSIS AND ABSTRACT DEADLOCK CYCLES

Given a program state S , its dependency graph G_S is formalized in [1]. If there exists a cycle in G_S , then S is a deadlock state. The abstraction of this graph is called abstract dependency graph \mathcal{G} .

Definition 1 (Abstract Dependency Graph \mathcal{G}): Let \mathcal{L} and \mathcal{T} be the set of abstract locations and abstract tasks, respectively. The abstract dependency graph is a directed graph \mathcal{G} , whose nodes are \mathcal{L} and \mathcal{T} , and whose edges are:

- 1) *Location-Task:* $l \xrightarrow{pp:tk} tk'$ iff there is a location $l \in \mathcal{L}$ and there are tasks $tk, tk' \in \mathcal{T}$ such that l is blocked in an instruction $pp: x=y.get$ of tk awaiting for the termination of tk' .
- 2) *Task-Task:* $tk \xrightarrow{pp:tk} tk'$ iff there are tasks $tk, tk' \in \mathcal{T}$ such that tk is awaiting in an instruction $pp: x=y.get$ for the termination of task tk' .
- 3) *Task-Location:* $tk \xrightarrow{pp:tk} l$ iff there is a task $tk \in \mathcal{T}$ and a location $l \in \mathcal{L}$ such that tk is awaiting to be executed by location l , where pp is the entry program point of task tk .

If \mathcal{G} does not contain any abstract deadlock cycle, then the program is deadlock-free. Otherwise, the deadlock analysis of [1] returns a set of abstract deadlock cycles in \mathcal{G} . Each cycle is of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \dots \xrightarrow{p_n:tk_n} e_1$, where p_1, \dots, p_n are program points, tk_1, \dots, tk_n are task abstractions, and nodes e_1, \dots, e_n are either location abstractions or task abstractions. Each arrow $e \xrightarrow{p:tk} e'$ should be interpreted like “location or task e is waiting for the termination of location or task e' due to a **get** instruction at program point p of task tk ”. We can distinguish three kinds of arrows: (1) *task-task*, a task is blocked waiting for the termination of another one; (2) *task-location*, a task is awaiting to be executed by a (possibly) blocked location; and (3) *location-task*, the location is blocked in a **get** instruction awaiting for the termination of the task. Let us notice that *location-location* arrows do not happen using the analyzer in [1].

This analyzer allows us to perform both task and location abstractions at different levels of accuracy. For the sake of simplicity, we assume a simple abstraction for our formalization: a concrete location o created at program point pp is abstracted by o_{pp} , and each task is abstracted by the corresponding method name. Such a simple abstraction could be rather imprecise as soon as there are loops which create many locations at the same program point and invoke the same method on the different locations. The analyzer uses *points-to analysis* as the basis to infer such abstractions. Much more clever abstractions could be used during the points-to analysis making the deadlock analysis, and consequently, our deadlock-guided testing approach potentially more accurate.

Example 4: In our working example we have two abstract locations: o_7 , which represents a concrete location **database**

created at program point 7; and o_9 , which represents the n concrete locations **worker**, created in the loop at program point 9. We also have four abstract tasks: **register**, **getData**, **work** and **ping**. The deadlock analysis reports the following cycle: $o_7 \xrightarrow{25:register} ping \xrightarrow{41:ping} o_9 \xrightarrow{38:work} o_7$. The first arrow indicates that the location created at program point 7 is blocked awaiting for task **ping** to be executed because of the **get** instruction at program point 25 of task **register**. Also, the second arrow means that task **ping** is awaiting to be executed on the (possibly) blocked location o_9 . The deadlock analysis produces as a result a set of this kind of abstract deadlock cycles. It can be seen, however, that such information can be rather insufficient for a user to figure out how these dependencies may arise, and, more importantly, the scheduling decisions that are causing the deadlock.

Let function α denote a mapping from concrete cycles in the dependency graph to their corresponding cycles in the abstract dependency graph.

Definition 2 (Representative of an Abstract Deadlock Cycle c): An extended state (S, IT) is a representative of an abstract deadlock cycle c if $\exists \gamma \in G_S$ such that $\alpha(\gamma) = c$.

We say that every derivation leading to a state which is a representative of an abstract deadlock cycle is a *deadlock trace*.

2) GUIDING TESTING TOWARDS DEADLOCK CYCLES

This section presents a novel technique to improve the performance of the symbolic execution guiding it towards executions that might lead to a representative of a given abstract deadlock cycle, by discarding as soon as possible derivations that cannot be a representative. This novel technique works as follows: (1) we generate *deadlock-cycle constraints* using a given abstract deadlock cycle c reported by the analyzer. Each of these constraints must hold in all states of derivations whose final state is a representative of c . (2) The execution semantics is extended to support these constraints, with the aim of avoiding the exploration of the remaining executions as soon as one of these constraints is not satisfied.

Before going into the formalization and technical details of the generation of the deadlock-cycle constraints and the semantics extension, let us illustrate the main intuitions through our working example.

α : AN ILLUSTRATIVE EXAMPLE

The first two arrows of the deadlock cycle in Example 4 indicate that, in order to reach a deadlock, a task **register** and a task **ping** must arise at some state during the computation (or must already be present), so that **register** blocks waiting for **ping**, and **ping** has not yet finished. Similarly, the last two arrows of the cycle indicate that a task **work** and a task **getData** must also arise at some state during the computation (or must already be present), so that **work** blocks waiting for **getData**, and **getData** has not yet finished either. All this information will be encoded in our deadlock-cycle constraints. Therefore,

if we are interested only in deadlock executions, as soon as we can infer that (i) at least one of such tasks cannot arise from an state, i.e. it is not reachable, we can safely stop the exploration. Also, we can stop the exploration if (ii) at some state, (iia) we already have some of such tasks, (iib) we can infer that no more representatives of them can arise, and, (iic) either task **ping** or **getData** has finished. Our extended semantics check that such constraints hold at every state stopping the execution otherwise.

Let us outline the exploration performed by our deadlock-guided testing scheme on our working example (see Figure 3). At state 0 only task **simulate** is scheduled. Looking at its code we observe that tasks **register** and **work** are reachable, and from them also tasks **ping** and **getData**. At state 2 tasks **register** and **work** are scheduled, and from them we still see that tasks **ping** and **getData** may arise. At state 3, we have that **register** is blocked waiting for task **ping**, and **ping** has not yet finished. This information is obtained from the state's interleavings table (see S_3 in Figure 4). At state 4 we have that task **ping** has already finished. Also, we can infer that no more **register** nor **ping** tasks may arise from that state. We therefore have the situation (ii) above and the exploration can be safely stopped at this point. A similar situation happens at state 8, in this case with tasks **work** and **getData**. \square

In the following, we define deadlock-cycle constraints and the associated generation procedure. Let us notice that we represent incomplete information in these constraints by using variables in uppercase letters.

Definition 3 (Deadlock-Cycle Constraints): Given an extended state (S, IT) , a deadlock-cycle constraint takes one of the following two forms:

- 1) $\exists t_{L,T,PP} \mapsto \langle N, \rho \rangle$, that requires the existence (at some point) of a table entry of this form in IT (time constraint)
- 2) $\exists fut(F, L, Tk, p, \perp)$, that requires the existence (at some point) of a future variable of this form in S , whose bound task Tk cannot have finished (fut constraint)

Given an abstract deadlock cycle c , function $\phi(c)$ computes the set of deadlock-cycle constraints associated to c .

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

$$\left\{ \begin{array}{l} \{ \exists t_{L_i, Tk_i, _} \mapsto \langle _, p_i: F_i.\mathbf{get} \rangle, \exists fut(F_i, L_j, Tk_j, p_j, \perp) \} \cup \\ \phi(e_j \xrightarrow{p_j:tk_j} \dots, L_j, Tk_j) \quad \text{if } e_j = tk_j \\ \phi(e_j \xrightarrow{p_j:tk_j} \dots, L_i, Tk_j) \quad \text{if } e_j = \ell \end{array} \right.$$

Uppercase letters appearing for the first time in the constraints are fresh variables. This function has two different cases depending if e_j is an abstract task or an abstract location. The former handles location-task and task-task arrows, whereas the latter handles task-location arrows. Note the

following observations: (1) We do not use the abstract location and task identifiers of the deadlock cycle to generate the deadlock-cycle constraints, since they refer to concrete identifiers. Even if the same identifier is contained on two different nodes or arrows of the cycle, the corresponding variables in the constraints cannot be bound (i.e., the same variables cannot be used) since they could refer to different concrete identifiers. (2) The information contained by the cycle about the program points (p_i and p_j) is employed to generate both 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). We achieve that by using the 2nd and 3rd parameters of function ϕ . (4) In the second case, we use a fresh variable Tk_j since the location executing Tk_i can be blocked due to a (possibly) different task. Intuitively, these constraints characterize all possible deadlock traces representing the input cycle. By abuse of notation, we use $\phi(c)$ as $\phi(c, L, T)$, where c is an abstract deadlock-cycle and L and T are fresh variables.

Example 5: Let us see the deadlock-cycle constraints that function ϕ computes for cycle c in Example 4:

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

We show these constraints according to the order in which ϕ computes them. The first two constraints require the existence of a concrete time in IT in which some blocked database is awaiting while executing task **register** at line 25 for a certain worker to obtain the result of task **ping** at line 41. Notice that the worker has not executed **ping** because the future variable has the value \perp .

Moreover, the last two constraints require IT to contain a concrete time in which the this worker is blocked awaiting at line 38. It will not resume its execution until the reception of the data stored by some database at line 29. These data cannot be returned because the future variable has the value \perp . The constraints correspond to the intuition given above in the illustrative example of Section V-B2.a. It is important to highlight that, to preserve completeness, we do not bind the former and the latter databases. In a general example with several databases, there could be a deadlock in which a database waits for a worker which in turn waits for another database and worker, so that the last one waits to get the data stored by the first database. In case these two databases share the same variable name during the computation of function ϕ , we would not find the previous deadlock. That is, a deadlock trace can traverse several times an abstract deadlock cycle and still be one of its representatives.

Now, we use the deadlock-cycle constraints computed by ϕ for the given cycle to monitor and guide the systematic/symbolic execution, stopping derivations as soon as we explore a program state where one of these constraints is not satisfied. Boolean function $check_c$ checks the satisfiability of the constraints at a given state.

Definition 5: Given a set of deadlock-cycle constraints \mathcal{C} , and an extended state (S, IT) , function **check** holds, written $\text{check}_{\mathcal{C}}(S, IT)$, if:

$$\forall \{t_{L_i, Tk_i, p_i} \mapsto \langle N, p_i : F_i.\text{get} \rangle, \text{fut}(F_i, L_j, Tk_j, p_j, \perp)\} \in \mathcal{C}$$

one of the following conditions holds:

- 1) $\text{reachable}(t_{L_i, Tk_i, p_i}, S)$
- 2) $\exists t_{o_i, tk_i, p_i} \mapsto \langle n, p_i : f_i.\text{get} \rangle \in IT \wedge \text{fut}(f_i, o_j, tk_j, p_j, \perp) \in S$

Function **reachable** checks whether a given task might be spawned in subsequent states from state S . This is syntactically over-approximated by computing the transitive call relations from all tasks in the buffers of all locations in S . The use of more advanced analyses can improve the precision of the reachability of these tasks.

Intuitively, function **check** returns false if there is at least a time constraint so that: (i) its time identifier is not reachable, and, (ii) for every entry in IT matching the time constraint, there is a related **fut** constraint which is violated, i.e., its bound task has already finished in S (the return value is different from \perp). Condition (i) guarantees that there cannot be more representatives of the given cycle in future states. Consequently, if the execution can lead to deadlock, the interleavings table must already contain the time identifiers of the tasks involved in the abstract cycle. Condition (ii) guarantees that the current state cannot lead to deadlock since, at least one of the future variables which are being awaited by a blocked task is ready, and thus, such a task can resume its execution. Therefore, the current explored state cannot lead to a deadlock execution for the given cycle, and we can stop the derivation to avoid its exploration.

Given an extended state (S_0, IT_0) , $\text{exec}_c(S_0, IT_0)$ denotes the set of all complete derivations starting at (S_0, IT_0) using our enhanced semantics of Section V-A for cycle c . That includes finished derivations, deadlock derivations and derivations stopped by the **check** function. Formally, given the abstract deadlock-cycle c and a derivation $d \equiv (S_0, IT_0) \longrightarrow \dots \longrightarrow (S_n, IT_n)$, we say $d \in \text{exec}_c(S_0, IT_0)$ if (S_0, IT_0) is the extended initial state and one of the following holds: (1) $\nexists S_{n+1} \neq S_n$ such that $(S_n, IT_n) \longrightarrow (S_{n+1}, IT_{n+1})$ (S_n is a deadlock final state or the buffer of every location in S_n is empty), or (2) function $\text{check}_{\phi(c)}$ does not hold in (S_n, IT_n) (the execution cannot lead to a deadlock state since a deadlock-cycle constraint does not hold). Let us also define function $\text{deadlock}_c(S, IT)$ that checks if the extended state (S, IT) is a representative of the abstract deadlock cycle c .

Definition 6 (Deadlock-Cycle Guided-Testing (DCGT_c)): Consider an abstract deadlock cycle c and an extended initial state (S_0, IT_0) . We define DCGT_c , as the set $\{d : d \in \text{exec}_c(S_0, IT_0), \text{deadlock}_c(S_d, IT_d)\}$, where (S_d, IT_d) is the last state in derivation d . Let us also denote with $\text{DCGT}_c^{\text{search}}$ the exploration that is performed by DCGT_c until the first representative of the abstract deadlock cycle c is found.

Intuitively, DCGT_c computes all derivations leading to a representative of the abstract deadlock cycle c , possibly pruning (by means of the **check** function) derivations that do

not lead to it. Instead, $\text{DCGT}_c^{\text{search}}$ stops the exploration as soon as the first representative of c is found, hence its result can only be an unitary set or an empty set. Note that, DCGT_c , and also $\text{DCGT}_c^{\text{search}}$, can yield the empty set which means that the abstract cycle c obtained by the static analysis is a false positive.

Example 6: Let us consider the deadlock-cycle c of Example 4, and thus, the deadlock-cycle constraints $\phi(c)$ of Example 5. We apply DCGT_c to find all the deadlock traces that are representatives of cycle c . The interleavings table in the fourth state of the second derivation contains the entries $t_{0, \text{simulate}, 6} \mapsto \langle 0, \text{return} \rangle$, $t_{db, \text{register}, 22} \mapsto \langle 1, 25 : f_0.\text{get} \rangle$ and $t_{w, \text{ping}, 41} \mapsto \langle 2, \text{return} \rangle$. Constraints $\{\exists t_{L_1, Tk_1, _} \mapsto \langle _, 25 : F_1.\text{get} \rangle, \exists \text{fut}(F_1, L_2, Tk_2, 41, \perp)\}$ are not satisfiable (task **ping** has already finished at this point, as we can see in the interleavings table). **check**_c does not hold since $t_{L_1, Tk_1, 25}$ is not reachable anymore and thus, the derivation is pruned because there is no deadlock state reachable from this state. At state 8, the seventh derivation is similarly stopped. Also, the 4th and 5th derivations are deadlock executions which represent the abstract cycle c . These two executions are the deadlock traces obtained applying DCGT_c . Our methodology therefore explores 9 states instead of the 25 explored by the full symbolic/systematic execution. $\text{DCGT}_c^{\text{search}}$ stops the exploration of the first two executions at states 1 and 4, respectively. Finally, it detects state 5 as a representative of c and thus, the exploration is finished.

3) PROOF OF SOUNDNESS

The next theorem claims the soundness of the approach. Intuitively, if the symbolic/systematic execution explores a deadlock trace d which is a representative of an abstract cycle c , then DCGT_c will also explore d .

Theorem 1 (Soundness of DCGT_c): Given an extended initial state (S_0, IT_0) and an abstract cycle c , $\forall d \in \text{exec}(S_0, IT_0)$, $d \equiv (S_0, IT_0) \longrightarrow^* (S_n, IT_n)$,

$$\exists \gamma \in G_{S_n} \text{ such that } \alpha(\gamma) = c \implies d \in \text{exec}_c(S_0, IT_0)$$

Proof: By contradiction, let us suppose that $\exists d \in \text{exec}(S_0, IT_0)$ and $d \notin \text{exec}_c(S_0, IT_0)$. Hence, $\exists (S_i, IT_i) \in d$ such that $\text{check}_{\phi(c)}(S_i, IT_i)$ returns false and, consequently, the derivation $(S_0, IT_0) \longrightarrow^* (S_i, IT_i)$ stops. Therefore, at (S_i, IT_i) , there exists a set of deadlock-cycle constraints $\{t_{L_i, Tk_i, PP} \mapsto \langle N, p_i : F_i.\text{get} \rangle, \text{fut}(F_i, L_j, Tk_j, p_j, \perp)\} \subseteq \phi(c)$ that do not hold neither (1) nor (2) in Definition 5. However, this situation cannot occur, since function $\phi(c)$ requires necessary constraints for the existence of some representative of c and G_{S_n} contains a cycle that is a representative of c , then (1) or (2) must be fulfilled in every state of d and, in particular, in (S_i, IT_i) . As a result, we get a contradiction. \square

C. DEADLOCK-BASED TESTING CRITERIA

In the application of testing for deadlock detection, and in a general setting where there could occur different potential deadlock cycles, the following practical questions arise: do we need to obtain all deadlock traces? or are we rather

interested in just finding the first deadlock trace? For the purpose of the programmer to identify and fix the sources of the deadlock error(s), it could be instead more useful to find a deadlock trace per abstract deadlock cycle.

Let us define the following tree *deadlock-based testing adequacy criteria*:

- **all–deadlocks**, which requires the exploration of all executions leading to a final deadlock state;
- **first–deadlock**, which requires the exploration of a deadlock execution; and,
- **deadlock–per–cycle**, which, for each deadlock cycle reported by the analyzer, requires the exploration of a deadlock execution representing the given deadlock cycle (if it exists).

We have implemented the corresponding concrete testing schemes for each of the above criteria relying on our DCGT methodology.

1) DEADLOCK-GUIDED TESTING FOR THE all–deadlocks CRITERION

Let us define a deadlock-guided testing scheme for the all–deadlocks criterion, written DGT^{all} , as follows:

$$DGT^{all} = \bigcup_{c \in \Gamma} DCGT_c$$

where Γ is the set of abstract deadlock cycles returned by the deadlock analysis. Let us observe that in this case $DCGT_c$ is invoked for each abstract cycle c . It is important to notice that we can run each $DCGT_c$ in parallel, as these processes are independent with each other. One can also set a time-limit per $DCGT_c$ to avoid that the state explosion on a $DCGT_c$ for a certain cycle affects the performance of the whole process.

Example 7: Let us consider an initial context in which the location main contains the tasks 1:simulate(N_1) and 2:simulate(N_2). The deadlock analysis for such an initial context returns two abstract deadlock cycles, c_1 and c_2 , which are both analogous to cycle c in Example 4, but with their abstract nodes containing not only the program point where they were created but also the identifier of their corresponding creator task. This information is obtained thanks to the points-to analysis used as the basis to infer the abstractions of the deadlock analysis. DGT^{all} invokes $DCGT_{c_1}$ and $DCGT_{c_2}$, possibly in parallel, and explores a total of 14 deadlock traces instead of the 49 derivations that would be explored by a full systematic/symbolic execution.

2) DEADLOCK-GUIDED TESTING FOR THE first–deadlock CRITERION

Let us define a deadlock-guided testing scheme for the first–deadlock criterion, written DGT^{first} , algorithmically as follows:

```

function  $DGT^{first}$  =
  for each ( $c \in \Gamma$ )
    let  $d = DCGT_c^{search}$ ;
    if ( $d \neq \emptyset$ ) then break;
  return  $d$ ;

```

where Γ is again the set of abstract deadlock cycles returned by the deadlock analysis.

Example 8: Let us consider the same context as in Example 7. DGT^{first} would start invoking $DCGT_{c_1}^{search}$ resulting in a unitary set with a derivation analogous to the derivation ending at state 5 in Figure 3. The process then finishes returning such set.

3) DEADLOCK-GUIDED TESTING FOR THE deadlock–per–cycle CRITERION

Let us finally define a deadlock-guided testing scheme for the deadlock–per–cycle criterion, written DGT^{d-p-c} , as follows:

$$DGT^{d-p-c} = \bigcup_{c \in \Gamma} DCGT_c^{search}$$

where Γ is the set of abstract deadlock cycles returned by the deadlock analysis. Let us observe that in this case $DCGT_c^{search}$ is invoked for each abstract cycle c . Again, it is important to observe that we can run each $DCGT_c^{search}$ in parallel since they are completely independent. For practical reasons, we have set a time-limit per $DCGT_c^{search}$ to prevent that the state explosion on a $DCGT_c$ for a certain cycle c downgrades the performance of DGT^{d-p-c} .

Example 9: Let us consider again the same context as in Example 7. DGT^{d-p-c} invokes $DCGT_{c_1}^{search}$ and $DCGT_{c_2}^{search}$, possibly in parallel, obtaining for them two unitary sets, each with the corresponding derivation analogous to the derivation ending at state 5 in Figure 3. The union of the two sets is hence produced by DGT^{d-p-c} . Therefore DGT^{d-p-c} only needs to explore two deadlock traces, each of them covering one of the abstract deadlock cycles, instead of the 49 possible derivations that would be explored by a full systematic/symbolic execution.

VI. EXPERIMENTAL EVALUATION

All the techniques developed throughout the paper have been integrated within the tool SYCO/aPET [37], [38]. The tool is able to perform both dynamic and static testing for the ABS concurrent objects language [16]. It also incorporates techniques based on partial-order reduction [8], [9] to reduce the number of executions considered during the testing process by identifying and avoiding redundancies. The web site <http://costa.fdi.ucm.es/syco> contains a user-friendly interface to use SYCO online. Most of the benchmarks used in the experimental evaluation can also be found at the web site.

A. THE ABS LANGUAGE

In the ABS language, concurrent objects communicate with each other by means of *asynchronous* method calls. Each call spawns a new task to be executed by the receiving concurrent object. The synchronization between two tasks is performed by the use of future variables and two different kind of synchronization instructions. The blocking instruction **f.get** retains the processor of the executing concurrent object until the task related to the future variable f has finished. On the

other hand, the non-blocking instruction **await f?** releases the processor if the task related to **f** has not finished. Consequently, other tasks can be executed while the future variable **f** gets ready. In the proposed framework, handling the **await** instruction does not add any technical complication and our implementation includes support for it. However, for the sake of simplicity, we have not included this formalization.

B. GOALS OF THE EXPERIMENTAL EVALUATION

In this section, we show our experimental evaluation whose goals are the following:

- G1 Show that systematic/symbolic execution based testing complements static deadlock analysis. We consider these two sub-goals:
 - G1a In cases where there are deadlocks, show that the testing engine allows obtaining concrete deadlock traces, i.e., the locations that are involved, and, the exact sequence of task interleavings that cause the deadlock.
 - G1b Discard possible false positives obtained by the static analysis.
- G2 Study the effectiveness of the exploration reduction of our deadlock-guided methodology over the standard systematic/symbolic execution, and show its improvement in terms of scalability.
- G3 Evaluate the possible overhead that our technique may introduce. In particular, it is interesting to observe how our methodology performs in a worst-case scenario where we are not able to effectively guide the exploration towards deadlocks.
- G4 Show and compare the different testing schemes for the different deadlock-based criteria of Section V-C, and evaluate the potential impact of parallelizing the different DCGT processes.

C. DESCRIPTION OF THE EXPERIMENTS

We have used two sets of benchmarks. The first one contains classical concurrency programs with executions leading to deadlock. Benchmark *DBW* is a communication protocol between several workers requesting access to the data stored by a database; benchmark *F* is a distributed implementation of the factorial of a natural number; benchmark *PP* is a solution for the pairing problem; benchmark *UF* contains a loop that creates several concurrent objects and spawns asynchronous tasks on them; benchmark *HB* is a solution for the classical hungry birds problem; benchmark *SB* is a generalization of the sleeping barber problem with several barbers and clients. Finally, benchmark *TF* (borrowed from [1]) is a work protocol between several workers in a factory, which have been slightly adapted so that it produces deadlock executions. The second set of benchmarks contains deadlock-free versions of some of the previous benchmarks. We use *fP* to denote the *P* program for which the deadlock analyzer is not able to prove deadlock freedom and returns a set of abstract deadlock cycles. Finally, each of these benchmarks contains a class *Main* and a method *main* with several input parameters.

Table 1 shows the results of the experimental evaluation. It compares the performance of our deadlock guided testing (DGT) methodology both for the **deadlock-per-cycle** and **all-deadlocks** criteria (columns **d-p-c** and **all**, respectively), against the performance of the standard symbolic execution. In order to observe how the gain of DGT evolves with bigger explorations, we execute each benchmark with two different limits on the number of loop iterations (column **k**). These limits are chosen differently for each benchmark according to its complexity. We measure the number of complete executions and the total time taken for both the systematic/symbolic execution and the DGT with the **all-deadlocks** criterion (columns **Ans** and **T**). On the other hand, for the DGT with the **deadlock-per-cycle** criterion we measure the “number of deadlock executions”/“number of unfeasible cycles”/“number of abstract cycles inferred by the deadlock analysis” (column **D/U/C**). Let us notice that the DCGTs for each cycle could be performed in parallel. Consequently, we also show the maximum time among the different DCGTs (column **T_{max}**) to show the best performance we could get if the support for such parallelism would be implemented (goal G4). We have imposed a timeout of 180 seconds and show ∞ when the timeout is reached.

For instance, for the *HB* benchmark with limit $k = 4$, we have that the standard systematic/symbolic execution is not able to scale and reaches the timeout. However, our DGT is able to find 1145 deadlock executions in 2847 ms. Also, the columns for the **deadlock-per-cycle** criterion shows us that the deadlock analysis finds five different abstract deadlock cycles, but three of them were false positives. For the other two, we are able to find at least one feasible deadlock execution. The total time needed is 6912 ms. If we assume an ideal parallel setting with 5 processors, one per deadlock cycle, the total time could potentially be 2337 ms. Note that this time also includes the time of the deadlock analyzer.

We show the gains in time (columns in the group **Speedup**) of deadlock-guided testing both for **deadlock-per-cycle** (columns **T_{gain}** and **T_{gain}^{max}**) and **all-deadlocks** (column **T_{gain}^{all}**) over the standard systematic/symbolic execution. In the case of the **deadlock-per-cycle** criterion, the gains are provided both (1) assuming a sequential setting, hence we consider value *T* of DGT (column **T_{gain}**), and (2) an ideal parallel setting, therefore considering **T_{max}** (column **T_{gain}^{max}**). We compute these gains as X/Y , where *X* is the measure of the systematic/symbolic execution and *Y* is that of the corresponding DGT.

The experimental results are obtained on an Intel(R) Core(TM) i7 CPU at 2.3GHz with 8GB of RAM, running Mac OS X 10.8.5. Times are in milliseconds. We have imposed a timeout of 180 seconds. If the timeout is reached by DGT, we use $>X$ to denote that we have obtained *X* units for the considered measure right in the timeout. In the case of the speedups, $>X$ indicates that the speedup would be *X* if DGT finishes in the timeout, and hence the real speedup is guaranteed to be greater than *X*.

TABLE 1. Experimental evaluation.

Benchmark	k	Symb. Exec.		DGT (d-p-c)			DGT (all)			Speedup		
		Ans	T	D/U/C	T	T _{max}	Ans	T	T _{gain}	T _{gain} ^{max}	T _{gain} ^{all}	
DBW	2	1k	1k	1/0/1	22	3	50	99	50.8	374.9	10.8	
	3	196k	∞	1/0/1	21	3	3k	6k	>9k	>60k	>32.3	
F	3	11k	7k	3/0/3	17	7	1	30	426.1	1k	241.5	
	4	269k	∞	3/0/3	33	5	1	72	>5k	>36k	>3k	
PP	2	16	15	2/0/2	12	3	2	9	1.3	5.0	1.7	
	3	310	224	2/0/2	10	3	8	16	22.5	74.7	14.0	
HB	3	16k	10k	2/3/5	325	325	1k	120	31.4	31.4	80.8	
	4	206k	∞	2/3/5	7k	7k	2k	3k	>27	>27	>64	
UF	2	1	∞	1/0/1	31	4	36	100	>58k	>450k	>18k	
	3	167k	∞	1/0/1	66	4	72	261	>3k	>45k	>690	
SB	1	29	31	1/0/1	20	4	3	20	1.6	7.8	1.66	
	2	217k	∞	1/0/1	19	4	70	177	>94k	>450k	>10k	
TF	4	19k	15k	1/0/1	747	747	4	12k	20.4	20.4	1.2	
	5	228k	∞	1/0/1	8	8	16	∞	>23k	>23k	-	
fUF	2	201k	∞	0/1/1	430	430	0	427	>4k	>4k	>4k	
	3	147k	∞	0/1/1	17k	17k	0	18k	>11	>11	>10	
fF	3	5k	4k	0/1/1	34	34	0	34	131.0	131.0	131.0	
	4	207k	∞	0/1/1	90	90	0	111	>2k	>2k	>1k	
fPP	3	7k	4373	0/2/2	29	29	0	30	150.8	150.8	145.8	
	6	341k	∞	0/2/2	3k	3k	0	3k	>535	>534.3	>535.1	

D. ANALYSIS OF THE RESULTS

To the light of the experimental results, we can confirm that deadlock analysis and systematic/symbolic execution based testing can complement each other (goal G1). For the first set of benchmarks, concrete test cases (including input data, full execution traces and scheduling decisions) for feasible deadlock cycles are obtained. This information can help to understand the cause of the deadlock and fix it (goal G1a). For the second set of benchmarks, for which the analyzer is imprecise and produce false positives, all potential deadlocks are discarded and therefore we are able to prove deadlock freedom modulo the termination limit that we have to impose to guarantee the termination of the symbolic execution (goal G1b).

As regards goal G2, our results also show that the proposed technique is effective and that we can obtain a notable reduction of the search space over the standard systematic/symbolic execution. The gains of DGT are huge (in many cases, more than three orders of magnitude) both in time and in number of explored states. It is also important to notice that these gains are much larger in the first set of benchmarks (namely, in DBW(3), F(4), F(10), PP(9), UF(3)). The reason is that, in general, the generated constraints for unfeasible cycles are usually less effective in guiding the execution towards the deadlock (e.g. in HB(4)). Indeed, considering HB(5), we are not able to prove that one of the abstract cycles is a false positive but neither we can find a representative execution for it. Even in these cases, systematic/symbolic execution is outperformed by deadlock-guided testing.

On the other hand, we can observe that the reductions of DGT over the systematic/symbolic execution are less notable in the second set of benchmarks. The two main reasons are that (1) neither of the deadlock cycles is feasible, and, (2) each DCGT cannot stop until all potential deadlock executions have been considered. The most notable evidence that DGT improves in terms of scalability over the standard systematic/symbolic execution (second part of goal G2), is the fact that DGT is able to successfully finish the exploration for all our experiments in reasonable time whereas in many cases the standard systematic/symbolic execution blows-up (see the ∞ 's in the fourth column).

Benchmark TF exposes a situation that can be considered close to the worst-case scenario for our methodology. Namely, there is no information that allows us to detect non-deadlock executions until the very end, and, therefore, we are not able to effectively guide the exploration towards deadlocks. This benchmark hence allows us to study goal G3. In particular, we can observe that even without being able to effectively guide the exploration, we still get a slight gain in $T_{\text{gain}}^{\text{all}}$ (1.2 for TF(4)) which indicates that the overhead introduced by our technique remains quite low (less than the time which is saved by the few prunings that are produced in this case).

Regarding goal G4, as expected, the potential impact of parallelizing the different DCGT processes is notable. This can be seen by comparing the values of the T and T_{max} columns of the **DGT (d-p-c)** group and also the corresponding gains (columns T_{gain} and $T_{\text{gain}}^{\text{max}}$), which are clearly much larger when considering a parallel setting.

All in all, our experimental results show that our proposed technique complements deadlock analysis. It discards unfeasible abstract deadlock cycles, and we explore deadlock executions with a notable reduction on the number of explored states and time. Thus, DGT is very effective for programs containing deadlocks, but it can also be used to prove deadlock freedom for many cases in which the analyzer produce false positives.

VII. RELATED WORK

Deadlock detection is a deep research area. Since our methodology is a combination of both static and dynamic techniques, and the individual methods can be employed for various goals, our related work covers a wide range of existing approaches that we organize as follows.

A. DEADLOCK ANALYSIS

Even though there are numerous dynamic approaches, most of the techniques for deadlock detection in the literature are based on static analysis both for thread-based programs [39], [40] and for asynchronous programs [1], [41]. These analyses are able to ensure deadlock freedom, however the loss of precision (due to approximations) might lead to a “don’t know” answer. This is especially usual in the context of pointer aliasing.

The technique proposed in this work can help the static analysis techniques to prove deadlock freedom in cases where the analysis is unable to do it by itself. Deadlock-guided testing uses the static information reported by a deadlock analysis to guide the exploration towards deadlock states. We have used the information reported by [1] to guide the exploration, but we could also use the output of other static analyzers (e.g., [41]). We would only need to change the function ϕ to make use of the new reported information to generate our deadlock-cycle constraints.

In [3], a deadlock analysis is proposed for programs with threads and lock-reentrancy, that is, programs where a thread may acquire several times the same lock. This is achieved thanks to the use of program abstractions called *lams*, which are basic recursive models collecting the dependencies and feature recursion inside the program. The program is deadlock free if there is not a circularity in the *lam* models of the program. We could use the information within the *lams* associated to circularities to guide the execution. Similarly, in [3] a new behavioral type system is proposed to prove deadlock freedom in Java-like programs using shared objects and coordination primitives like *wait*, *notify* and *notifyAll*. These behavioral types are called *usages* and may be encoded as a class of Petri net. This work reduces deadlock freedom of a program to check a reachability problem in these nets. One of the drawbacks of this approach is that the number of acquisitions for each shared object must be known, which is not always available.

Finally, [2] presents a novel notion of dependency an deadlock for condition synchronization, (**await** statement for boolean conditions), and a sound deadlock analysis that extends [1]. It integrates a theorem prover into the deadlock

analysis to handle unbounded data types and recursion without performance loss. However, this analysis still produces false positives. Our approach could discard these ones using the information reported by this new analysis.

B. SYMBOLIC EXECUTION, VERIFICATION, MODEL CHECKING, TESTING

The symbolic execution engine of our framework is the one presented in [13]. We could directly use this component to perform (non-guided) deadlock detection. This engine can also be used to detect other types of errors (e.g., critical states that can produce a malfunction of the program). Model checking, among other verification techniques, uses symbolic execution to automatically verify correctness properties. Indeed, deadlock detection has been deeply studied in this context [42]. Both static and dynamic testing aim at finding bugs, among them deadlocks (see, e.g., [6], [43]–[47]). Indeed, static testing systems use symbolic execution as the core of their engines, and in particular, as it is the case for our aPET [48] tool. The specification of more general properties has been widely studied in the area of *property-based testing* [49], [50] and, thus, we believe that it can be helpful to automatically generate constraints leading the execution towards other specific properties of interest.

In the context of the packet switching networks and Networks on Chip, the work in [51] proposes a new classification of different types of deadlock and uses a symbolic model-checker, called nuXmv, to check deadlock freedom for each of such type of deadlock. In this context, a deadlock happens when a message is stuck in a channel, and it will never be processed or received by the target of the channel. Specifically, three kinds of deadlocks are defined: global, local and weak deadlocks. It is important to highlight that this deadlock classification could also be used in our framework by considering the different dependencies produced by the use of non-blocking synchronization.

In the context of concurrent programs, a main target of ongoing research on testing is to avoid the exploration of redundant derivations. Two derivations are considered redundant or equivalent if we can swap the execution of independent processes to obtain one from the other according to the Partial Order Reduction theory (POR) [11]. Dynamic POR [52], [53] is a successful approach to avoid the exploration of such redundancies. We can use the POR theory within our framework to guide even more the execution towards deadlock derivations. Thus, both lines of research are orthogonal.

C. HYBRID APPROACHES

Hybrid approaches usually employ information reported by static analysis to improve the performance of testing for deadlock detection, namely [54] and [55]. As regards [54], it applies dynamic testing over a trace program that only contains the relevant instructions for deadlock detection after applying a transformation over the original program. Because of this transformation, it could detect deadlocks that appear in the trace program but cannot occur in the original program.

This work differs from ours in that, we test the original program and we always detect real deadlock traces that can be later reproduced by the programmers. Besides, our approach considers any unknown parameters thanks to the use of symbolic execution.

As regards [55], it detects potential abstract deadlock cycles and accelerates the process thanks to the information reported by a type system. The main similarity with our work is that both use static information reported by an analysis to improve the performance and guide the exploration of the testing tool. However, it differs from our approach in two important aspects: first, our technique works on a general characterization of deadlock of asynchronous programs. Their work captures deadlocks in thread-based programs caused by the use of locks, but not other deadlocks caused by more general mechanisms like wait-notify instructions. Second, they use a type inference algorithm to infer deadlock types that must be understood by the checking algorithm to take advantage of them. Our work is based on a deadlock analysis which report descriptions of deadlock cycles and our semantics is extended to interpret them and guide the exploration.

D. GENERATION OF INITIAL CONTEXTS

Static testing generally assumes no knowledge on the inputs of the system under test. In the context of distributed and concurrent programs, a set of locations and their connections is usually provided, denoted as an initial configuration or initial context. E.g. we could consider a *main* location with a *simulate* task on our motivating example as an initial context. The framework presented in this work is able to symbolically execute a system without assuming an initial context [56]. Consequently, our method is able to automatically generate initial contexts in order to detect deadlocks and, systematically, discard those contexts in which a deadlock cannot occur. Following with our motivating example, our framework could generate a context with a database location that must execute a *register* task and a worker location, whose buffer contains a *work* task. However, also many other contexts that can never lead to deadlock. This introduces a new combinatorial explosion on the number of initial contexts to be considered, most of which will be deadlock-free. Interestingly, we can use the information reported by the deadlock analysis to detect the conflicting task interactions that can lead to deadlock, only generating initial contexts that contain such tasks.

VIII. CONCLUSION AND FUTURE WORK

Deadlocks are one of the most usual problems in concurrent programs. Static analyses can prove the absence of deadlocks but due to the use of approximations can usually produce false positives. Moreover, they are usually not able to provide precise information that can help the user in finding the source of the problem. On the other hand, systematic/symbolic testing can be used for detecting the presence of deadlocks providing precise information about the source of the problem, namely,

producing a concrete execution trace where the deadlock occurs. However, the state space, even without redundancies (which could be detected relying on POR techniques) can be intractable.

This paper proposes a hybrid approach that combines static deadlock analysis and systematic/symbolic testing for effective deadlock detection. Specifically, our methodology consists in using the abstract information about potential deadlocks obtained by the deadlock analysis in order to steer the systematic/symbolic execution towards paths that can lead to deadlock, trying to discard as soon as possible deadlock-free paths.

The experimental evaluation carried out supports our claim that this methodology improves significantly in terms of scalability over the standard, non-guided, systematic/symbolic execution. Also, in cases where static deadlock analysis is imprecise and produce false positives, our methodology is able to discard all potential deadlock traces, and therefore, it is able to prove deadlock freedom (modulo the termination/coverage criterion of the symbolic execution). Our technique hence improves over static deadlock analysis in accuracy but at the same time inherits the main inadequacy of systematic/symbolic execution, namely its scalability issues due to the state explosion problem, although, as our experiments demonstrate, scalability gets considerably alleviated thanks to our guided search.

Unlike the original scheme in [12], which can only be applied in dynamic settings with full knowledge of the program inputs and initial states (usually the testing process must start from a *main* method that creates a concrete initial state), our new methodology based on symbolic execution is able to find deadlock traces or to ensure deadlock freeness in static contexts where the input data is possibly unknown, hence notably boosting its potential applicability. It is also important to highlight that our technique can benefit from the most advanced POR techniques in the literature.

As lines for future research, besides the integration of such most advanced POR techniques, we plan to study guiding the exploration towards other properties of interest. For instance, for symbolic test case generation, it could be interesting to guide the symbolic execution towards concurrency-based test adequacy criteria.

REFERENCES

- [1] A. Flores-Montoya, E. Albert, and S. Genaim, "May-happen-in-parallel based deadlock analysis for concurrent objects," in *Formal Techniques for Distributed Systems (Lecture Notes in Computer Science)*, vol. 7892. Florence, Italy: Springer, Jun. 2013, pp. 273–288.
- [2] E. Kamburjan, "Detecting deadlocks in formal system models with condition synchronization," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 14, p. 76, May 2018, doi: [10.14279/tuj.eceasst.76.1070](https://doi.org/10.14279/tuj.eceasst.76.1070).
- [3] C. Laneve, "A lightweight deadlock analysis for programs with threads and reentrant locks," *Sci. Comput. Program.*, vol. 181, pp. 64–81, Jul. 2019, doi: [10.1016/j.scico.2019.06.002](https://doi.org/10.1016/j.scico.2019.06.002).
- [4] C. Laneve and L. Padovani, "Deadlock analysis of wait-notify coordination," in *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy (Lecture Notes in Computer Science)*, vol. 11760, M. S. Alvim, K. Chatzikokolakis, C. Olarte, and F. Valencia, Eds. Cham, Switzerland: Springer, 2019, pp. 50–67, doi: [10.1007/978-3-030-31175-9_4](https://doi.org/10.1007/978-3-030-31175-9_4).

- [5] M. Christakis, "On narrowing the gap between verification and systematic testing," *Inf. Technol.*, vol. 59, no. 4, p. 197, Jan. 2017, doi: [10.1515/itit-2017-0001](https://doi.org/10.1515/itit-2017-0001).
- [6] M. Christakis, A. Gotovos, and K. Sagonas, "Systematic testing for detecting concurrency errors in erlang programs," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation*, Luxembourg, U.K., Mar. 2013, pp. 154–163.
- [7] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," in *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, vol. 3922. Vienna, Austria: Springer, Mar. 2006, pp. 339–356.
- [8] E. Albert, M. G. de la Banda, M. Gómez-Zamalloa, M. Isabel, and P. J. Stuckey, "Optimal context-sensitive dynamic partial order reduction with observers," in *Proc. 28th ACM Int. Symp. Softw. Test. Anal.*, Jul. 2019, pp. 352–362.
- [9] E. Albert, M. Gómez-Zamalloa, M. Isabel, and A. Rubio, "Constrained dynamic partial order reduction," in *Computer Aided Verification (Lecture Notes in Computer Science)*, vol. 10982, Oxford, U.K.: Springer, Jul. 2018, pp. 392–410, doi: [10.1007/978-3-319-96142-2_24](https://doi.org/10.1007/978-3-319-96142-2_24).
- [10] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *Proc. 41st Symp. Princ. Program. Lang.*, Jan. 2014, pp. 373–384.
- [11] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, J. Palsberg and M. Abadi, Eds., Long Beach, CA, USA, Jan. 2005, pp. 110–121.
- [12] E. Albert, M. Gómez-Zamalloa, and M. Isabel, "Combining static analysis and testing for deadlock detection," in *Integrated Formal Methods (Lecture Notes in Computer Science)*, vol. 9681, E. Ábrahám and M. Huisman, Eds. Reykjavik, Iceland: Springer, Jun. 2016, pp. 409–424, doi: [10.1007/978-3-319-33693-0_26](https://doi.org/10.1007/978-3-319-33693-0_26).
- [13] E. Albert, P. Arenas, and M. Gómez-Zamalloa, "Test case generation of actor systems," in *Proc. 13th Int. Symp. Automat. Technol. Verification Anal.* (Lecture Notes in Computer Science), vol. 9364. Shanghai, China: Springer, Oct. 2015, pp. 259–275, doi: [10.1007/978-3-319-24953-7_21](https://doi.org/10.1007/978-3-319-24953-7_21).
- [14] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, doi: [10.1145/2408776.2408795](https://doi.org/10.1145/2408776.2408795).
- [15] M. Isabel, "Deadlock-guided testing in CLP," M.S. thesis, Dept. Sistemas Inf. Comput., Máster en Ingeniería Informática, Facultad de Inf., Universidad Complutense de Madrid, Madrid, Spain, 2017. [Online]. Available: <https://eprints.ucm.es/43974/>
- [16] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, "ABS: A core language for abstract behavioral specification," in *Formal Methods for Components and Objects (Lecture Notes in Computer Science)*, vol. 6957, B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds. Graz, Austria: Springer, Nov./Dec. 2012, pp. 142–164.
- [17] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 261–322, Sep. 1989.
- [18] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.
- [19] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Raleigh, NC, USA: Pragmatic Bookshelf, 2007.
- [20] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, nos. 2–3, pp. 202–220, 2009.
- [21] D. Wyatt, *Akka Concurrency*. Walnut Creek, CA, USA: Artima, 2013.
- [22] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Burlington, MA, USA: Morgan Kaufmann, 2012.
- [23] H. C. Baker and C. Hewitt, "The incremental garbage collection of processes," in *Proc. Symp. Artif. Intell. Program. Lang.*, New York, NY, USA, 1977, pp. 55–59.
- [24] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," in *Proc. Conf. Program. Lang. Design Implement. (PLDI)*, R. L. Wexelblat, Ed., New York, NY, USA, 1988, pp. 260–267.
- [25] R. H. Halstead, "MULTILISP: A language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, Oct. 1985.
- [26] Y. Yokote and M. Tokoro, "Concurrent programming in Concurrent-Smalltalk," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. Cambridge, MA, USA: MIT Press, 1987, pp. 129–158.
- [27] A. Yonezawa, J.-P. Briot, and E. Shibayama, "Object-oriented concurrent programming ABCL/1," in *Proc. Conf. Proc. Object-Oriented Program. Syst., Lang. Appl.* New York, NY, USA: ACM, 1986, pp. 258–268.
- [28] D. Caromel and Y. Roudier, "Reactive programming in Eiffel//," in *Proc. Conf. Object-Based Parallel Distrib. Comput. (OBPDC)* (Lecture Notes in Computer Science), vol. 1107, J. Briot, J. Geib, and A. Yonezawa, Eds. Tokyo, Japan, Jun. 1996, pp. 125–147.
- [29] G. Cugola and C. Ghezzi, "CJava: Introducing concurrent objects in Java," in *Proc. 4th Int. Conf. Object Oriented Inf. Syst.*, M. E. Orlowska and R. Zicari, Eds., 1997, pp. 504–514.
- [30] D. Caromel and L. Henrio, *A Theory Distribution Object*. Berlin, Germany: Springer, 2005.
- [31] F. S. de Boer, D. Clarke, and E. B. Johnsen, "A complete guide to the future," in *Programming Languages and Systems (Lecture Notes in Computer Science)*, vol. 4421, R. de Nicola, Ed. Braga, Portugal: Springer, Mar./Apr. 2007, pp. 316–330.
- [32] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.
- [33] A. Gotlieb, B. Botella, and M. Rueher, "A CLP framework for computing structural test data," in *Proc. CL*, vol. 1861. London, U.K.: Springer, Jul. 2000, pp. 399–413.
- [34] C. Meudec, "ATGen: Automatic test data generation using constraint logic programming and symbolic execution," *Softw. Test., Verification Rel.*, vol. 11, no. 2, pp. 81–96, 2001.
- [35] R. A. Müller, C. Lembeck, and H. Kuchen, "A symbolic java virtual machine for test case generation," in *Proc. IASTEDSE*, 2004, pp. 365–371.
- [36] N. Tillmann and J. de Halleux, "Pex—white box test generation for .net," in *Tests and Proofs (Lecture Notes in Computer Science)*, vol. 4966. Prato, Italy: Springer, Apr. 2008, pp. 134–153.
- [37] E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y. Wong, "APET: A test case generation tool for concurrent objects," in *Proc. ESEC/FSE*, 2013, pp. 595–598. [Online]. Available: <http://costa.fdi.ucm.es/apet>
- [38] E. Albert, M. Gómez-Zamalloa, and M. Isabel, "SYCO: A systematic testing tool for concurrent objects," in *Proc. 25th Int. Conf. Compiler Construct.*, Mar. 2016, pp. 269–270.
- [39] S. P. Masticola and B. G. Ryder, "A model of ada programs for static deadlock detection in polynomial time," in *Proc. Parallel Distrib. Debugging*, 1991, pp. 97–107.
- [40] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.
- [41] E. Giachino, C. Grazia, C. Laneve, M. Lienhardt, and P. Wong. (2013). *Deadlock Analysis of Concurrent Objects—Theory and Practice*. [Online]. Available: <https://www.cs.unibo.it/laneve/papers/daco.pdf>
- [42] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Computer Aided Verification (Lecture Notes in Computer Science)*, vol. 3576, K. Etessami and S. K. Rajamani, Eds. Scotland, U.K.: Springer, Jul. 2005, pp. 82–97, doi: [10.1007/11513988_9](https://doi.org/10.1007/11513988_9).
- [43] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proc. ACM SIGPLAN Conf. Program. Lang. design Implement.*, Dublin, Ireland, 2009, pp. 110–120, doi: [10.1145/1542476.1542489](https://doi.org/10.1145/1542476.1542489).
- [44] B. K. Ali Kheradmand and G. Candea. (2013). *Lockout: Efficient Testing for Deadlock Bugs*. [Online]. Available: <http://dslab.epfl.ch/pubs/lockout.pdf>
- [45] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *Proc. 7th Int. SPIN Workshop*, Stanford, CA, USA, Aug./Sep. 2000, pp. 245–264, doi: [10.1007/10722468_15](https://doi.org/10.1007/10722468_15).
- [46] J. Hamin and B. Jacobs, "Deadlock-free monitors," in *Programming Languages and Systems*, A. Ahmed, Ed. Cham, Switzerland: Springer, 2018, pp. 415–441.
- [47] J. Park, B. Choi, and S. Jang, "Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments," *Int. J. Parallel Program.*, vol. 48, no. 6, pp. 1032–1060, Dec. 2020, doi: [10.1007/s10766-020-00661-3](https://doi.org/10.1007/s10766-020-00661-3).
- [48] E. Albert, P. Arenas, and M. Gómez-Zamalloa, "Symbolic execution of concurrent objects in CLP," in *Proc. PADL*, vol. 7149. Philadelphia, PA, USA: Springer, Jan. 2012, pp. 123–137.
- [49] C. V. Espinosa, E. Martin-Martin, A. Riesco, and J. Rodriguez-Hortala, "FlinkCheck: Property-based testing for apache flink," *IEEE Access*, vol. 7, pp. 150369–150382, 2019, doi: [10.1109/ACCESS.2019.2947361](https://doi.org/10.1109/ACCESS.2019.2947361).

- [50] A. Loscher and K. Sagonas, "Automating targeted property-based testing," in *Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST)*, Västerås, Sweden, Apr. 2018, pp. 70–80, doi: [10.1109/ICST.2018.00017](https://doi.org/10.1109/ICST.2018.00017).
- [51] A. Stramaglia, J. J. A. Keiren, and H. Zantema, "Deadlock in packet switching networks," Eindhoven Univ. Technol., Eindhoven, The Netherlands, Tech. Rep. CoRR abs/2101.06015, 2021.
- [52] 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," in *Formal Techniques for Distributed Systems (Lecture Notes in Computer Science)*, vol. 7273, H. Giese and G. Rosu, Eds. Stockholm, Sweden: Springer, 2012, pp. 219–234.
- [53] E. Albert, P. Arenas, and M. Gómez-Zamalloa, "Actor- and task-selection strategies for pruning redundant state-exploration in testing," in *Formal Techniques for Distributed Objects, Components, and Systems (Lecture Notes in Computer Science)*, vol. 8461. Berlin, Germany: Springer, Jun. 2014, pp. 49–65.
- [54] P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," in *Proc. 18th Int. Symp. Found. Softw. Eng.*, Santa Fe, NM, USA, Nov. 2010, pp. 327–336, doi: [10.1145/1882291.1882339](https://doi.org/10.1145/1882291.1882339).
- [55] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Proc. 1st Int. Haifa Verification Conf.*, Haifa, Israel, Nov. 2005, pp. 191–207, doi: [10.1007/11678779_14](https://doi.org/10.1007/11678779_14).
- [56] E. Albert, M. Gómez-Zamalloa, and M. Isabel, "Generation of initial contexts for effective deadlock detection," in *Logic-Based Program Synthesis and Transformation (Lecture Notes in Computer Science)*, vol. 10855. Namur, Belgium: Springer, Oct. 2018, pp. 3–19.



MIGUEL GÓMEZ-ZAMALLOA received the Ph.D. degree in computer science from the Complutense University of Madrid (UCM), in 2009. Since then, he has promoted over different positions with UCM, where he currently holds a permanent position as a Senior Lecturer. He is the author of more than 50 publications in international journals and conferences. His work has received more than 940 citations according to Google Scholar. His main research interest includes the development of techniques and tools for automated analysis, testing, and verification.



MIGUEL ISABEL received the Ph.D. degree in computer science from the Complutense University of Madrid, in 2020. From 2014 to 2016, he was an Assistant Researcher with UCM. He is currently working as an Assistant Teacher of Computer Networks with the Technical University of Madrid. His research interests include testing of concurrent programs, deadlock analysis, partial order reduction, and software verification.

...