Generation of Initial Contexts for Effective Deadlock Detection[⋆]

Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel

elvira@fdi.ucm.es, mzamalloa@fdi.ucm.es, miguelis@ucm.es

Complutense University of Madrid (UCM), Spain

Abstract. It has been recently proposed that testing based on symbolic execution can be used in conjunction with static deadlock analysis to define a deadlock detection framework that: (i) can show deadlock presence, in that case a concrete test-case and trace are obtained, and (ii) can also prove deadlock freedom. Such symbolic execution starts from an *initial distributed context*, i.e., a set of locations and their initial tasks. Considering all possibilities results in a combinatorial explosion on the different distributed contexts that must be considered. This paper proposes a technique to effectively generate initial contexts that can lead to deadlock, using the possible conflicting task interactions identified by static analysis, discarding other distributed contexts that cannot lead to deadlock. The proposed technique has been integrated in the above-mentioned deadlock detection framework hence enabling it to analyze systems without the need of any user supplied initial context.

1 Motivation

Deadlocks are one of the most common programming errors and they are therefore one of the main targets of verification and testing tools. We consider a distributed programming model with explicit *locations* (or distributed nodes) and *asynchronous* tasks that may be spawned and awaited among locations. Each location represents a processor with a procedure stack and an unordered queue of pending tasks. Initially all processors are idle. When an idle processor's task queue is non-empty, some task is selected for execution, this selection is non-deterministic. Let us see now our motivating example in Figure 1 which simulates a simple communication protocol between a database location and a worker location. Our implementation has the main method, and two classes Worker and DB implementing the worker and the database, respectively. The main method creates two distributed locations: the database and the worker, and (asynchronously) invokes methods register and work on each of them, respectively. The work method of a worker simply accesses the database (invoking asynchronously method getData) and then *blocks* until it gets the result, which is

[⋆] This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, by the Spanish MECD FPU Grant FPU15/04313 and by the CM project S2013/ICE-3006.

```
1  main(){
2      DB db = new DB();
3      Worker w = new Worker();
4      db!register(w);
5      w!work(db);}
6
7  class Worker{
8    Data data;
9    int work(DB db){
10     Future⟨Data⟩ f;
11     f = db!getData(this);
12     data = f.get;
13     return 0;
14   }
15   int ping(int n){return n;}
16 }// end of class Worker
17
18 class DB{
19   Data data = ...;
20   Worker client = null;
21   int connected = 1;
22   int connect(){
23       connected = 3;
24       return connected;
25   }
26   int register(Worker w){
27       connected = 5;
28       Future⟨Data⟩ g;
29       g = this!getData(w);
30       await g?;
31       if (connected > 0){
32          connected = connected − 1;
33          Future⟨int⟩ f = w!ping(5);
34          if (f.get == 5) client = w;
35       }
36       return 0;
37   }
38   Data getData(Worker w){
39     if (client == w) return data;
40     else return null;
41   }
42 }// end of class DB
```

Fig. 1. Working example. Communication protocol between a DB and a worker

assigned to its data field. The instruction get blocks the execution in the current location until the awaited task has terminated. We use future variables [7,8] to detect the termination of asynchronous tasks. The register method of the database makes a call to getData and waits for its execution. Once it has finished, it checks if the number of possible connections is bigger than 0. In that case connected is decreased by one, and the database makes sure that the worker is online. This is done by invoking asynchronously method ping with a concrete value and blocking until it gets the result with the same value. Then, the database registers the provided worker reference storing it in its client field. Method getData of the database returns its data field if the caller worker is registered, otherwise it returns **null**. Finally, method connect sets the field connected to 3. Depending on the sequence of interleavings, the execution of this program can finish: (1) as one would expect, i.e., with worker.data = db.data, (2) with w.data = **null** if getData is executed before the assignment at line 34, or, (3) in a deadlock.

We have recently proposed a deadlock detection framework [3,2] that combines static analysis and symbolic execution based testing [1,3,6,14]. The deadlock analysis (for example, [9]) is first used to obtain descriptions of potential deadlock cycles which are then used to guide the testing process. The resulting deadlock detection framework hence can: (i) show deadlock presence, in which case a concrete test-case and trace are obtained, and (ii) prove deadlock freedom (up to the symbolic execution exploration limit). However, the symbolic execution phase needs to start from a concrete initial distributed context, i.e., a set of locations and their initial tasks. In our example, such an initial context is provided by the main method, which creates a Database and a Worker location, and schedules a work task on the worker with the database as parameter, and, a register task on the database with the worker as parameter. This is however only one out of the possible contexts, and, of course, it could be the case that it does not expose an error that occurs in other contexts (for example, it does not manifest any deadlock). This clearly limits the framework potential.

A fundamental challenge for a symbolic execution framework of distributed programs is to automatically and systematically generate *relevant* distributed contexts for the type of error that it aims at detecting. This would allow for instance applying symbolic execution for system and integration testing. The generation of relevant contexts involves two challenging aspects: (1) A first challenge is related to the elimination of redundant (useless) contexts. Observe that there is a combinatorial explosion on the different possible distributed contexts that can be generated when one considers all possible types and number of distributed locations and tasks within them. Therefore, it is crucial to provide the *minimal* set of initial contexts that contains only one representative of equivalent contexts. (2) For the particular type of error that one aims at detecting, an additional challenge is to be able to only generate initial contexts in which the error can occur. In the case of generating initial contexts for deadlock detection in our working example, this would mean generating for instance, a context with a database location and some worker location with a scheduled work task and a register task on the database for it, i.e., the context created by the main method. For instance, contexts that do not include both tasks would be useless for deadlock detection. Let us observe that if the assignment at Line 23 is changed to assign 0, then the initial contexts must also include a connect task, otherwise no deadlock will be produced. Interestingly, deadlock analyses provide [9,11,12] potential *deadlock cycles* which contain the possibly conflicting task interactions that can lead to deadlock. This information will be used to help our framework anticipate this information and discard initial distributed contexts that cannot lead to deadlock from the beginning. Briefly, the main contributions of this paper are the following:

- We introduce the concept of *minimal* set of initial contexts and extend a static testing framework to automatically and systematically generate them.
- We present a deadlock-guided approach to effectively generate initial contexts for deadlock detection and prove its soundness.
- We have implemented our proposal within the aPET/SYCO system [4] and performed an experimental evaluation to show its efficiency and effectiveness.

2 Asynchronous Programs

A program consists of a set of classes that define the types of locations, each of them defines a set of fields and methods of the form $M{::=}T\ m(\bar{T}\ \bar{x})\{s\}$, where statements s take the form s::=s; s | x=e | **if** e then s **else** s | **while** e **do** s |

return x; | b=**new** $\mathsf{T}(\bar{\mathsf{z}})$ | f = x ! $\mathsf{m}(\bar{\mathsf{z}})$ | **await** f? | x = f.**get**. Syntactically, a location will therefore be similar to a *concurrent object* that can be dynamically created using the instruction **new** $\mathsf{T}(\bar{\mathsf{z}})$. The declaration of a future variable is as follows Future⟨T⟩ f, where T is the type of the result r, it adds a new future variable to the state. Instruction f = x ! $\mathsf{m}(\bar{\mathsf{z}})$ spawns a new task (instance of method m) and it is set to the future f in the state. Instruction **await** f? allows non-blocking synchronization. If the future variable f we are awaiting for points to a finished task, then the **await** can be completed. Otherwise the task yields the lock so that any other task of the same location can take it. On the other hand, instruction f.**get** allows blocking synchronization. It waits for the future variable without yielding the lock, i.e., it blocks the execution of the location until the task that is awaiting is finished. Then, when the future is ready, it retrieves the result and allows continuing the execution. This instruction introduces possible deadlocks in the program, as two tasks can be awaiting for termination of tasks on each other's locations. Finally, instruction **return** x; releases the lock that will never be taken again by that task. Consequently, that task is *finished* and removed from the task queue. All statements of a task takes place serially (without interleaving with any other task) until it gets to a **return** or **await** f? instruction. Then, the processor becomes idle again, chooses non-deterministically the next pending task, and so on.

A *program state* or *configuration* is a set of locations $\{loc_0, ..., loc_n\}$. A *location* is a term $loc(o, tk, h, \mathcal{Q})$ where o is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\perp$ if the location's lock is free, h is its local heap, and $\mathcal{Q}$ is the set of tasks in the location. A *task* is a term $tsk(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed. We assume that the execution starts from a *main* method without parameters. The initial state is $S=\{loc(0, 0, \perp, \{tsk(0, main, l, body(main))\}$ with an initial location with identifier 0 executing task 0, maps local variables to their initial values, and $body(m)$ is the sequence of instructions in method m and $ini(main)$ is the initial program point in method m. From now on, we represent the state as a Prolog list, and we write $[x \mapsto v]$ to denote $h(x) = v$ (resp. $l(x) = v$), that is, field x in the heap h (resp. local variable x in the mapping l) takes the value v.

In what follows, a *derivation* or *execution* [20] is a sequence of states $S_0 \xrightarrow{o_1 \cdot t_1} ... \xrightarrow{o_n \cdot t_n} S_n$, where $S_i \xrightarrow{o_i \cdot t_i} S_{i+1}$ denotes the execution of task t_i in location $o_i \in S_i$. The derivation is *complete* if S_0 is the initial state and $\nexists\, loc(o, \_, \_, \{tk\} \cup \mathcal{Q}) \in S_n$ such that $S_n \xrightarrow{o.tk} S_{n+1}$ and $S_n \neq S_{n+1}$. Given a state S, $exec(S)$ denotes the set of all possible complete executions starting at S.

3 Specifying and Generating Initial Contexts

In our asynchronous programs, the most *general* initial contexts consist of sets of locations with *free* variables in their fields, and initial tasks in each location queue with *free* variables as parameters, i.e., neither the fields nor the parameters have concrete values. A first approach to systematically generate initial contexts could consist in generating, on backtracking, all possible multisets of initial tasks (method names), and for each one, generate all aliasing combinations with the locations of the tasks belonging to the same type of location. They are multisets because there can be multiple occurrences of the same task. To guarantee termination of this process we need to impose some limit in the generation of the multisets. For this, we could simply set a limit on the multiset global size. However it would be more reasonable and useful to set a limit on the maximum cardinality of each element in the multiset. To allow further flexibility, let us also set a limit on the minimum cardinality of each element. For instance, if we have a program with just one location type A with just one method m, and we set 1 and 2 as the minimum and maximum cardinalities respectively, then there are two possible multisets, namely, $\{m\}$ and $\{m, m\}$. The first one leads to one initial context with one location of type A with an instance of task m in its queue. The second one leads to two contexts, one with one location of type A with two instances of task m in its queue, and the other one with two different locations, each with an instance of task m in its queue.

On the other hand, it makes sense to allow specifying which tasks should be considered as initial tasks and which should not. A typical scenario is that the user knows which are the main tasks of the application and does not want to consider auxiliary or internal tasks as initial tasks. Another scenario is in the context of integration testing, where the tester might want to try out together different groups of tasks to observe how they interfere with each other. Also, the use of static analysis can help determine a subset of tasks of interest to detect some specific property. This is the case of our deadlock-guided approach of Section 4. With all this, the input to our automatic generation of initial contexts is: a set of tuples (C.M,C^{min}, C^{max}), where C.M is an *abstract task*, i.e., a task name, being C and M the class and method name resp., and, C^{min} resp. C^{max} is the associated minimum resp. maximum cardinality. Note that this does not limit the approach in any way since one could just include in $\mathcal{T}_{ini}$ all methods in the program and set $C^{min} = 0$ and a sufficiently large C^{max}.

Example 1. Let us consider the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.connect}, 0, 1)\}$. The corresponding multisets are $\{\mathsf{register}\}$ and $\{\mathsf{register}, \mathsf{connect}\}$. All contexts must contain exactly one instance of task register and at most one instance of task connect. This leads to three possible contexts: (1) a DB location instance with a task register in its queue, (2) a DB location instance with tasks register and connect in its queue, and, (3) two different DB location instances, one of them with an instance of task register and the other one with an instance of task connect. For instance, the state corresponding to the latter context would be:

$$\begin{aligned}
\mathtt{S} = [&\mathtt{loc}(\mathtt{DB1}, \mathtt{bot},\ [\mathtt{data} \mapsto \mathtt{D1}, \mathtt{clients} \mapsto \mathtt{Cl1}, \mathtt{checkOn} \mapsto \mathtt{B1}],\\
&\qquad\qquad [\mathtt{tsk}(1, \mathtt{register}, [\mathtt{this} \mapsto \mathtt{r}(\mathtt{DB1}), \mathtt{m} \mapsto \mathtt{W1}], \mathtt{body}(\mathtt{register}))])\\
&\mathtt{loc}(\mathtt{DB2}, \mathtt{bot},\ [\mathtt{data} \mapsto \mathtt{D2}, \mathtt{clients} \mapsto \mathtt{Cl2}, \mathtt{checkOn} \mapsto \mathtt{B2}],\\
&\qquad\qquad [\mathtt{tsk}(2, \mathtt{connect}, [\mathtt{this} \mapsto \mathtt{r}(\mathtt{DB2})], \mathtt{body}(\mathtt{connect}))])],
\end{aligned}$$

where $\mathsf{D1}, \mathsf{Cl1}$, and $\mathsf{B1}$ (resp. $\mathsf{D2}, \mathsf{Cl2}$, and $\mathsf{B2}$) are the fields data, clients, and checkOn of location DB1 (resp. DB2), and W1 resp. W2 the parameter of the task register resp. connect, and body(m) is the sequence of instructions in method m. Note that both fields and task parameters are fresh variables so that the context is the most general possible. Note that the first parameter of a task is always the location this and it is therefore fixed. □

In the following, we formally define the contexts that must be produced from a set of abstract tasks $\mathcal{T}_{ini}$ with associated cardinalities.We use the notation $\{[m_1, ..., m_n]_{o_i}\}$ for an initial context where there exists a location $loc(o_i, \bot, h,$ $\{tk(tk_1, m_1, l_1, body(m_1))\} \cup ... \cup \{tk(tk_n, m_n, l_n, body(m_n))\})$. Note that we can have $m_i = m_j$ with $i \neq j$. For instance, the three contexts in Example 1 are written as $\{[\mathsf{register}]_{\mathsf{db_1}}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db_1}}\}$ and $\{[\mathsf{register}]_{\mathsf{db_1}}, [\mathsf{connect}]_{\mathsf{db_2}}\}$, respectively. Let us first define the set of initial contexts from a given $\mathcal{T}_{ini}$ when all tasks belong to the same class.

Definition 1 (Superset of initial contexts (same class C_i)). *Let $\mathcal{T}_{ini} = \{(C_i.m_1, C_1^{min}, C_1^{max}), \ldots, (C_i.m_n, C_n^{min}, C_n^{max})\}$ be the set of abstract tasks with associated cardinalities. Let us have $\sum_{i=1}^{n} C_i^{max}$ different identifiers: $o_{1,1}, \ldots, o_{1,C_1^{max}}$, $\ldots, o_{n,1}, \ldots, o_{n,C_n^{max}}$. We can find at most $\sum_{i=1}^{n} C_i^{max}$ instances of class C_i, that is, each abstract task m_i ($i \in [1, n]$) has at most C_i^{max} instances and each of them can be inside a different instance of class C_i. Let $u_{i,j}^{m_k}$ be an integer variable that denotes the number of instances of task m_k inside the location $o_{i,j}$ and let us consider the following integer system:*

$$\begin{cases} C_1^{min} \leq u_{1,1}^{m_1} + \ldots + u_{1,C_1^{max}}^{m_1} + \ldots + u_{n,1}^{m_1} + \ldots + u_{n,C_n^{max}}^{m_1} \leq C_1^{max} \\ \ldots \\ C_n^{min} \leq u_{1,1}^{m_n} + \ldots + u_{1,C_1^{max}}^{m_n} + \ldots + u_{n,1}^{m_n} + \ldots + u_{n,C_n^{max}}^{m_n} \leq C_n^{max} \end{cases}$$

Each formula requires at least C_k^{min} and at most C_k^{max} instances of task m_k. Each solution to this system corresponds to an initial context.
Let $(d_{1,1}^{m_1}, \ldots, d_{n,C_n^{max}}^{m_1}, \ldots, d_{1,1}^{m_n}, \ldots, d_{n,C_n^{max}}^{m_n})$ be a solution, then the corresponding initial context contains:

- *$loc(o_{i,j}, \bot, h, \mathcal{Q})$, that is, a location $o_{i,j}$ whose lock is free, the fields in h are mapped to fresh variables, and the queue $\mathcal{Q}$ contains: $d_{i,j}^{m_1}$ instances of abstract task m_1,..., and $d_{i,j}^{m_n}$ instances of m_n, if $i \in [1, n]$, $j \in [1, C_i^{max}]$ and $\exists d_{i,j}^{m_k} > 0, k \in [1, n]$, where each instance of m_i is $tsk(tk, m_i, l, body(m_i))$ and every argument in l is mapped to a fresh variable.*

Example 2. Let us consider the example $\mathcal{T}_{ini}$=$\{(\mathsf{DB.register}, 0, 1), (\mathsf{DB.connect}, 1, 1)\}$. The identifiers are $o_{1,1}$ and $o_{2,1}$, and the variables of the system are $u_{1,1}^{reg}$, $u_{2,1}^{reg}$, $u_{1,1}^{get}$ and $u_{2,1}^{get}$. Finally, we obtain the next system:

$$\begin{cases} 0 \leq u_{1,1}^{reg} + u_{2,1}^{reg} \leq 1 \\ 1 \leq u_{1,1}^{get} + u_{2,1}^{get} \leq 1 \end{cases}$$

We obtain 6 solutions: $(0,0,1,0), (0,0,0,1), (1,0,1,0), (1,0,0,1), (0,1,1,0)$ and $(0,1,0,1)$. Then, the superset of initial contexts is

$$\{\{[\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{connect}]_{\mathsf{o}_{2,1}}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{o}_{2,1}}\},$$

$$\{[\mathsf{register}]_{\mathsf{o}_{2,1}}, [\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}]_{\mathsf{o}_{1,1}}, [\mathsf{connect}]_{\mathsf{o}_{2,1}}\}\}$$

□

Let us observe that the two last contexts are equivalent since they are both composed of two instances of DB with tasks register and connect respectively. Therefore, we only need to consider one of these two contexts for symbolic execution. Considering both would lead to *redundancy*. The notion of minimal set of initial contexts below eliminates redundant contexts, hence avoiding useless executions.

Definition 2 (Equivalence relation $\sim$). *Two contexts C_1 and C_2 are equivalent, written $C_1 \sim C_2$, if $C_1 = C_2 = \emptyset$ or $C_1 = \{loc(o_1, \bot, h_1, \mathcal{Q}_1)\} \cup C_1'$, and $\exists\ o_2 \in C_2$ such that:*

1. *$C_2 = \{loc(o_2, \bot, h_2, \mathcal{Q}_2)\} \cup C_2'$,*
2. *$\mathcal{Q}_1$ and $\mathcal{Q}_2$ contain the same number of instances of each task, and*
3. *$C_1' \sim C_2'$.*

Example 3. The superset in Example 2 contains 3 equivalence classes induced by the relation $\sim$: (1) the class $\{\{[\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{connect}]_{\mathsf{o}_{2,1}}\}\}$, where both contexts are composed of a location with a task connect, (2) the class $\{\{[\mathsf{register}, \mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{o}_{2,1}}\}\}$, whose locations have two tasks register and connect. and, finally, (3) the class $\{\{[\mathsf{register}]_{\mathsf{o}_{2,1}}, [\mathsf{connect}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}]_{\mathsf{o}_{1,1}}, [\mathsf{connect}]_{\mathsf{o}_{2,1}}\}\}$, where both contexts have two locations with a task register and a task connect, respectively. □

Definition 3 (Minimal set of initial contexts $\mathcal{I}^{C_i}$ (same class Cl_i)). *Let $\mathcal{T}_{ini}$ be the set of abstract tasks, then the* minimal set of initial contexts $\mathcal{I}^{Cl_i}$ *is composed of a representative of each equivalence class induced by the relation $\sim$ over the superset of initial contexts for the input $\mathcal{T}_{ini}$.*

Example 4. As we have seen in the previous example, there are three different equivalence classes. So, the minimal set of initial contexts is composed of a representative of each class (we have renamed the identifiers for the sake of clarity):

$$\mathcal{I}^{DB} = \{\{[\mathsf{connect}]_{\mathsf{db}_1}\}, \{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db}_1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{connect}]_{\mathsf{db}_2}\}\}$$

□

Let us now define the set of initial contexts $\mathcal{I}$ when the input set $\mathcal{T}_{ini}$ contains tasks of different types of locations.

Definition 4 (Minimal set of initial contexts $\mathcal{I}$ (Different classes)). *Let $\mathcal{T}_{ini} = \{(C_1.m_1, C_1^{min}, C_1^{max}), \ldots, (C_n.m_n, C_n^{min}, C_n^{max})\}$ be the set of abstract tasks with associated cardinalities, and let us consider a partition of this set where every equivalence class is composed of abstract tasks of the same class. Hence, we have: $\mathcal{T}_{ini}^{C_1} = \{C_1.m'_1, \ldots, C_1.m'_{j_1}\}, \ldots, \mathcal{T}_{ini}^{C_n} = \{C_n.m''_1, \ldots, C_n.m''_{j_n}\}$ where $C_i \neq C_j, \forall i, j \in [1, n], i \neq j$.*
Then, let $\mathcal{I}^{C_i}$ be the minimal set of initial contexts for the input $\mathcal{T}_{ini}^{C_i}$, $i \in [1, n]$ and $U : \mathcal{I}^{C_1} \times \ldots \times \mathcal{I}^{C_n} \to \mathcal{I}$, defined by $U(s_1, \ldots, s_n) = s_1 \cup \ldots \cup s_n$. The set $\mathcal{I}$ is defined by the image set of application U.

Example 5. Let us consider the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.connect}, 1, 1), (\mathsf{Worker.work}, 1, 1)\}$ from which we get the initial contexts $\mathcal{I}^{Worker} = \{\{[\mathsf{work}]_{\mathsf{w_1}}\}\}$ and $\mathcal{I}^{DB} = \{\{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db,1}}\}, \{[\mathsf{register}]_{\mathsf{db_1}}, [\mathsf{connect}]_{\mathsf{db_2}}\}\}$. Then, by Def. 4,

$$\mathcal{I} = \{\{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db_1}}, [\mathsf{work}]_{\mathsf{w_1}}\}, \{[\mathsf{register}]_{\mathsf{db_1}}, [\mathsf{connect}]_{\mathsf{db_2}}, [\mathsf{work}]_{\mathsf{w_1}}\}\}$$

□

It is straightforward to implement a function that generates the minimal set of initial contexts from a provided set of initial tasks (for instance [5]). Such a function is denoted as *generate_contexts($\mathcal{T}_{ini}$)*. The main complication is to avoid the generation of equivalent contexts (Definition 2) as soon as possible during the process. For this aim one can rely on the definition of a normal form according to the number of tasks inside each location.

4 On Automatically Inferring Deadlock-Interfering Tasks

The systematic generation of initial contexts produces a combinatorial explosion and therefore it should be used with small sets of abstract tasks (and low cardinalities). However, in the context of deadlock detection, in order not to miss any deadlock situation, one has to consider in principle all methods in the program, hence producing scalability problems. Interestingly, it can happen that many of the tasks in the generated initial contexts do not affect in any way deadlock executions. Our challenge is to only generate initial contexts from which a deadlock can show up. For this, the deadlock analysis provides the possibly conflicting task interactions that can lead to deadlock. We propose to use this information to help our framework discard initial contexts that cannot lead to deadlock from the beginning. Section 4.1 summarizes the concepts of the deadlock analysis used to obtain the deadlock cycles, and Section 4.2 presents the algorithm to generate the set of initial tasks $\mathcal{T}_{ini}$.

4.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [9] returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points,

$tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions. The abstractions for tasks and locations can be performed at different levels of accuracy during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location o by the program point at which it is created o_{pp}, and each task by the method name executing (as in Section 3). They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Points-to analysis [18,9] can be used to infer such abstractions with more precision, for instance, by distinguishing the actions performed by different location abstractions. Each arrow $e \xrightarrow{p:tk} e'$ should be interpreted like "abstract location or task e is waiting for the termination of abstract location or task e' due to the synchronization instruction at program point p of abstract task tk". Three kinds of arrows can be distinguished, namely, *task-task* (an abstract task is awaiting for the termination of another one), *task-location* (an abstract task is awaiting for an abstract location to be idle) and *location-task* (the abstract location is blocked due the abstract task). *Location-location* arrows cannot happen.

Example 6. In our working example there are two abstract locations, o_2, corresponding to location database created at line 2 and o_3, corresponding to the n locations worker, created inside the loop at line 3; and four abstract tasks, *register*, *getD*, *work* and *ping*. The following cycle is inferred by the deadlock analysis: $o_2 \xrightarrow{\text{34:register}} \text{ping} \xrightarrow{\text{15:ping}} o_3 \xrightarrow{\text{12:work}} \text{getD} \xrightarrow{\text{38:getD}} o_2$. The first arrow captures that the location created at Line 2 is blocked waiting for the termination of task ping because of the synchronization at L34 of task register. Also, a dependency between a task and a location (for instance, ping and o_3) captures that the task is trying to execute on that (possibly) blocked location. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can be observed, it is complex to figure out from them why these dependencies arise, and more importantly the interleavings scheduled to lead to this situation. □

4.2 Generation of initial tasks

The underlying idea is as follows: we select an abstract cycle detected by the deadlock analysis, and extract a set of potential abstract tasks which can be involved in a deadlock. In a naive approximation, we could take those abstract tasks that are inside the cycle and contain a blocking instruction. We also need to set the maximum cardinality for each task to ensure finiteness (by default 1) and require at least one instance for each task (minimum cardinality).

This approach is valid as long as we only have blocking synchronization primitives, i.e., when the location state stays unchanged until the resumption of a suspended execution. However, this kind of concurrent/distributed languages usually include some sort of non-blocking synchronization primitive. When a location stops its execution due to an **await** instruction, another task can interleave its execution with it, i.e., start to execute and, thus, modify the location state (i.e., the location *fields*). Then, if a call or a blocking instruction involved in a deadlock depends on the value of one of these fields, and we do not consider all the possible values, a deadlock could be missed. As a consequence, we need to consider at release points, all possible interleavings with tasks that modify the fields in order to capture all deadlocks.

Let us consider now a simple modification of our working example. Line 27 is replaced by connected = 0. Now it is easy to see that if we only consider register and work as input, deadlocks are lost: once register is executed and the instruction at line 30 is reached, the location's queue only contains task getData but no connect and, therefore, when task register is resumed, field connected stays unchanged and the body of the condition is not executed, so we cannot have a deadlock situation.

In the following we define the *deadlock-interfering* tasks for a given abstract deadlock cycle, i.e., an *over-approximation* of the set of tasks that need to be considered in initial contexts so that we cannot miss a representative of the given deadlock cycle. In our extended example, those would be, register and work but also connect.

Definition 5 (initialTasks(C)). *Let C an abstract deadlock cycle. Then,*

$$initialTasks(C) := \bigcup_{i_{call}\in t\in C} initialTasks(t, i_{call}, C) \cup \bigcup_{i_{sync}\in t\in C} initialTasks(t, i_{sync}, C)$$

where:

- $initialTasks(t,i,C) = \emptyset \quad if\ o \xrightarrow{t} t_2 \notin C\ and\ i \neq i_{mod}\ and\ \nexists\ i_{await} \in [t_0, i]$
- $initialTasks(t,i,C) = \{t\}\ if\ (o \xrightarrow{t} t_2 \in C\ or\ i = i_{mod}\ )\ and\ \nexists\ i_{await} \in [t_0, i]$
- $initialTasks(t,i,C) = \{t\} \cup \bigcup_{f\in fields(i)} \left(\bigcup_{(i_{mod},t_{mod})\in mods(f)} initialTasks(t_{mod}, i_{mod}, C) \right)$
  $if\ \exists\ i_{await}\ \in [t_0, i]$

The definition relies on function fields(I) which, given an instruction I, returns the set of class fields that have been read or written until the execution of instruction I. Let mods(f) be the set of pairs (instruction,task) that modify field f. We can observe that *initialTasks(C)* is the union of the initial tasks for each relevant instruction inside the cycle C, i.e., asynchronous calls and synchronization primitives. We can also observe in the auxiliary function *initialTasks(t,i,C)* that: (1) if the instruction i is not producing a *location-task edge* and it is not an instruction modifying a field, then t does not need to be added as initial task, (2) if i produces a *location-task edge* or is modifying a field, and we do not have any **await** instruction between the beginning of the task and i, then i is going to be executed under the most general context, so we do not need to add more initial tasks but t, and (3) on the other hand, if there exists an **await** instruction between the beginning of task t, namely t_0, and instruction i, each field f inside the set fields(i) could be changed before the resumption of the **await** by any task modifying f. Thus, tasks containing any of the possible f-modifying instructions must be considered and, recursively, their initial tasks.

It is important to highlight that this definition could be non-terminating depending on the program we are working with. For instance, if we apply the

```
Data: An abstract cycle C and a maximum cardinality M
Result: A list with the interfering tasks for C
$Q = \emptyset$; $L = \emptyset$;
forall $t \in C$ do
    $i_{call}$ = receiveCall(t,C); enqueue(Q,($i_{call}$,t));
    $i_{await}$ = receiveSync(t,C); enqueue(Q,($i_{await}$,t));
    $i_{get}$ = receiveSync(t,C); enqueue(Q,($i_{get}$,t));
    if $\exists \in o \xrightarrow{t} t_2 \in C$ then
        insert(L,($i_{get}$,t));
    end
end
while !empty(Q) do
    (i,t) = dequeue(Q);
    if $\exists i_{await} \in t$ between the beginning of $t$ and $i$ then
        forall $f \in fields(i)$ do
            forall $(i_{mod}, t_{mod}) \in mods(f)$ do
                if !member(L,($i_{mod}, t_{mod}$)) then
                    insert(L,($i_{mod}, t_{mod}$));
                    enqueue(Q,($i_{mod}, t_{mod}$));
                end
            end
        end
    end
end
return [(m,1,M) : m $\in$ set(project$_y$(L))];
```
Algorithm 1: Algorithm to infer interfering tasks for a given deadlock cycle

definition to the abstract cycle C in Example 6, $initialTasks(\mathsf{db.register}, 32, C)$ will be evaluated. It fits well with the conditions on the third clause, as there exists an **await** instruction, fields(32) = {connected} and then again 32 is a modifier instruction of field connected, so $initialTasks(\mathsf{db.register}, 32, C)$ will be evaluated again recursively.

Algorithm 1 shows how to finitely infer the interfering-tasks for a given deadlock cycle as defined by Def 5. Function $receiveCall(t, C)$ ($receiveSync(t, C)$) receives the asynchronous call (synchronization instruction) of a task t inside the cycle C. Q is the queue of pending pairs {instruction, task}, and L is the list containing all such pairs whose tasks we have to consider. Finiteness is guaranteed because each instruction is added to Q and L at most once, and the number of instructions is finite. For each task in the cycle, we take the call and the corresponding synchronization instruction, and we add them to Q. Instructions **get** producing a *location-task edge*, are also added to L, as they have to be inside the initial context. The other tasks included in the initial context are the ones which could affect the conditions of the aforementioned instructions.

In the second loop, we take a pending instruction inside Q and we check if there exists an **await** instruction where the field values could be changed (third clause in definition 5). In case it does, we need to include all tasks which contain instructions modifying such field. However, this change could be inside an if-else body and we also need to consider the fields inside such condition. Therefore, we add the modifier instruction to the pending instructions queue Q. The algorithm finishes when Q is empty and L is the list of pairs with all interfering instructions and their container tasks. Finally, we only take the tasks, i.e., the second component of each pair (project$_y$), remove duplicates (set) and set their minimum and maximum cardinalities. From now on, we denote *initial_tasks(c,M)*, the set of initial tasks inferred for the abstract deadlock cycle c and the maximum cardinality M.

Example 7. Let us show how the algorithm works for our modified example and the maximum cardinality $M = 1$. For the sake of clarity, instructions are identified by their line numbers. After executing the first **forall** loop, the value of Q and L is $\{(33, \mathsf{DB.register}), (34, \mathsf{DB.register}), (11, \mathsf{Worker.work}), (12, \mathsf{Worker.work})\}$ and $[(34, \mathsf{DB.register}), (12, \mathsf{Worker.work})]$, respectively. Let us assume Q uses a LIFO policy, hence $(12, \mathsf{Worker.work})$ is taken first. Since $\mathsf{fields(12)} = \emptyset$, L stays unchanged. The same happens with $(11, \mathsf{Worker.work})$. At the beginning of the third loop, Q is $\{(33, \mathsf{DB.register}), (34, \mathsf{DB.register})\}$ and $(34, \mathsf{DB.register})$ is taken. Now, $\mathsf{fields(34)} = \{\mathsf{connected}\}$ and $\exists inst_{await}$ (line 30) between lines 26 and 34. We find three pairs modifying the field connected: (23,DB.connect), (27,DB.register) and (32,DB.register). None of them is a member of L and hence they are added to both queues. Now, Q is $\{(33, \mathsf{DB.register}), (27, \mathsf{DB.register}), (32, \mathsf{DB.register}), (23, \mathsf{DB.connect})\}$ but again $\mathsf{fields(32)} = \mathsf{fields(23)} = \emptyset$ and, thus, L stays unchanged. Finally, both $(33, \mathsf{DB.register})$ and $(27, \mathsf{DB.register})$ are taken and $\mathsf{fields(33)}$= $\mathsf{fields(27)}$=$\{\mathsf{connected}\}$, but the modifier instructions have been previously added to L, hence L remains unchanged. At the end of **while**, L is $\{(34, \mathsf{DB.register}), (12, \mathsf{Worker.work}), (27, \mathsf{DB.register}), (32, \mathsf{DB.register}), (23, \mathsf{DB.connect})\}$. Finally, the algorithm projects over the second component of each pair in the list, removes duplicates and returns the set $\mathcal{T}_{ini}$=$\{(\mathsf{DB.register}, 1, 1), (\mathsf{Worker.work}, 1, 1), (\mathsf{DB.connect}, 1, 1)\}$. Our generation of initial contexts for this set (see Example 5) produces

$$\begin{aligned}\mathcal{I} = \{\ &\{[\mathsf{register}, \mathsf{connect}]_{\mathsf{db_1}}[\mathsf{work}]_{\mathsf{w_1}}\},\\ &\{[\mathsf{register}]_{\mathsf{db_1}}, [\mathsf{connect}]_{\mathsf{db_2}}, [\mathsf{work}]_{\mathsf{w_1}}\}\},\end{aligned}$$

where both initial contexts are composed of a worker location with a task work. However, the former context contains a database location with tasks register and connect, whereas the latter one contains two locations with a task register and a task connect, respectively. □

The next theorem establishes the soundness of our approach. Intuitively, soundness states that, for a given deadlock cycle c and maximum cardinality M, if there is an initial context, fulfilling M, from which a deadlock representative of c can be obtained, then our approach will generate a context (possibly different from the above) from which a deadlock representative of c is obtained.

Theorem 1 (Soundness). *Given a program P, an abstract deadlock cycle c and a maximum cardinality M, if there exists a derivation starting at a state*

S_{ini} and ending at S_{end} such that the cardinality of each task in S_{ini} is less than M and S_{end} is a representative of the cycle c, then there exists an initial context $St_0 \in generate\_contexts(initial\_tasks(c, M))$ such that $S_{end_2} \in exec(St_0)$ and S_{end_2} is also a representative of the cycle c.

Proof. (Sketch) Let us define a task t as necessary in S_{ini} for the deadlock cycle c if and only if $\nexists S_{e'}$ such that $S_{ini} \backslash \{t\} \xrightarrow{*} S_{e'}$ and $S_{e'}$ is a representative of c, where $S \backslash \{t\}$ denotes the context S without the task t. Let us define now an initial context $nec(S)$ as the initial context that only contains the necessary tasks in S for c. In order to prove soundness, we need to prove that $nec(S_{ini}) \in generate\_contexts(initial\_tasks(M, c))$. We reason by contradiction. Assume that there exists a necessary task $t \in nec(S_{ini})$, instance of method m, which is not in any initial context generated. This is equivalent to assume that method m is not inferred by Algorithm 1. We can distinguish two different roles which task t plays in the deadlock situation:

- If task t gets blocked, then t contains an instruction pp:**get** where pp is the program point, and, by the soundness of the deadlock analysis (Theorem 1 of [9]), pp:**get** is the tag of an edge inside the deadlock cycle c. So, the pair (pp, m) is added to L in the first loop of Algorithm 1 and m is finally inferred. Thus, we have a contradiction.
- If task t modifies a field f at program point pp that appears in a condition of another task r, then we cannot get a deadlock if t is not executed before the evaluation of condition in task r (t is necessary). Here, we need to notice that if task r does not contain any **await**, symbolic execution explores all possible execution paths and t would be unnecessary. But we have supposed that t is necessary, then r contains an **await**. Then, (pp, m) will be added to L because of the third forall in Algorithm 1 and m is inferred, what contradicts our assumption. □

5 Experimental Evaluation

We have implemented the proposed techniques within the aPET/SYCO tool [4], a testing tool for the ABS [13] *concurrent objects* language. The tool is available for online use at `http://costa.ls.fi.upm.es/syco`, where the benchmarks below can also be found. This section summarizes our experimental evaluation whose objectives are the following:

1. Show the effectiveness of our approach in Section 4 to generate initial contexts for deadlock detection w.r.t the full systematic generation of Section 3.
2. Demonstrate the potential of the technique when being applied in practice within our deadlock detection framework.

The benchmarks we have used include classical concurrency patterns containing deadlocks, namely: *DBProt* is an extension of the database communication protocol of our working example; *Barber* is an extension of the *sleeping barber*

		M = 1				**M = 2**				**M = 3**			
Bench.	$\mathbf{T_A/C}$	**Syst.**	**G**	**D**	**T**	**Syst.**	**G**	**D**	**T**	**Syst.**	**G**	**D**	**T**
DBProt	5/1	30	2	1	35	>12960	57	30	101s*	>6308	576	156	974s*
Barber	5/1	8	1	1	35	6859	9	9	57	>8310	36	36	309
Fact	6/2	15	2	2	11	2419	6	6	14	>4771	12	12	16
Loop	20/1	3375	1	1	30	>13433	27	27	495	>4771	216	216	77s*
Pairing	4/2	2	2	2	9	57	12	12	37	576	42	42	162

Table 1. Evaluating generation of initial contexts: Systematic vs. deadlock-guided

problem, *Fact* is a distributed and recursive implementation of a factorial function, *Loop* is a loop that creates asynchronous tasks and locations, and, *Pairing* is the pairing problem.

Effectiveness of generation of initial contexts for deadlock detection: Table 1 shows, for each benchmark: the number of generated initial contexts using the full systematic generation of contexts of Section 3 (column *Syst.*), the number of contexts generated using our deadlock-guided generation of Section 4 (column G), and, the number of contexts among those generated that lead to a deadlock (column D). This is done for three different values of maximum cardinality, namely, $M = 1$, $M = 2$ and $M = 3$. The rest of the columns are explained in the next paragraph. A timeout of 30s is used and, when reached, we write $>X$ to indicate that we encountered X contexts up to that point. The reductions of our deadlock guided generation of contexts w.r.t the full systematic generation are huge. As expected the full systematic generation blows up fast for most examples. We can also observe that our deadlock guided generation of contexts is very precise, producing no false positives, i.e., contexts that do not lead to deadlock, except for *DBProt.* The reason of the loss of precision in the *DPProt* example is that task register only gets blocked if task connect changes the value of field connected. Therefore, contexts in which these two tasks do not belong to the same location will not lead to deadlock. This can be observed in Example 7. Improving our method to capture this situation is left for future work.

Application within our deadlock detection framework: Our deadlock-guided generation of initial contexts has been integrated within the deadlock detection feature of the testing system aPET/SYCO as follows: After running the static deadlock analysis, and only in case it outputs a non-empty set of potential abstract cycles (i.e. if the program is not already proven deadlock-free), we run our deadlock guided generation of initial contexts for each of the cycles inferred by the analysis. For each generated initial context, we start (possibly in parallel) a deadlock-guided symbolic execution [3,2] that stops as soon as it finds a deadlock. As a result, we obtain a concrete test-case with its associated trace and sequence of interleavings. A local timeout for each symbolic execution is set so that it does not degrade the overall process in case a blowup is produced before finding a deadlock. This is relatively frequent with false-positive contexts (see paragraph above). Table 1 shows, for each benchmark, the time of the static deadlock analysis and the number of generated deadlock cycles (column T_A/C), and, the overall time of the rest of the process (column T), which includes both the time of the generation of contexts and the symbolic executions. Times are in milliseconds except where indicated and are obtained on an Intel(R) Core(TM) i7 CPU at 2.5GHz with 8GB of RAM, running Ubuntu 5.4.0. A timeout of 5s is set for each symbolic execution and an asterisk in the time indicates the timeout has been reached at least once.

Overall, our deadlock guided generation of initial contexts hence enables our deadlock detection framework to analyze systems without the need of any user supplied initial context. Also, it allows generating concrete test cases that lead to deadlock for integration and system testing.

6 Conclusions and Related Work

We have proposed a framework for the automatic generation of initial contexts for deadlock-guided symbolic execution. Such initial contexts are composed of the interfering tasks which, according to a static deadlock analyzer, might lead to deadlock. Given the initial contexts, we can drive symbolic execution towards paths that are more likely to manifest a deadlock, discarding safe contexts. There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [9,10] and thread-based programs [17,19], is based on static analysis techniques. Although we have used the static analysis of [9], the information provided by other deadlock analyzers could be used in an analogous way. Deadlock detection has been also studied in the context of dynamic testing and model checking [6,15,16], where sometimes has been combined with static information [1,14]. The initial contexts generated by our framework are of interest also in these approaches. As regards the application in a thread-based concurrency model, the fundamental difference is that our whole approach is defined at the level of atomic tasks that execute concurrently using non-preemptive scheduling, unlike thread-based preemption. However, our approach would be adaptable to thread-based applications that rely on synchronized blocks of code (such as in monitors or concurrent objects). As future work, we plan to investigate how our framework could be adapted to this model.

References

1. R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Conf. on Hardware and Software Verification and Testing*, volume 3875 of *Lecture Notes in Computer Science*, pages 191–207. Springer, 2006.
2. E. Albert, M. Gómez-Zamalloa, and M. Isabel. Deadlock Guided Testing in CLP. Technical report, 2017. Available at `http://costa.ls.fi.upm.es/papers/costa/AlbertGI17tr.pdf`.

3. Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. Combining Static Analysis and Testing for Deadlock Detection. In *Proc. of IFM'16*, volume 9681 of *LNCS*, pages 409–424. Springer, 2016.
4. Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A Systematic Testing Tool for Concurrent Objects. In *Proc. of CC'16*. ACM, 2016.
5. Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. On the generation of initial contexts for effective deadlock detection. Technical report, October 2017. Available at `https://arxiv.org/abs/1709.04255`.
6. Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013.
7. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
8. C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.
9. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS 7892, pages 273–288. Springer, 2013.
10. E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.
11. Kobayashi N. Giachino E. and Laneve C. Deadlock analysis of unbounded process networks. In *CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 63–77, 2014.
12. Laneve C. Giachino E. and Lienhardt M. A framework for deadlock detection in core ABS. *Software and System Modeling*, 15(4):1013–1048, 2016.
13. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*. Springer, 2012.
14. P. Joshi, M. Naik, K. Sen, and Gay D. An effective dynamic analysis for detecting generalized deadlocks. In *Proc. of FSE'10*. ACM, 2010.
15. P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proc. of PLDI'09*. ACM, 2009.
16. A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report, 2013. Available at `http://dslab.epfl.ch/pubs/lockout.pdf`.
17. S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *Parallel and Distributed Debugging*, pages 97–107. ACM, 1991.
18. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering Methodology*, 14:1–41, 2005.
19. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
20. Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.