Deadlock-Guided Testing in CLP

Miguel Isabel Márquez

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE MÁSTER
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Junio 2017

Directores:
Elvira Albert Albiol
Miguel Gómez-Zamalloa Gil

Deadlock-Guided Testing in CLP

Trabajo de Fin de Máster realizado por
Miguel Isabel Márquez

Dirigido por
Elvira Albert
Miguel Gómez-Zamalloa Gil

Calificación: 10 - Sobresaliente

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Julio 2017

Autorización de difusión

Miguel Isabel Márquez

Julio 2017

El abajo firmante, matriculado en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Deadlock-Guided Testing in CLP", realizado durante el curso académico 2016-2017 bajo la dirección de Elvira Albert y Miguel Gómez-Zamalloa Gil en el Departamento de Sistemas Informáticos y Computación (DSIC), y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.
Firmado:

Miguel Isabel Márquez

Contents

Agradecimientos

Aprender nuevos conocimientos es una de mis pasiones y, por ello, me gustaría agradecer a mis directores *Elvira* y *Miky*, que consiguen motivarme y sacar lo mejor de mí, siempre proponiéndome nuevos problemas a resolver. También, a los miembros del grupo COSTA, por estar siempre dispuestos a echar una mano y, en especial, a *Puri*, que ha hecho una profunda revisión de este trabajo hasta desentrañar los más oscuros detalles.

He podido terminar este trabajo entre risas y buen humor, gracias a todos los compañeros que han pasado por el Aula 16. No han sido unos años fáciles, pero habéis estado ahí para mí siempre que lo he necesitado. En especial, a *Pablo*, por ser el hombro incondicional al que contarle mis penas y alegrías.

No habría escrito de tan buen agrado este trabajo, sin las tardes de calor insoportable en casa y desesperados por terminar nuestras respectivas memorias. Eso sí, siempre con una sonrisa en la cara. Gracias *Cristhian, me quedo contigo.*

A mi familia, ya que son quienes más orgullosos están de este trabajo y, sin embargo, no entienden ni una palabra de lo que está escrito. En especial, a mi madre, con quién, probablemente, me he pasado la mitad de este trabajo discutiendo y la otra mitad encerrado en mi habitación, trabajando. *Tienes una paciencia infinita.* Por último, a *Hugo* y *Alejandra*, a quienes espero un día poder contarle con detalles lo que significa ser *ingeniero informático, os quiero.*

Resumen

Los análisis estáticos de *deadlock* son, a menudo, capaces de asegurar la ausencia de bloqueos, pero cuando detectan un posible ciclo de *deadlock*, la información que devuelven como salida es escasa e insuficiente. Debido al complejo flujo de ejecución existente en los programas concurrentes, el usuario podría ser incapaz de encontrar la causa del comportamiento anómalo a partir de la información abstracta proporcionada por el análisis estático.

Este trabajo propone el uso combinado de análisis estático y el *testing* para la detección efectiva de *deadlocks* en programas asíncronos. Estos programas imperativos son primero traducidos a una versión declarativa (CLP), de manera que la aproximación combinada es llevada a cabo completamente por el mecanismo de *backtracking* inherente y al manejo de restricciones de CLP. Cuando el programa encuentra un deadlock, el uso combinado del *testing* y el análisis estático nos proporciona una técnica efectiva para encontrar trazas de *deadlock*. En caso de que el programa no contenga ninguno, pero el analizador sí que los encuentre por pérdidas de precisión, nosotros podríamos ser capaces de demostrar la ausencia de *deadlock*. Los principales resultados de este trabajo han sido presentados a:

- la edición especial *Computacional Logic for Verification* de la revista *Theory and Practice of Logic Programming* y
- la conferencia internacional de *Logic-Based Program Synthesis and Transformation (LOPSTR'17)*

y se encuentran actualmente bajo revisión.

Todas las técnicas desarrolladas en este trabajo han sido implementadas en la herramienta SYCO, la cual ha sido aceptada y presentada en la conferencia internacional de *Compilers Construction (CC'16)* [10].

Palabras Clave

Testing, Detección de deadlocks, Ejecución simbólica, Generación de casos de prueba, Verificación, Análisis de deadlock

Abstract

Static deadlock analyzers might be able to verify the absence of deadlock. However, they are usually not able to detect its presence. Also, when they detect a potential deadlock cycle, they provide little (or even no) information on their output. Due to the complex flow of concurrent programs, the user might not be able to find the source of the anomalous behaviour from the abstract information computed by static analysis. This work proposes the combined use of static analysis and testing for effective deadlock detection in asynchronous programs. The asynchronous program is first translated into a CLP-version so that the whole combined approach is carried out by relying on the inherent backtracking mechanism and constraint handling of CLP. When the program features a deadlock, our combined use of analysis and testing provides an effective technique to catch deadlock traces. While if the program does not have deadlock, but the analyzer inaccurately spotted it, we might be able to prove deadlock freedom. The main results in this project have been submitted to:

- *the special issue on Computational Logic for Verification* of the journal *Theory and Practice of Logic Programming* and
- *the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'17)*,

and are currently under revision.

The techniques developed in this work has been implemented in the SYCO tool, which appears in the proceedings of the *25th International Conference on Compilers Construction (CC'16)* [10].

Keywords

Testing, Deadlock detection, Symbolic execution, Test case generation, Verification, Deadlock analysis

Chapter 1

Introduction

A deadlock occurs when a concurrent program reaches a state in which one or more tasks are waiting for each other termination and none of them can make any progress. Deadlocks are one of the most common errors in concurrent programming and, thus, a main goal of verification and testing tools is, respectively, proving deadlock freedom and *deadlock detection*. We consider an *asynchronous* language which allows spawning asynchronous tasks at distributed locations, with no shared memory among them, and with an operation for *blocking* synchronization with the termination of asynchronous tasks. In this setting, in order to detect deadlocks, all possible *interleavings* among tasks executing at the distributed locations must be considered. Basically, each time that the processor can be released, any of the available tasks can start its execution, and all combinations among the tasks must be tried, as any of them might lead to deadlock.

Static analysis and testing are two different ways of detecting deadlocks. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks or months after releasing the application. This aspect of static analysis is especially important in security assurance – security attacks try to exercise an application in unpredictable and untested ways. However, due to the use of approximations, most static analyses can only verify the absence of deadlock but not its presence, i.e., they can produce false positives. Moreover, when a potential deadlock is detected, state-of-the-art analysis tools [15, 17, 18] provide little (and often no) information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and locations), it is essential to know the task interleavings that have occurred and the locations involved in the deadlock, i.e., provide a concrete *deadlock trace* that allows the programmer to identify and fix the problem.

In contrast, testing consists in executing the application. In dynamic testing, the application is executed for concrete input values, while in static testing it is executed symbolically (i.e., without any knowledge on the input variables). Since a deadlock can manifest only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings. The primary advantage of *systematic testing* [11, 30] for deadlock detection is that it can provide the detailed deadlock trace. There are two shortcomings though: (1) Although recent research tries to avoid redundant exploration as much as possible using Partial Order Reduction (POR) techniques [1, 6, 11, 14], the search space (even without redundancies) can be huge. This is a

threat to the application of systematic testing in concurrent programming. (2) In dynamic testing, one can only guarantee deadlock freedom for finite-state terminating programs (i.e., terminating executions for concrete inputs). In static testing, one needs to assume some termination criteria (e.g., loops can only be executed a fixed number of iterations) and, thus, deadlock freedom can be ensured for the considered termination criterion (e.g., the program is deadlock free provided its loops are unrolled the fixed number of iterations).

This work proposes a seamless combination of static analysis and testing for effective deadlock detection as follows: an existing static deadlock analysis [15] is first used to obtain *abstract* descriptions of potential deadlock cycles which are then used to guide a testing tool in order to find associated deadlock traces (or discard them). When the program features a deadlock, our combined use of analysis and testing provides an effective technique to catch deadlock traces. The development of such combined framework requires: (1) The use of a backtracking mechanism to prune those paths that are deadlock free as soon as possible, and keep on exploring the search tree in order to find deadlock paths (if there are), and (2) handling the constraints that describe the potential deadlock cycles as well as the path conditions. Our framework is based on *Constraint Logic Programming over Finite Domains CLP(FD)*, which imposes an integer domain for the program variables. Our framework benefits from the inherent mechanisms of constraint logic programming for achieving the above requirements as follows: the asynchronous program is translated into a CLP equivalent program so that the whole combined approach is carried out by relying on the inherent backtracking mechanism and constraint handling applied on the CLP-translated program.

1.1 Summary of Contributions

In summary, the main contributions of this project are the following:

1. We extend a standard semantics for asynchronous programs with information about the task interleavings made and the status of tasks.

2. We provide a formal characterization of *deadlock state* which can be checked along the execution and allows us to early detect deadlocks.

3. We present a new methodology to detect deadlocks that combines testing and static analysis as follows: the deadlock cycles inferred by static analysis are used to guide the testing process towards paths that might lead to a deadlock cycle while discarding deadlock-free paths. Our method can be used both for static and dynamic testing.

4. We introduce several deadlock-based testing criteria to find the first deadlock trace, a representative trace for each deadlock cycle, or all deadlock traces.

5. We propose a new algorithm to infer which tasks produce or may produce conflicting interactions from the deadlock cycles. This information is useful to discard initial states or contexts that cannot lead to deadlock.

6. We implement our methodology in the SYCO/`aPET` testing system and perform a thorough experimental evaluation on some classical examples.

7. We develop two new user-friendly web interfaces for SYCO and `aPET` which are well suited for any kind of user. They can be used online (see Chapter 6).

The main results in this project have been submitted to:

- *the special issue on Computational Logic for Verification* of the journal *Theory and Practice of Logic Programming*, where we present the basic framework for static testing and

- *the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'17)*, where we present the inference of initial contexts.

Both submissions are currently under revision.

This project is a revised and extended version of my Bachelor's Final Project [22] and a conference paper that appeared in the proceedings of iFM'16 [9]. There are three fundamental contributions w.r.t. [22]. The first one is that we extend our approach and our implementation to the *static* testing setting, i.e., we can apply symbolic execution without any knowledge on the input data in order to find deadlock traces. In contrast, [22] is defined for *dynamic* testing which requires the use of particular values for the input data. This has a clear theoretical interest as it generalizes the previous work to a static setting. Besides, the practical impact of such extension is important as now we can use it to prove deadlock freeness, i.e., if we are able to symbolically execute the whole program without finding any deadlock trace, nor termination problem, then the program is proven to be deadlock free. Note that the use of symbolic execution requires the use of termination strategies, since when the program contains loops or recursion, we might need a termination criterion (e.g., one that allows unfolding loops a constant number of times k) to guarantee termination of the process. In such case, our combined approach would prove deadlock freeness up for the selected termination criterion. The second contribution is that our whole approach is formalized in CLP, while [22] was developed on the imperative language. This gives us the advantages mentioned above, in particular the implicit support for backtracking greatly simplifies the definition of the symbolic testing framework. Finally, [22] presented a prototype tool SYCO, which is implemented in *Prolog*, but it is not very suitable for end-users. As a third fundamental contribution, we have developed a new user-friendly web interface for SYCO, which appears in the proceedings of CC'16 [10]. This tool is available for online use at `http://costa.ls.fi.upm.es/syco`.

1.2 Organization of the project

The remainder of the work is organized as follows. Chapter 2 presents the syntax and semantics of the asynchronous programs that we consider. Essentially, it includes three instructions for concurrency: create a new concurrency unit, spawn a task on a concurrency unit, and block the execution of a concurrency unit until a spawned task has finished its execution. Section 2.1 motivates our work by means of an example that will be used throughout the work to explain the different concepts and techniques.

Chapter 3 recaps the CLP-based testing framework that is the basis of our combined approach. We will first review the translation of standard imperative programs into CLP-equivalent ones. Next we describe how to perform systematic testing on the concurrent CLP-translated programs. This already requires the use of backtracking to exercise all possible task interleavings in a non-deterministic way. Finally, we leverage the testing framework to the static context in which one does not assume any information on the input data.

Chapter 4 contains the main contributions of this work. We first introduce the interleavings table in which we store the decisions about task interleavings made during the execution. Using the interleavings table, we provide a technique for the early detection of deadlock states during the execution. Finally, we will present our combined testing and static analysis framework for detecting deadlock traces. Chapter 5 provides a new technique to generate *the most interesting* initial contexts (discarding others) for the symbolic execution based on the information stored by the abstract deadlock cycles.

Chapter 6 describes the implementation of SYCO and `aPET` and shows the users how to use them through a detailed step-by-step tutorial. Chapter 7 describes our experimental evaluation that demonstrates the applicability and effectiveness of the proposed techniques. Chapter 8 reviews related work and we conclude in Chapter 9 pointing out several directions for future research.

Chapter 2

The language: Syntax and Semantics

We consider a distributed programming model with explicit 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. Besides accessing its own processor's global storage, each task can post tasks to the queues of any processor, including its own, and synchronize with the termination of tasks. The language uses *future variables* [12] for synchronizing program execution with the completion of asynchronous tasks. A future variable acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete. When the future variable is ready, the result can be retrieved. The declaration of a future variable is as follows Fut<T> f, where T is the type of the result r. An asynchronous call $\mathsf{m}(\bar{\mathsf{z}})$ spawned at location x is associated with a future variable f as follows $\mathsf{f} = \mathsf{x}\ !\ \mathsf{m}(\bar{\mathsf{z}})$. Instruction $\mathsf{r} = \mathsf{f}.\textbf{get}$ allows blocking the execution until the task executing m that is associated to f terminates, and it retrieves the result in r. When a task is completed, its processor becomes idle again, chooses the next pending task, and so on. The number of distributed locations need not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to a *concurrent object* that can be dynamically created using the instruction **new**. For instance, the instruction $\mathsf{b} = \textbf{new}\ \mathsf{Location}()$; creates a distributed location of type Location which is referenced by b. The program consists of a set of classes that define the types of locations, each of them defines a set of fields and methods of the form $\mathsf{M} ::= \mathsf{T}\ \mathsf{m}(\bar{\mathsf{T}}\ \bar{\mathsf{x}})\{\mathsf{s}\}$, where statements s take the form

$$
\begin{array}{rl}
\mathsf{s} ::= & \mathsf{s}; \mathsf{s} \\
\mid & \mathsf{x} = \mathsf{e} \\
\mid & \textbf{if}\ \mathsf{e}\ \textbf{then}\ \mathsf{s}\ \textbf{else}\ \mathsf{s} \\
\mid & \textbf{while}\ \mathsf{e}\ \textbf{do}\ \mathsf{s} \\
\mid & \textbf{return}\ \mathsf{x}; \\
\mid & \mathsf{b} = \textbf{new}\ \mathsf{T}(\bar{\mathsf{z}}) \\
\mid & \mathsf{f} = \mathsf{x}\ !\ \mathsf{m}(\bar{\mathsf{z}}) \\
\mid & \mathsf{x} = \mathsf{f}.\textbf{get}
\end{array}
$$

where x is a variable, e an expression, b a reference, $\bar{\mathsf{z}}$ a list of argument values, and f a future variable. All methods must return something, hence **void** methods are by-default expressed as **int** methods returning 0. For the sake of generality, the syntax of expressions e and types

$$(\textsc{mstep})\ \frac{selectTask(S) = tk,\ S = loc(o, \bot, h, \mathcal{Q} \cup \{tk\}) \cdot S' \qquad loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot S' \overset{o \cdot tk}{\rightsquigarrow}{}^{*}\ S''}{S \xrightarrow{o \cdot tk} S''}$$

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

$$(\textsc{async})\ \frac{\begin{array}{c} tk = tsk(tk, m, l, \mathsf{y = x\ !\ m_1(\bar{z})}; \mathsf{s}),\ l(x){=}o_1,\ \text{fresh}(tk_1),\ \text{fresh}(f_1), \\ l_1{=}buildLocals(\bar{z}, m_1, l),\ l' = l[y \to f_1] \end{array}}{\begin{array}{l} loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \cdot loc(o_1, \_, \_, \mathcal{Q}_1) \rightsquigarrow loc(o, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l', \mathsf{s})\}) \cdot \\ loc(o_1, \_, \_, \mathcal{Q}_1 \cup \{tsk(tk_1, m_1, l_1, body(m_1))\}) \cdot fut(f_1, o_1, tk_1, ini(m_1), \bot) \end{array}}$$

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

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

$$(\textsc{get2})\ \frac{tk{=}tsk(tk, m, l, \mathsf{x = y.}\textbf{get}; \mathsf{s}),\ l(y) = f,\ fut(f, \_, \_, \_, \bot) \in Futs}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \rightsquigarrow loc(o, tk, h, \mathcal{Q} \cup \{tk\})}$$

Figure 2.1: Macro-Step Semantics of Asynchronous Programs

T is left open.

Figure 2.1 presents the semantics of the language. A *state* or *configuration* is a set of locations and future variables $\{loc_0, ..., loc_n, fut_0, ..., fut_m\}$. A *location* is a term $loc(o, tk, h, \mathcal{Q})$ where o is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\bot$ if the location's lock is free, h is its local heap (a function that maps every field to its value), and $\mathcal{Q}$ is the set of tasks in the location. A *future variable* is a term $fut(id, o, tk, pp, r)$ where id is a unique future variable identifier, o is the location identifier that executes the task tk awaiting for the future, pp is the initial program point of tk, and r is the return value of the task tk, or $\bot$ if tk has not finished. A *task* is a term $tsk(tk, m, l, s)$ 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 (integers, booleans or references), and s is the sequence of instructions to be executed. We assume that the execution starts from a main method without parameters included in a Main class. The initial state is $S{=}\{loc(0, 0, \bot, \{tsk(0, main, l, body(main))\}), fut(0, 0, 0, ini(main), \bot)\}$ with an initial location with identifier 0 executing task 0 and its corresponding future variable. Here, l maps local variables to their initial values (**null** in case of reference variables), $\bot$ is the empty heap, $body(m)$ is the sequence of instructions in method m, $ini(m)$ refers to the first program point of method m, and we can know the program point pp where an instruction s is in the program as follows $pp{:}s$.

As locations do not share their states, the semantics can be presented as a macro-step semantics [30] (defined by means of the transition "$\longrightarrow$") in which the evaluation of all

statements of a task takes place serially (without interleaving with any other task) until it gets a **return** instruction. In this case, we apply rule MSTEP to select non-deterministically an available task from one *active* location in the state (i.e., a location with a non-empty task queue). The transition $\rightsquigarrow$ defines the evaluation within a given location. Rule NEWLOC creates a new location of class D without tasks, with a fresh identifier $fresh(o_1)$ and a new heap built with parameters $\bar{z}$. Rule ASYNC spawns a new task (whose arguments are initialized by function *buildLocals*) with a fresh task identifier tk_1, and it adds a new future variable to the state. We assume $o \neq o_1$, but the case $o = o_1$ is analogous, the new task tk_1 is added to $\mathcal{Q}$ of o. The rules for sequential execution are standard and are thus omitted.

When **return** is executed, the lock is released and will never be taken again by that task. Consequently, that task is *finished* and removed from the task queue, as rule RETURN claims. The instruction y.**get** waits for the future variable $\mathsf{y} \in \mathit{Futs}$ (where *Futs* is the future variables set in the state) without yielding the lock, i.e., it blocks the execution of the location until the task that is awaiting is finished (rule GET2). Then, when the future is ready, rule GET1 allows continuing the execution.

In what follows, a *derivation* or *execution* $E \equiv S_0 \longrightarrow \cdots \longrightarrow S_n$ is a sequence of macro-steps (applications of rule MSTEP). The derivation is *complete* if S_0 is the initial state and there is no exists a state S_{n+1} such that $S_n \longrightarrow S_{n+1}$ and $S_n \neq S_{n+1}$. Since the execution is non-deterministic, multiple derivations are possible from a state. Given a state S, $exec(S)$ denotes the set of all possible complete derivations starting at S. We sometimes label transitions with $o \cdot tk$, the name of the location o and task tk selected (in rule MSTEP) or evaluated in the step (in the transition $\rightsquigarrow$). The systematic exploration of $exec(S)$ thus corresponds to the standard systematic testing setting in which all possible explorations are performed without eliminating any redundancy.

2.1 Motivating Example

Our running example simulates a simple communication protocol among a database and n workers. Our implementation in Figure 2.2 has three classes, a Main class which includes the main method, and classes Worker and DB implementing the workers and the database, respectively. The main method just calls method simulate with the number of workers to create in its parameter (in this case, 1). Method simulate creates the database and the n workers, and invokes methods register and work on each of them, respectively. The work method of a worker simply accesses the database (invoking asynchronously method getData) and then blocks until it gets the result, which is assigned to its data field. The register method of the database registers the provided worker reference adding it to its clients list field. In case checkOn is true, before adding the worker, it makes sure that the worker is online. This is done by invoking asynchronously method ping with a concrete value and blocking until it gets the result with the same value. Method getData of the database returns its data field if the caller worker is registered, otherwise it returns **null**. Depending on the sequence of interleavings, the execution of this program can finish:

- (I) As one would expect, i.e., with worker.data = db.data,
- (II) with worker.data = **null**, or,

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

Figure 2.2: Working example: Communication protocol among a DB and n workers.

- (III) in a deadlock.

(I) happens when the worker is registered in the database (assignment in Line 25) before getData is executed. (II) happens when getData is executed before the assignment at Line 25. A deadlock is produced if both register and work start executing before getData and ping.

Figure 2.3 shows the derivation tree computed by a systematic testing of the main method. Derivations that contain a dotted node are not deadlock, while those with a black node are deadlock. A main motivation of our work is to detect as early as possible that the dotted derivations will not lead us to deadlock and prune them. All derivations which can be found in this program start by executing methods main and simulate, one after the other. The 1st and 2nd derivations finish with the expected output state (scenario I above), the 3rd and 4th branches are deadlocks (scenario III), and finally, the 5th and 6th derivations correspond to scenario II. Let us see two selected branches in detail. In the first derivation, the third macro-step executes register on location db and then ping on w. It is clear that location db will not deadlock in this derivation, since the **get** at Line 25 will succeed and location w will also be able to complete its tasks, namely, the **get** at Line 38 will succeed because the task in getData will eventually finish as its location is not blocked. However, in the third branch, we first select register (and block database waiting for the termination of ping), and then we select work (blocking w waiting for the termination of getData). The **get** at Line 38 will never succeed since it is awaiting for the termination of a task of a blocked location. Hence, we

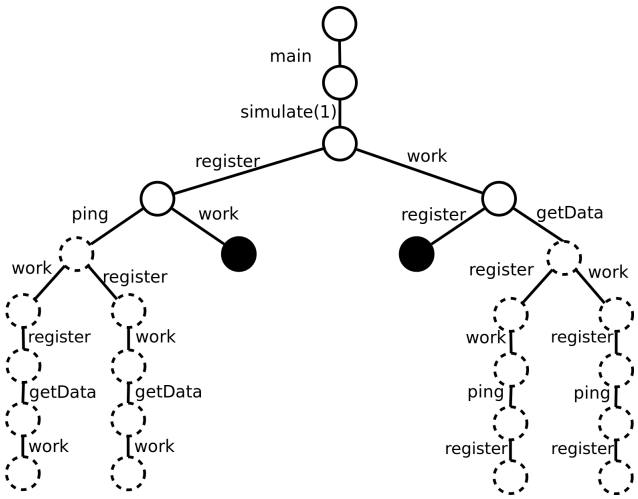

Figure 2.3: Systematic testing tree of working example

have a deadlock. Let us outline five states of this derivation:

$$S_0 \equiv loc(0,0,\_,\{tsk(0,main,\_,\_)\})\cdot fut(f_0,0,0,2,\bot) \xrightarrow{0.0}$$
$$S_1 \equiv loc(0,0,\_,\{tsk(1,simulate,\_,\_)\})\cdot fut(f_0,0,0,2,0)\cdot fut(f_1,0,1,6,\bot) \xrightarrow{0.1}$$
$$S_2 \equiv loc(0,0,...)\cdot loc(db,\bot,h_{db},\{tsk(2,reg,\_,\_)\})\cdot loc(w,\bot,h_w,\{tsk(3,wo,\_,\_)\})\cdot$$
$$\cdot fut(f_0,0,0,2,0)\cdot fut(f_1,0,1,6,0)\cdot fut(f_2,db,2,22,\bot)\cdot fut(f_3,w,3,36,\bot) \xrightarrow{db.2}$$
$$S_3 \equiv loc(db,2,\_,\{tsk(2,reg,f_4.\textbf{get})\})\cdot loc(w,\bot,\_,\{tsk(4,ping,\_,\_),\_\})\cdot ...\cdot fut(f_4,w,4,41,\bot) \xrightarrow{w.3}$$
$$S_4 \equiv loc(w,3,\_,\{tsk(3,wo,f_5.\textbf{get})\})\cdot loc(db,2,\_,\{tsk(5,getData,\_,\_),\_\}\cdot ...\cdot fut(f_5,db,5,29,\bot)$$

The third state S_2 is obtained after executing methods main and simulate. It contains the initial location 0, the two locations created at Lines 7 and 9 resp., and the two tasks created at Lines 10 and 11 resp. in the corresponding queues. Note that each location and task is assigned a unique identifier (we use numbers as identifiers for tasks and short names as identifiers for locations). In the next state, the task register has been selected and fully executed (we have shortened the name of the methods, e.g., reg for register). Let us observe the addition of the future variable created at Line 24 at S_3. In S_4 we have executed task work in the worker and added a new future variable. This state is already deadlock as we will see in Section 4.2. In the rest of the work, for clarity we use the location and task names instead of numeric identifiers.

Chapter 3

A CLP-based Testing Framework

In this chapter we review the CLP-based testing framework for concurrent asynchronous programs developed in previous work (namely it comprises the following three articles: [5, 19, 20]). The framework consists in two main phases. (1) The source asynchronous program is transformed into an equivalent CLP program, which includes calls to specific operations to handle non-native CLP features that include concurrency builtins for asynchronous calls, task scheduling, and synchronization. The *CLP-transformed* program can be executed in CLP as long as such operations are encoded in CLP. (2) Systematic and symbolic execution is performed using CLP's execution mechanism over the CLP-transformed program. The use of CLP as the enabling technology for the testing framework has the following important advantages that will become apparent throughout this chapter:

- CLP is well known to be a very appropriate paradigm to do meta-programming.
- CLP's native backtracking mechanism allows systematically executing the program in order to try out all non-deterministic task interleavings.
- CLP's backtracking and constraint solving mechanisms allow performing symbolic execution and test case generation [20].

3.1 CLP-translated Programs

Constraint Logic Programming is a programming paradigm that extends Logic Programming with *Constraint solving*. It augments the LP expressive power and application domain while maintaining its semantic properties (e.g., existence of a fixpoint semantics). In CLP, the bodies of clauses may contain constraints in addition to ordi- nary literals. CLP integrates the use of a constraint solver to the operational semantics of logic programs. As a consequence of this extension, whereas in LP a computation state consists of a goal and a substitution, in CLP a computation state also contains a *constraint store*. The special *constraint* literals are stored in the constraint store instead of being solved according to SLD-resolution. The satisfiability of the constraint store is checked by a constraint solver. The CLP paradigm can be instantiated with many constraint domains. A particularly useful constraint domain is CLP(FD) (Constraint Logic Programming over Finite Domains). CLP(FD) constraints

are usually intended to be arithmetic constraints over finite integer domain variables. Our framework will rely on CLP(FD) to translate conditional statements over integer variables into CLP constraints. Moreover, the labeling mechanism is essential to concretize the obtained test cases in order to obtain concrete input data amenable to be used and validated by testing tools. The translation of imperative, object-oriented, and concurrent programs into equivalent CLP-translated programs has been subject of previous work (see, e.g. [3, 5, 19]). Therefore, we only state the features of the translated programs without going into deep details of how the translation is done.

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

1. *In and Out are, respectively, the (possibly empty) list of input and output arguments.*
2. *S_{in} and S_{out} are, respectively, the input and (possibly modified) output states.*
3. *If m_i is defined by more than one rule, then g is the constraint that guards the execution of each rule, i.e., it must hold for the execution of the rule to proceed.*
4. *$b_1, \ldots, b_j$ is a sequence of instructions including arithmetic operations, calls to other predicates, field accesses, and concurrent operations as defined in Figure 3.1.*

Specifically, CLP-translated programs adhere to the grammar in Figure 3.1. As customary, terminals start with lowercase (or special symbols) and non-terminals start with uppercase; subscripts are provided just for clarity. Non-terminals *Block*, *Num*, *Var*, *PP*, *MSig*, *FSig*, and C denote, respectively, the set of predicate names, numbers, variables, source program-points, method signatures, field signatures, and class names. A clause indistinguishably defines either a method which appears in the original source program (*MSig*), or an additional predicate which corresponds to an intermediate block in the control flow graph of the original program (*Block*). A field signature *FSig* contains the field name. An asterisk on a non-terminal denotes that it can be either as defined by the grammar or a (possibly constrained) variable (e.g., Num^*, denotes that the term can be a number or a variable). Location references are written as terms of the form $r(Ref)$ or *null*. As expected, the operation `new_location`$(S_{in}, C, Ref, S_{out})$ creates a new location of class C in state S_{in} and returns its assigned reference *Ref* and the updated state S_{out}; `get_field`$(S, Ref, FSig, V)$ retrieves in variable V the value of field *FSig* of the location referenced by *Ref* in the state S; and, `set_field`$(S_{in}, Ref, FSig, V, S_{out})$ sets the field *FSig* of the location referenced by *Ref* in S_{in} to V and returns the modified state S_{out}. Implementations of these operations in CLP can be found in [20]. Operations `async`/6, `get`/6 and `return`/3 simulate in CLP the asynchronous concurrency and are explained later.

Figure 3.2 shows (a simplified and pretty-printed version of) the CLP-translated program of our working example. Let us observe the following:

- Conditional statements and loops in the source program are transformed into guarded rules and recursion in the CLP program, respectively. E.g. the **if** of the getData method

$Clause ::= Pred(Args_{in},Args_{out},S_{in},S_{out})$:- $[G,]B_1,B_2,\ldots,B_n.$
$G ::= Num^*\ ROp\ Num^* \mid Ref_1^*\ \backslash== Ref_2^*$
$B ::= Var\ \#= Num^*\ AOp\ Num^*$
$\mid Pred(Args_{in},Args_{out},S_{in},S_{out})$
$\mid$ `new_location`$(S_{in},C^*,Ref^*,S_{out}) \mid$ `async`$(S_{in},Ref^*,Call,Var,PP,S_{out})$
$\mid$ `return`$(S_{in},Var,S_{out}) \mid$ `get`$(S_{in},Var,Var,PP,Call,S_{out})$
$\mid$ `get_field`$(S_{in},Ref^*,FSig,Var) \mid$ `set_field`$(S_{in},Ref^*,FSig,Data^*,S_{out})$
$Call ::= Pred(Args_{in},Args_{out})$ $\quad Pred ::= Block \mid MSig$
$Args ::= [\,] \mid [Data^*|Args]$ $\quad ROp ::= \#> \mid \#< \mid \#>= \mid \#=< \mid \#= \mid \#\backslash=$
$Data ::= Num \mid Ref$ $\quad AOp ::= + \mid - \mid * \mid / \mid mod$
$Ref ::= null \mid r(Var)$ $\quad S ::= Var$

Figure 3.1: Syntax of CLP-translated programs

is encoded in CLP by means of predicate `if3/4`, whereas the **while** of the main method is encoded by the recursive rule `while/4`. Mutual exclusion between the rules of a predicate is ensured either by means of mutually exclusive *guards*, or by information made explicit on the heads of rules, as usual in CLP.

- A global state, which includes the locations (with their fields and task queues) and the future variables, is explicitly handled. Observe that each rule includes as arguments an input and an output state (3rd and 4th arguments respectively). The state is carried along the execution being used and transformed by the corresponding operations as a black box, therefore it is always a variable in the CLP program. The operations that modify the state are explained later.

- An additional predicate is produced for the continuation after a **get** statement. The call to such continuation predicate is included within the arguments (5th argument) of the `get`/6 operation (see e.g. the second rule of `if1/4`). This allows implementing in CLP the concurrent behavior of asynchronous programs. Namely, a task can suspend its execution due to a blocking synchronization, and afterwards, when the awaiting task has finished, the task has to resume its execution at this precise point.

- Exceptional behavior is ignored for simplicity but can be easily handled by means of additional guarded rules and an exception flag in an additional parameter (see [20] for details).

- Predicates `contains/4` and `add/4` are the CLP-translated versions of the library operations `contains` and `add`. They are basically equivalent, resp., to the classical Prolog predicates `member/2` (returning true/false instead of succeeding/failing) and `append/3` (with a unitary list in the second argument with the element to be added). The last two arguments (input and output states) are ignored.

```
43 main([This],0,S0,S2) :-
44   async(S0,This,
45         simulate([This,1],_),_,6,S1),
46   return(S1,0,S2).
47
48 simulate([This,N],R,S0,S2) :-
49   new_location(S0,'DB',DB,S1),
50   while([N,DB],R,S1,S2).
51 while([N,_],0,S0,S1) :-
52   N #=< 0, return(S0,0,S1).
53 while([N,DB],R,S0,S4) :-
54   N #> 0,
55   new_location(S0,'Worker',W,S1),
56   async(S1,DB,
57        register([DB,W],_),_,22,S2),
58   async(S2,W,work([W,DB],_),_,36,S3),
59   N2 #= N - 1,
60   while([N2,DB],R,S3,S4).
61
62 register([This,W],R,S0,S1) :-
63   get_field(S0,This,checkOn,Ch),
64   if1([This,W,Ch],R,S0,S1).
65 if1([_,_,false],R,S0,S1) :-
66   cont1([],R,S0,S1).
67 if1([This,W,true],R,S0,S2) :-
68   async(S0,W,ping([W,5],_),F,41,S1),
69   get(S1,F,FV,25,if2([This,W,FV],R),S2).
70 if2([_,_,FV],R,S0,S1) :-
71   FV #\= 5,
72   cont1([],R,S0,S1).
73 if2([This,W,5],R,S0,S1) :-
74   get_field(S0,This,clients,Cls),
75   add([Cls,W],Cls2,_,_),
76   set_field(S0,This,clients,Cls2,S1),
77   cont1([],R,S1,S2).
78 cont1([],0,S0,S1) :- return(S0,0,S1).
79
80 getData([This,W],R,S0,S1) :-
81   get_field(S0,This,clients,Cls),
82   contains([Cls,W],RC,_,_),
83   if3([This,RC],R,S0,S1).
84 if3([_,false],null,S,S1) :-
85   return(S0,null,S1).
86 if3([This,true],Data,S0,S1) :-
87   get_field(S0,This,data,Data),
88   return(S0,Data,S1).
89
90 work([This,DB],R,S0,S2) :-
91   async(S0,DB,getData([DB],_),F,29,S1),
92   get(S1,F,FV,38,cont2([FV],R),S2).
93 cont2([FV],FV,S0,S1) :- return(S0,FV,S1).
94
95 ping([_,N],N,S0,S1) :- return(S0,N,S1).
```

Figure 3.2: CLP-translated program for the working example

3.2 Systematic (Concrete) Execution

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

3.2.1 The Global State

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

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

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

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

$$
\begin{array}{rl}
\texttt{Locs1} = & [\texttt{loc}(0, \texttt{active}(1), [\,], [\texttt{tsk}(1, \texttt{simulate}, \texttt{simulate}([\texttt{r}(0), 1], \texttt{Ret}), 6, \texttt{call})]), \\
& \texttt{loc}(1, \texttt{bot}, [\texttt{field}(\texttt{data}, ..), \texttt{field}(\texttt{clients}, \texttt{null}), \texttt{field}(\texttt{checkOn}, \texttt{true})], [\,]), \\
& \texttt{loc}(2, \texttt{bot}, [\texttt{field}(\texttt{data}, \texttt{null})], [\,])], \ \textit{and} \\
\texttt{Futs1} = & [\texttt{fut}(0, 0, 0, 2, 0), \texttt{fut}(1, 0, 1, 6, \texttt{bot})].
\end{array}
$$

As expected, two new locations appear in the state. None of them is executing, so their locks are bound to `bot`*; their fields have the initial values and their queues are empty since we have not performed any asynchronous call yet. Location* 0 *has finished task* `main`*, hence its related future variable (*`fut` 0*) has the return value* 0*. However, the return value of future variable* 1 *is* `bot` *since task* `simulate` *has not finished yet.* □

3.2.2 Distribution, Concurrency and Synchronization

Figure 3.3 shows the relevant part of the CLP implementation of the operations that simulate in CLP the distribution, concurrency, and synchronization of our asynchronous programs.

Essentially, they correspond to the rules with the same names in the semantics of Figure 2.1. Specifically:

- Predicate `async`/6, given a state S, builds and adds a call task with *Call* and its initial program point *PP* to the queue of location *LocId* (by means of predicate `add_task/7`); and builds and adds a *fut/5* term linked to the new task to the list of futures (by means of `add_future/6`), resulting in state $S3$. The call to `fresh_fut/1` in Line 105 produces a fresh integer future identifier which is stored in the corresponding *fut/5* term and bound to the local variable `Fut` in Line 106.

- Predicate `return`/3, given a state S, sets the value of the future variable corresponding to the task that has just finished with the value of local variable *Ret*, and then calls `mstep/3` to switch context to the following macro-step (if possible), resulting in the final state $S3$.

- Predicate `get`/6, given a state S, if the future variable *Fut* is ready, i.e., if it is not `bot`, (first rule) then the value of the future is bound to *FutVal* and the execution proceeds using the continuation predicate *Cont* with initial state S and final state $S2$ by means of `run_task/3`. Predicate `run_task/3` simply builds a full call putting together *Cont* and the states S and $S2$, and performs a meta-call with it.

- Predicate `mstep/3` either finishes the execution (first rule), in case there are no more tasks to be executed (`select_task/2` fails), or selects (non-deterministically) an enabled task *Task* from a non-blocked location (second rule) and continues the execution calling *Task* with initial state $S2$ and final state $S3$ (predicate `run_task/3`). Predicate `update_locks/4` sets accordingly the locks of both the location which was previously executing and the one which is about to start. The *Status* term, which is set in the calls to `mstep/3` from predicates `return`/3 (Line 110) and `get`/6 (Line 120), allows knowing whether the previous macro-step had ended in a **return** or in a blocked **get**, so that `bot` or *blocked*(*Id*) respectively is set on the previously active location's lock.

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

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

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

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

$$\begin{aligned}
\texttt{Locs2} \;=\; & [\texttt{loc}(0,\texttt{active}(1),[\,],[\texttt{tsk}(1,\texttt{simulate},\texttt{simulate}([\texttt{r}(0),1],\texttt{Ret}),6,\texttt{call})]),\\
& \texttt{loc}(1,\texttt{bot},[\texttt{field}(\texttt{data},_),\texttt{field}(\texttt{clients},\texttt{null}),\texttt{field}(\texttt{checkOn},\texttt{true})],\\
& \qquad [\texttt{tsk}(2,\texttt{register},\texttt{register}([\texttt{r}(1),\texttt{r}(2)],\texttt{Ret2}),22,\texttt{call})]),\\
& \texttt{loc}(2,\texttt{bot},[\texttt{field}(\texttt{data},\texttt{null})],[\,])],\ \textit{and}\\
\texttt{Futs2} \;=\; & [\texttt{fut}(0,0,0,2,0),\texttt{fut}(1,0,1,6,\texttt{bot}),\texttt{fut}(2,1,2,22,\texttt{bot})].
\end{aligned}$$

```
96 mstep(S,_,S) :-
97   \+ select_task(S,_).
98 mstep(S,Status,S3) :-
99   select_task(S,Task),
100  update_locks(S,Task,Status,S2),
101  run_task(Task,S2,S3).
102
103 async(S,LocId,Call,Fut,PP,S3) :-
104  add_task(S,LocId,call,Call,PP,TaskId,S2),
105  fresh_fut(FutId),
106  Fut = FutId,
107  add_future(S2,FutId,LocId,TaskId,PP,S3).
108 return(S,Ret,S3) :-
109  set_fut_value(S,Ret,S2),
110  mstep(S2,return,S3).
111
112 get(S,Fut,FutVal,_PP,Cont,S2) :-
113  ready(S,Fut), !,
114  get_fut_value(S,Fut,FutVal),
115  run_task(Cont,S,S2).
116 get(S,Fut,FutVal,PP,Cont,S3) :-
117  get_this_id(S,ThisId),
118  add_task(S,ThisId,get(Fut,FutVal),
119           Cont,PP,_,S2),
120  mstep(S2,PP:Fut:get,S3).
```

Figure 3.3: CLP operations for distribution, concurrency, and synchronization

We can observe how asynchronous calls do not transfer control from the caller, i.e., they are not executed when they occur, but rather added as pending tasks on the receiver locations that will eventually schedule them for execution. We now continue the execution until the end of method `simulate` *(Line 60). The* `return` *operation sets the value of the future and calls* `mstep/3` *to switch context and select non-determinitically the task to be executed next. Let us assume task* `work` *is chosen and starts to execute. The call to the* `get` *at Line 92 first checks whether the future variable of the call to* `getData` *is already available. Since it is not the case (i.e,* `Fut` *is bound to* `bot`*) the execution of the current task cannot proceed, therefore the corresponding* **get** *task is added to the current location (so that it can be resumed later on) and* `mstep/3` *is called again. The current state is* `(Locs3,Futs3)` *where:*

$$\begin{aligned}\texttt{Locs3} = [&\texttt{loc}(0,\texttt{active}(1),[\,],[\,]),\\ &\texttt{loc}(1,\texttt{bot},[\texttt{field}(\texttt{data},..),\texttt{field}(\texttt{clients},\texttt{null}),\texttt{field}(\texttt{checkOn},\texttt{true})],\\ &\qquad[\texttt{tsk}(2,\texttt{register},\texttt{register}([\texttt{r}(1),\texttt{r}(2)],\texttt{Ret2}),22,\texttt{call})\\ &\qquad\texttt{tsk}(4,\texttt{getData},\texttt{getData}([\texttt{r}(1),\texttt{r}(2)],\texttt{Ret4}),29,\texttt{call})]),\\ &\texttt{loc}(2,\texttt{bot},[\texttt{field}(\texttt{data},\texttt{null})],[\texttt{tsk}(3,\texttt{work},\texttt{cont2}([\texttt{FV}],\texttt{Ret3}),38,\texttt{get})])]),\ \textit{and}\\ \texttt{Futs3} = [&\texttt{fut}(2,1,2,22,\texttt{bot}),\texttt{fut}(3,2,3,36,\texttt{bot}),\texttt{fut}(4,1,4,29,\texttt{bot}),...].\end{aligned}$$

The full derivation tree contains 6 *successful branches that correspond to those of Figure 2.3.* □

3.3 Symbolic Execution

It is shown in [20] that in the context of a sequential source language without support for dynamic memory, the symbolic execution of a method using the CLP-translated program is attained by simply using the standard CLP execution mechanism just calling the corresponding predicate with all arguments being free variables. However, in the case of heap-manipulating programs, the straighforward definition of the heap-related operations one could imagine fall short to generate arbitrary heap-allocated data structures correctly

handling aliasing during symbolic execution. In [20], we can find the definition of heap-related operations that allow performing symbolic execution and that can be used directly in our CLP-based framework for asynchronous programs. Given the heap-related operations and the CLP operations for concurrency, the inherent constraint solving and backtracking mechanisms of CLP provide the support for keeping track of path conditions (or constraint stores), failing and backtracking when unsatisfiable constraints are hit, hence discarding such execution paths; and succeeding when satisfiable constraints lead to a terminating state in the program. Thus, the symbolic execution of a method `m` in our source asynchronous program is performed in our CLP-based framework just by launching in CLP the goal

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

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

Example 3. *Let us perform a symbolic execution of method* `getData` *with a completely unknown input. Specifically, the input arguments are* `In = [r(0),r(W)]`*, and the initial state is* `SIn = ([L0|RL],[F0])` *being*

$$\texttt{L0} = \texttt{loc}(0, \texttt{active}(0), [\texttt{field}(\texttt{data}, \texttt{D}), \texttt{field}(\texttt{clients}, \texttt{Cls}), \texttt{field}(\texttt{checkOn}, \texttt{T})], [\texttt{tsk}(0, \texttt{getData}, \texttt{getData}(\texttt{In}, \texttt{Ret}), 29, \texttt{call})]) \text{ and}$$
$$\texttt{F0} = \texttt{fut}(0, 0, 0, 29, \texttt{bot}),$$

The following path conditions are obtained for the first three explored derivations:

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

□

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

In general, in symbolic execution, as soon as the source program has loops or recursion whose termination depends on unknown data, the derivation tree can be infinite. It is therefore necessary to establish a termination criterion, which guarantees that the number of paths traversed remains finite, while at the same time the obtained coverage on the source program is meaningful. One of the standard termination/coverage criteria is the *loop-k* criterion, which limits to a certain threshold the allowed number of iterations on each loop (or recursive calls). E.g., in the above example, if we set $k = 1$, the exploration finishes with the three successful derivations above. Unfortunately, in the context of concurrent programs,

the application of this criterion to all tasks of a state does not guarantee termination of the whole process. As studied in [7] it is required to take into account more factors that threaten termination. Namely, in symbolic execution, (1) we can switch from one task to another one an infinite number of times, and, (2) we can create an unbounded number of actors. Termination/coverage criteria taking into account those factors are defined in [7] and are simply adopted in our symbolic execution framework.

Example 4. *Let us now perform a symbolic execution of method* `simulate` *for an unknown input* `n`*. If we set* loop-k *with* $k = 1$*, we obtain seven derivations. The first one produces the condition* `N #=< 0` *and finishes with no workers in the output state and no executed tasks in the database. The other six derivations include the path-condition* `N = 1` *and correspond to those of Example 2 (see also Figure 2.3). In case the loop limit is* $k = 2$*, we obtain a total of* 1321 *derivations, out of which* 50 *of them are deadlock.* □

Chapter 4

Deadlock-Guided Testing

As already mentioned, systematically exploring all different task interleavings of a concurrent program, even applying the most advanced techniques to eliminate redundancies (see Chapter 8), presents scalability problems. In the case of symbolic execution, the problem is exacerbated by the intrinsic non-determinism of symbolic execution produced in branching statements involving partially unknown data. In this chapter we present two complementary techniques to reduce the state space exploration when the goal is to find deadlocks, both in concrete and symbolic execution. First, Section 4.1 proposes an extension of the program state to store information about the task interleavings made and the status of tasks. Section 4.2 provides a formal characterization of deadlock states using the above extension, which can be checked during the execution allowing us to early detect deadlocks. Section 4.3 presents a new methodology to guide the exploration towards paths that might lead to deadlock (discarding deadlock-free paths), by relying on the information inferred by a static deadlock analysis. Finally, Section 4.4 introduces several deadlock-based testing criteria focused on exposing deadlock bugs.

4.1 The Interleavings Table

The interleavings table stores all decisions about task interleavings made during the execution. This way, at the end of a concrete execution, the exact ordering of the performed macro-steps can be observed. Specifically, the Interleavings Table (IT) is defined as a mapping with entries of the form $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$, where:

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

$\mathtt{S}_1$	$t_{0,main,2} \mapsto \langle 0, return \rangle$	$\emptyset$
$\mathtt{S}_2$	$t_{0,simulate,6} \mapsto \langle 1, return \rangle$	$\emptyset$
$\mathtt{S}_3$	$t_{db,reg,22} \mapsto \langle 2, 25{:}f_4.\mathbf{get} \rangle$	$fut(f_4, w, ping, 41, \bot)$
$\mathtt{S}_4$	$t_{w,work,36} \mapsto \langle 3, 38{:}f_5.\mathbf{get} \rangle$	$fut(f_5, db, getData, 29, \bot)$

Table 4.1: Interleavings table for a derivation of our running example.

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

- We extend the program state with two new arguments: an *interleavings table* and a *clock*. The clock is a natural number. Its value in the initial state is 0, and it is incremented at each new macro-step. Then, a new program state is $s(Locs, Futs, IT, C)$, where *Locs* and *Futs* is the set of locations and future variables, respectively; IT is the interleavings table, and C is the current clock value.
- The initial state starts with an empty interleavings table and a value 0 for the clock.
- The second clause of rule `mstep/3` is extended as follows:

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

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

Example 5. *The interleavings table in Figure 4.1 is computed for the derivation in Section 2.1. It has as many entries as macro-steps in the derivation. We can observe that subsequent time values are assigned to each time identifier so that we can then know the order of execution. The rightmost column shows the future variables in the state that store the location and task they are bound to.* □

4.2 Early Deadlock Detection

Deadlocks can be easily detected in our CLP framework just by adding the following check to function *selectTask*: "if no task can be selected and there is at least a location with a non-empty task queue then there is a deadlock". However, deadlocks can be detected earlier.

To this aim, we present the notion of *deadlock state* which characterizes states that contain a *deadlock chain* in which one or more tasks are waiting for each other's termination and none of them can make any progress. Note that, from a deadlock state, there might be tasks that keep on progressing until the deadlock is finally made explicit. Even more, if one of those tasks runs into an infinite loop, the deadlock will not be captured using the above check. The early detection of deadlocks therefore allows reducing state exploration.

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

Definition 2 (Blocking Interval)**.** *Let $S = s(\_, Futs, IT, \_)$ be a state. We say that $I = (t_{stop}, t_{async}, t_{resume})$ is a* blocking interval *of S, written as $I \in S$, iff there exists $t_{stop} \equiv t_{o,tk_0,pp_0} \in IT$ such that* status$(t_{stop}, IT) = pp_1$:x.**get** *and $fut(x, o_x, tk_x, pp(M), \bot) \in Futs$ where:*

- $t_{resume} \equiv t_{o,tk_0,pp_1}$*, and*
- $t_{async} \equiv t_{o_x,tk_x,pp(M)}$*.*

In Condition 2, we can see that if the task starting at t_{async} has finished, then it is not a blocking interval. This is known by checking that this task has not reached return, i.e., $\nexists\, t \in suc(t_{async}, IT)$ such that $status(t) =$ **return** or $fut(x, o_x, tk_x, pp(M), \bot) \in Futs$. In Condition 1, we see that in ρ_{stop} we have the name of the future we are awaiting (whose corresponding information is stored in *fut*, Condition 2). In order to define t_{resume} in Condition 2, we search for the same task tk_0 and same location o that executes the task starting at program point pp_1 of the **get**, since this is the point that the macro-step rule uses to define the macro-step identifier t_{o,tk_0,pp_1} associated to the resumption of the waiting task.

Example 6. *Let us consider again the derivation in Section 2.1. We have the following blocking interval $(t_{db,reg,22}, t_{w,ping,41}, t_{db,reg,25}) \in S_3$ with $S_3 \equiv s(\_, Futs_3, IT_3, \_)$, since $t_{db,reg,22} \in IT_3$,* status$(t_{db,reg,22}, IT_3) = [25$:$f$.**get**$]$*, and $fut(f, w, ping, 41, \bot) \in Futs_3$. This blocking interval captures the fact that the task at $t_{db,reg,22}$ is blocked waiting for task* ping *to terminate. Similarly, we have the following interval in S_4: $(t_{w,work,36}, t_{db,getData,29}, t_{w,work,38})$.* □

The following notion of *deadlock chain* relies on the blocking intervals of Definition 2 in order to characterize chains of calls in which intuitively each task is waiting for the next one

to terminate until the last one which is waiting on the termination of a task executing on the initial location (that is blocked). Given a time identifier t, we use $loc(t)$ to obtain its associated location identifier.

Definition 3 (Deadlock Chain). *Let $S = s(\_, \_, IT, \_)$ be a state. A chain of time identifiers $t_0, ..., t_n$ is a* deadlock chain *in S, written as $dc(t_0, ..., t_n)$ iff*

$$\forall t_i \in \{t_0, ..., t_n\} \text{ s.t. } (t_i, t'_{i+1}, next(t_i)) \in S$$

and one of the following conditions holds:

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

and Condition 2 holds for t_n with $t_{n+1} \equiv t_0$.

Let us explain the two conditions in the above definition: In Condition 1, we check that when a task t_i is waiting for another task to terminate, the blocking interval contains the initial time t'_{i+1} in which the task will be selected. However, we look for any blocking interval for this task t_{i+1} (thus we check that t_{i+1} is a successor of time t'_{i+1}). As in Definition 6, this is because such task may have started its execution and then suspended due to a subsequent **get** instruction. Abusing terminology, we use the time identifier to refer to the task executing. In Condition 2, we capture deadlock chains which occur when a task t_i is waiting for the termination of another task t'_{i+1} which executes on a location $loc(t'_{i+1})$, which is blocked. The fact that is blocked is captured by checking that there is a blocking interval from a task t_{i+1} executing on this location. Finally, note the circularity of the chain, since we require that $t_{n+1} \equiv t_0$.

We state that *a state S is a deadlock state if and only if there exists a deadlock chain in S* and it will be proven in Theorem 3. This early deadlock detection is integrated into our CLP semantics just by adding the following clause as the new first clause of rule `mstep/3`.

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

where the `deadlock/1` predicate checks whether the conditions in Definition 3 hold.

We prove that our definition of deadlock is equivalent to the standard definition of deadlock of [15]. Definition 4 introduces the concept of *deadlock dependencies graph G_S* of the state S. Based on this definition, authors state that *if there is a cycle in the graph G_S, then the program (state S) is deadlock.*

Definition 4 (Deadlock Dependencies Graph). *Given a program state $S = s(Locs, Futs, \_, \_)$, where Locs and* `Futs` *are, respectively, the set of locations and futures, we define its* dependencies graph *G_S as the graph whose nodes are the existing location and task identifiers and whose edges are defined as follows:*

1. **Location-Task** : *$o \rightarrow tk_2$ iff there are two locations $loc(o, tk, h, \mathcal{Q})$, $loc(o_2, \_, h_2, \mathcal{Q}_2) \in Locs$, two tasks $tsk(tk, m, l, y.\textbf{get}; s) \in \mathcal{Q}$ $tsk(tk_2, m_2, l_2, s_2) \in \mathcal{Q}_2$ and a future $fut(y, o_2, tk_2, m_2, \bot) \in Futs$.*
2. **Task-Task** : *$tk_1 \rightarrow tk_2$ iff there are two locations $loc(o, \_, h, \mathcal{Q})$, $loc(o_2, \_, h_2, \mathcal{Q}_2) \in Locs$, two tasks $tsk(tk_1, m_1, l_1, y.\textbf{get}; s) \in \mathcal{Q}$ $tsk(tk_2, m_2, l_2, s_2) \in \mathcal{Q}_2$ and a future $fut(y, o_2, tk_2, m_2, \bot) \in Futs$.*
3. **Task-Location** : *$tk \rightarrow o$ iff there is a location $loc(o, tk_2, h, \mathcal{Q}) \in Locs$ and a task $tsk(tk, m, l, s) \in \mathcal{Q} s.t. tk_2 \neq tk$ and $fut(y, o, tk, pp(m), \bot) \in Futs$.*

The first type of dependency corresponds to the notion of blocking task and blocked location and the other two to waiting tasks. Dependencies are created as long as the task we are waiting for is not finished. Observe that a **get** instruction will generate two dependencies, whereas non-blocking synchronization instructions would generate only a dependency (*task-task*). Besides, every task without the location's lock (which is not finished) has a dependency to its location.

To prove equivalence between our approach and the one in [15], we define a new dependencies graph based on the information stored by the interleavings table and the set of future variables and in Theorem 1, we prove both graphs are equivalent.

Definition 5 (G_{IT}^{Futs})**.** *Given a state $S = (Locs, Futs, IT, \_)$ and an interleavings table IT, we define the interleavings graph G_{IT}^{Futs} as the tuple $<N, E>$ whose set of nodes N is defined as:*

$$N = \{o : fut(\_, o, \_, \_, \_) \in Futs\} \cup \{tk : fut(\_, \_, tk, \_, \_) \in Futs\}$$

and whose set of edges E is composed of:

1. **Location-Task**: $o \rightarrow tk_2$ *iff* $\exists\ t \in IT$ *such that*

$$t \equiv t_{o,tk,pp},\ \text{status}(t, IT) = pp_1{:}x.\textbf{get}\ \textit{and}\ fut(x, o_2, tk_2, pp_2, \bot) \in Futs$$

2. **Task-Task**: $tk \rightarrow tk_2$ *iff* $\exists\ t \in IT$ *such that*

$$t \equiv t_{o,tk,pp},\ \text{status}(t, IT) = pp_1{:}x.\textbf{get}\ \textit{and}\ fut(x, o_2, tk_2, pp_2, \bot) \in Futs$$

3. **Task-Location**: $tk \rightarrow o$ *iff*

$$\exists fut(x, o, tk, pp, \bot) \in Futs$$

Theorem 1. *Given a state $S{=}s(\_, Futs, IT, \_)$, then*

$$G_S \equiv G_{IT}^{Futs}$$

Proof. In order to see they are exactly the same graph, we need to check that (1) both have the same nodes and (2) both have the same edges. (1) is straightforward since nodes in G_S are all the location and task identifiers and nodes in G_{IT}^{Futs} are the location and task identifiers present in every future variable in *Futs*. But, there exists a future variable for each new task created (see rule ASYNC in Figure 2.1), so we have the same set of nodes in both graphs. Let us prove (2). We study every edge type on its own:

1. $o \rightarrow tk_2 \in G_S \Leftrightarrow$
   there are two locations $loc(o, tk, h, \mathcal{Q})$,
   $loc(o_2, \_, h_2, \mathcal{Q}_2) \in Locs$, two tasks $tsk(tk, m, l, y.\mathsf{get}; s) \in \mathcal{Q}$
   $tsk(tk_2, m_2, l_2, s_2) \in \mathcal{Q}_2$ and a future variable $fut(y, o_2, tk_2, m_2, \bot) \in Futs \Leftrightarrow$
   $\exists t{\equiv}t_{o,tk,pp} \in IT$ such that $status(t, IT) = pp_1{:}y.\textbf{get}$ and $fut(y, o_2, tk_2, m_2, \bot) \in Futs \Leftrightarrow$
   $o \rightarrow tk_2 \in G_{IT}^{Futs}$

2. $tk_1 \rightarrow tk_2 \in G_S \Leftrightarrow$
   there are two locations $loc(o, \_, h, \mathcal{Q})$,
   $loc(o_2, \_, h_2, \mathcal{Q}_2) \in Locs$, two tasks $tsk(tk_1, m_1, l_1, y.\mathsf{get}; s) \in \mathcal{Q}$
   $tsk(tk_2, m_2, l_2, s_2) \in \mathcal{Q}_2$ and a future variable $fut(y, o_2, tk_2, m_2, \bot) \in Futs \Leftrightarrow$
   $\exists t \equiv t_{o,tk_1,pp} \in IT$ such that $status(t, IT) = pp_1{:}y.\mathbf{get}$ and $fut(y, o_2, tk_2, m_2, \bot) \in Futs \Leftrightarrow$
   $o \rightarrow tk_2 \in G_{IT}^{Futs}$

3. $tk \rightarrow o \in G_S \Leftrightarrow$
   there is a location $loc(o, tk_2, h, \mathcal{Q}) \in Locs$ and a task $tsk(tk, m, l, s) \in \mathcal{Q}$ such that
   $tk_2 \neq tk$ and $fut(y, o, tk, pp(m), \bot) \in Futs \Leftrightarrow \exists fut(y, o, tk, pp, \bot) \in Futs \Leftrightarrow$
   $tk \rightarrow o \in G_{IT}^{Futs}$

□

Corollary 1. *Given a state $S = s(\_, Futs, IT, \_)$, G_S contains a cycle if and only if G_{IT}^{Futs} contains a cycle.*

Furthermore, the following lemma is used in Theorem 2. The proof can be found in [15].

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

As a consequence, G_S does not have cycles only composed of *task-task* edges and, by Theorem 1, G_{IT}^{Futs} does not have either. So, if there exists a cycle in G_{IT}^{Futs}, then it is composed of, at least, one location identifier.
Let us consider now a deadlock chain $dc(\{t_0, ..., t_n\}) \in S$, and let us see (1) how to build a path (a cycle) in G_{IT}^{Futs} using Definition 3 and (2) how to generate a deadlock chain from a cycle in G_{IT}^{Futs}.

Theorem 2. *Given a state $S = s(\_, Futs, IT, \_)$, there is a deadlock chain $dc(\{t_0, ..., t_n\}) \in S$ if and only if there is a cycle inside G_{IT}^{Futs}.*

Proof. We have to prove this double implication. First, we prove by induction that if there exists a deadlock chain in the state, then there is a cycle in G_{IT}^{Futs}. After that, we prove that if there is a cycle in G_{IT}^{Futs}, then we obtain a deadlock chain in the state. We proceed by induction on the length of the deadlock chain:

- Base case: Let $\{t_{o,tk,pp}\} \in S$ be a sequence of times of length 1, satisfying Conditions 1 or 2 in Definition 3. By Definition 2, $(t_{o,tk,pp}, t_{o_2,tk_2,\_}, \_) \in S$. Then, $status(t_{o,tk,pp}, IT) = \_ : x.\mathbf{get}$ and $fut(x, o_2, tk_2, \_, \bot) \in Futs$.
  Using this information, we can conclude there exist three edges $o \rightarrow tk_2$, $tk \rightarrow tk_2$ and $tk_2 \rightarrow o_2$. So, $o \rightarrow tk_2 \rightarrow o_2$ and $tk \rightarrow tk_2 \rightarrow o_2$ are paths of length 2 inside G_{IT}^{Futs}.

  Now, by Definition 3, if $t_{o,tk,pp}$ satisfies Condition 2, thus $loc(t_{o_2,tk_2,\_}) = (t_{o,tk,pp})$, that means, $o_2 = o$. Then, the path $o \rightarrow tk_2 \rightarrow o_2$ is equals to $o \rightarrow tk_2 \rightarrow o$ and, finally, we get a cycle inside G_{IT}^{Futs}.

- Induction Hypothesis (n): Let $\{t_0, ..., t_{n-1}\} \in S$ be a sequence of times satisfying Conditions 1 or 2 in Definition 3. Then, the sequence of times $\{t_0, ..t_{n-1}\}$ rises up a path inside G_{IT}^{Futs} and, if $dc(\{t_0, ..., t_{n-1}\}) \in S$, then this path is a cycle in G_{IT}^{Futs}.

- INDUCTIVE STEP (N+1): Let $\{t_{o_0,tk_0,pp_0}, ..., t_{o_n,tk_n,pp_n}\} \in S$ be a sequence of times of length $n+1$ satisfying Conditions 1 or 2 in Definition 3. By definition, $(t_{o_{n-1},tk_{n-1},pp_{n-1}}, t_{o_n,tk_n,\_}, \_) \in S$. Then, $status(t_{o_{n-1},tk_{n-1},pp}, IT) = \_{:}x.\textbf{get}$ and $fut(x, o_n, tk_n, \_, \bot) \in Futs$. Using this information, we can conclude there exist three edges $o_{n-1} \rightarrow tk_n$, $tk_{n-1} \rightarrow tk_n$ and $tk_n \rightarrow o_n$. So, by induction hypotesis over $\{t_{o_0,tk_0,pp_0}, ..., t_{o_{n-1},tk_{n-1},pp_{n-1}}\}$ we have that:

  - if $t_{o_{n-1},tk_{n-1},pp_{n-1}}$ holds 1 in Definition 3, we obtain a path $o_0 \rightarrow tk_1 \rightarrow .. \rightarrow tk_{n-1}$. Now, if we take in account the three aforementioned edges, we obtain the path $o_0 \rightarrow tk_1 \rightarrow .. \rightarrow tk_{n-1} \rightarrow tk_n \rightarrow o_n$ of length 2 inside G_{IT}^{Futs}.
  - if $t_{o_{n-1},tk_{n-1},pp_{n-1}}$ holds Condition 2 in Definition 3, we obtain a path $o_0 \rightarrow tk_1 \rightarrow .. \rightarrow o_{n-1}$. Now, if we take in account the three aforementioned edges, we obtain the path $o_0 \rightarrow tk_1 \rightarrow .. \rightarrow o_{n-1} \rightarrow tk_n \rightarrow o_n$ of length 2 inside G_{IT}^{Futs}.

  Let us consider t_{o_n,tk_n,pp_n}, satisfying Condition 2. By definition, there exists a blocking interval $(t_{o_n,tk_n,pp_n}, t_{o_{n+1},tk_{n+1},pp_{n+1}}, \_) \in S$, then there exist three edges $o_n \rightarrow tk_{n+1}$, $tk_n \rightarrow tk_{n+1}$ and $tk_{n+1} \rightarrow o_{n+1}$. But, $o_{n+1} = o_0$ (Condition 2 in Definition 3), then we get one of two cycles in G_{IT}^{Futs}: $o_0 \rightarrow tk_1 \rightarrow .. \rightarrow tk_{n-1} \rightarrow tk_n \rightarrow o_n \rightarrow tk_{n+1} \rightarrow o_0$ or $o_0 \rightarrow tk_1 \rightarrow .. \rightarrow o_{n-1} \rightarrow tk_n \rightarrow o_n \rightarrow tk_{n+1} \rightarrow o_0$.

Now, we prove that for each cycle inside G_{IT}^{Futs}, there exists a deadlock chain in S. First of all, by Lemma 1, we know that there cannot be cycles only composed of task identifiers. Thus, every cycle has the form $o_0 \rightarrow ... \rightarrow tk_n \rightarrow o_0$. Let us proceed again by induction on the cycle length.

- BASE CASE (3): Let $o_0 \rightarrow tk_1 \rightarrow o_1$ be a path inside G_{IT}^{Futs}. Then, by definition of the first edge, we get $\exists t_{o_0,tk_0,pp_0} \in IT$ such that $status(t_{o_0,tk_0,pp_0}) = pp_1{:}x.\textbf{get}$ and $fut(x, o_1, tk_1, pp_2, \bot) \in Futs$. But, we also have $\exists fut(y, o_1, tk_1, pp_3, \bot) \in Futs$ because of the second edge. Then, $\exists(t_{o_0,tk_0,pp_0}, t_{o_1,tk_1,pp_2}, t_{o_0,tk_0,pp_1}) \in S$. Now, if $o_0 = o_1$, then Condition 2 in Definition 3 holds trivially, thus $\exists dc(t_{o_0,tk_0,pp_0}) \in S$.

- INDUCTION HYPOTHESIS (N): Let $o_0 \rightarrow tk_1 \rightarrow ... \rightarrow tk_n \rightarrow o_n$ be a path in G_{IT}^{Futs}. Then, $\forall i \in [0, n-1], \exists(t_{o_i,tk_i,pp_i}, t'_{o_i+1}, \_) \in S$ satisfying Conditions 1 or 2 in Definition 3, and if $o_n = o_0$, then t_{o_n,tk_n,pp_n} satisfies Condition 2, that is, there exists a deadlock chain $dc(\{t_{o_0,tk_0,pp_0}, ..., t_{o_{n-1},tk_{n-1},pp_{n-1}}\}) \in S$.

- INDUCTIVE STEP (N+1): Let $o_0 \rightarrow tk_1 \rightarrow ... \rightarrow tk_{n+1} \rightarrow o_{n+1}$ be a path in G_{IT}^{Futs}. Then, the cycle has the form of one of the following: $o_0 \rightarrow tk_1 \rightarrow ... \rightarrow o_n \rightarrow tk_{n+1} \rightarrow o_{n+1}$ or $o_0 \rightarrow tk_1 \rightarrow ... \rightarrow tk_n \rightarrow tk_{n+1} \rightarrow o_{n+1}$.

  1. If it has the form $\mathbf{o_0} \rightarrow \mathbf{tk_1} \rightarrow ... \rightarrow \mathbf{o_n} \rightarrow tk_{n+1} \rightarrow o_{n+1}$, then we apply the induction hypothesis on the bold part and we obtain a set of times $\{t_{o_0,tk_0,pp_0}, ..., t_{o_n,tk_n,pp_n}\}$ which satisfy Conditions 1 or 2 in Definition 3, but because of the last bold node, we ensure that t_{o_n,tk_n,pp_n} satisfies Condition 2.
  2. If it has the form $\mathbf{o_0} \rightarrow \mathbf{tk_1} \rightarrow ... \rightarrow \mathbf{tk_n} \rightarrow tk_{n+1} \rightarrow o_{n+1}$, then we apply the induction hypothesis on the bold part and we obtain a set of times $\{t_{o_0,tk_0,pp_0}, ...,$

$t_{o_n,tk_n,pp_n}\}$ which satisfy Conditions 1 or 2 in Definition 3, but because of the last bold node, we ensure that t_{o_n,tk_n,pp_n} satisfies 1.

Now, by Definition of G_{IT}^{Futs}, both edges $o_n \rightarrow tk_{n+1}$ and $tk_n \rightarrow tk_{n+1}$ are defined as $\exists\ t_{n+1} \in IT$ such that:

$t_{n+1} \equiv t_{o_{n+1},tk_{n+1},pp_{n+1}},\ status(t, IT) = pp_{n+2}{:}x.\textbf{get}$ and $fut(x, o_{n+1}, tk_{n+1}, pp_{n+1}, \bot) \in Futs$

So $(t_{o_n,tk_n,pp_n}, t_{o_{n+1},tk_{n+1}}, pp_{n+1}, t_{o_n,tk_n,pp_{n+2}}) \in S$ satisfying Conditions 1 or 2. Now, if last edge is $tk_{n+1} \rightarrow o_0$ means that there exists $fut(y, o_0, tk_{n+1}, pp_{n+1}, \bot) \in Futs$. But $(t_{o_0,tk_0,pp_0}, t_{o_1,tk_1,pp_1}, \_) \in S$ is blocking, then t_{n+1} holds Condition 2 and we get a deadlock chain $dc(\{t_{o_0,tk_0,pp_0}, ..., t_{o_{n+1},tk_{n+1},pp_{n+1}}\})$.

□

Theorem 3 (Deadlock Equivalence)**.** *Let S be a program state,*

$$S \text{ is a deadlock state } \Leftrightarrow \exists\ dc(\{t_0, ..., t_n\}) \in S$$

Proof.

$$\begin{array}{rcl} \text{S is a deadlock state} & \Leftrightarrow & \exists\ cycle\ \gamma \in G_S \\ & \Leftrightarrow & \exists\ cycle\ \gamma \in G_{IT}^{Futs} \text{ (by Theorem 1)} \\ & \Leftrightarrow & \exists\ dc(\{t_0, ..., t_n\}) \in S \text{ (by Theorem 2)} \end{array}$$

□

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

4.3 Guiding Testing using Static Deadlock Analysis

This section proposes a deadlock detection methodology that combines static analysis and systematic testing as follows. First, a state-of-the-art deadlock analysis is run, in particular that of [15], which provides a set of abstractions of potential *deadlock cycles*. If the set is empty, then the program is deadlock-free. Otherwise, using the inferred set of deadlock cycles, we systematically test the program using a novel technique to guide the exploration towards paths that might lead to deadlock cycles. The goals of this process are: (1) finding concrete deadlock traces associated to the feasible cycles and (2) discarding unfeasible deadlock cycles, and in case all cycles are discarded, ensure deadlock freedom for the considered input. As our experiments show in Chapter 7, our technique reduces significantly the search space compared to the full systematic exploration.

4.3.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [15] returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} ... \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points, $tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions. Each arrow $e \xrightarrow{p:tk} e'$ should be interpreted like "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. The abstractions for tasks and locations can be performed at different levels of accuracy during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location o by the program point at which it is created o_{pp}, and each task by the method name executing. They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Points-to analysis is used as the basis to infer such abstractions. The analysis is object-sensitive, i.e., it distinguishes the actions performed by the different location abstractions. Both the analysis and the semantics can be made *object-sensitive* by keeping the k ancestor abstract locations (where k is a parameter of the analysis and any $k \geq 0$ can be used). For the sake of simplicity of the presentation, we assume $k = 0$ in the formalization (our implementation uses $k = 1$, e.g., an abstract task is of the form $o_{pp}.m$ where o_{pp} is the abstract location that executes method m).

Example 8. *In our working example in Figure 2.2,there are two abstract locations, o_7, correspoding to location* database *created at Line 7 and o_9, corresponding to the n locations* worker*, created inside the loop at Line 9; and four abstract tasks, register, getData, work, and ping. The following cycle is inferred by the deadlock analysis: $o_7 \xrightarrow{25:register} ping \xrightarrow{41:ping} o_9 \xrightarrow{38:work} getData \xrightarrow{29:getData} o_7$. The first arrow captures that the location created at Line 7 is blocked waiting for the termination of task* ping *because of the synchronization at Line 25 of task* register*. Also, a dependency between a task (e.g.,* ping*) and a location (e.g., o_9) captures that the task is trying to execute on that (possibly) blocked location. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can observed, it is complex to figure out from them why these dependencies arise, and more importantly the interleavings scheduled to lead to this situation.* □

4.3.2 Guiding Testing towards Deadlock Cycles

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

Definition 6 (Deadlock-cycle constraints)**.** *Given a state $S = s(\_, Futs, IT, \_)$, a deadlock-cycle constraint takes one of the following two forms:*

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

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

Definition 7 (Generation of deadlock-cycle constraints ϕ)**.** *Given an abstract deadlock cycle $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, and two fresh variables L_i, Tk_i, ϕ is defined as $\phi(e_i \xrightarrow{p_i:tk_i} e_j \xrightarrow{p_j:tk_j} \ldots, L_i, Tk_i) =$*

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

Uppercase letters appearing for the first time in the constraints are fresh variables. The first case handles location-task and task-task arrows (since e_j is a task abstraction), whereas the second case handles task-location arrows (e_j is an abstract location). Let us observe the following: (1) The abstract location and task identifiers of the abstract cycle are not used to produce the constraints. This is because constraints refer to concrete identifiers. Even if the cycle contains the same identifier on two different nodes or arrows, the corresponding variables in the constraints cannot be bound (i.e., we cannot use the same variables) since they could refer to different concrete identifiers. (2) The program points of the cycle (p_i and p_j) are used in time and *fut* constraints. (3) Location and task identifier variables of *fut* constraints and subsequent time or pending constraints are bound (i.e., the same variables are used). This is done using the second case of function ϕ. (4) In the second case, Tk_j is a fresh variable since the location executing Tk_i can be blocked due to a (possibly) different task. Intuitively, deadlock-cycle constraints characterize all possible deadlock chains representing the given cycle.

Example 9. *The following deadlock-cycle constraints are computed for the cycle in Example 8:*

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

They are shown in the order in which they are computed by ϕ. The first two constraints require IT to contain a concrete time in which some *database gets blocked while registering at Line* 25 *for a* certain *worker to receive the result of executing task* ping *at Line* 41*. The worker has not got it because of the value $\bot$ in the future. Furthermore, the last two constraints require a concrete time in which* this *worker waits at Line* 38 *to get the data stored by* some *database at Line* 29 *and the data is never returned. Note that, in order to*

preserve completeness, we are not binding the first and the second databases. If the example is generalized with several databases, there could be a deadlock in which a database waits for a worker which waits for another database and worker, so that the last one waits to get the data stored by the first database. This deadlock would not be found if the two databases are bound in the constraints (i.e., if we use the same variable name). In other words, we have to account for deadlocks which traverse the abstract cycle more than once. □

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

Definition 8. *Given a set of deadlock-cycle constraints $\mathfrak{C}$, and a state $S = s(Locs, Futs, IT, \_)$,* `check` *holds, written* $\texttt{check}_{\mathfrak{C}}(S)$*, if:*

$$\forall t_{L_i,Tk_i,PP} \mapsto \langle N, pp_i : F_i.\textbf{get}) \rangle \in \mathfrak{C}, fut(F_i, L_j, Tk_j, pp_j, \bot) \in \mathfrak{C}$$

one of the following conditions holds:

1. $\texttt{reachable}(t_{L_i,Tk_i,pp_i}, Locs)$ *or;*
2. $\exists t_{o_i,tk_i,pp} \mapsto \langle n, pp_i : f_i.\textbf{get} \rangle \in IT \wedge fut(f_i, o_j, tk_j, pp_j, \bot) \in Futs$

Function `reachable/2` checks whether a given task might arise in subsequent states. We over-approximate it syntactically by computing the transitive call relations from all tasks in the queues of all locations in S. Precision could be improved using more advanced analyses. Intuitively, $\texttt{check}_{\mathfrak{C}}\texttt{/1}$ does not hold if there is at least a time constraint so that: (i) its time identifier is not reachable, and (ii) in the case that the interleavings table contains entries matching it, for each one, there is an associated *fut* constraint which is violated, i.e., there is an associated future variable in the state where the associated task has finished (the return value is not equal to $\bot$). Condition (i) implies that there cannot be more representatives of the given abstract cycle in subsequent states, therefore if there are potential deadlock cycles, the associated time identifiers must be in the interleavings table. Condition (ii) implies that, for each potential cycle in the state, there is no deadlock chain since at least one of the blocking tasks has finished. This means there cannot be derivations from this state leading to the given cycle, hence the derivation can be stopped.

Definition 9 (Deadlock-cycle guided-testing (DCGT))**.** *Consider an abstract deadlock cycle c, and an initial state S_0. Let $\mathfrak{C} = \phi(c, L_{init}, Tk_{init})$ where L_{init} and Tk_{init} are fresh variables. We define the DCGT for S_0, written $exec_c(S_0)$, as the set $\{d : d \in exec(S_0), \texttt{deadlock}(S_n)\}$, where S_n is the last state in d.*

Example 10. *Let us consider the DCGT of our working example with the deadlock-cycle of Example 8, and hence with the constraints $\mathfrak{C}$ of Example 9. The interleavings table in the fifth state of the first derivation contains the entries:*

$$t_{0,main,2} \mapsto \langle 0, return \rangle,$$
$$t_{0,simulate,6} \mapsto \langle 1, return \rangle,$$
$$t_{db,register,22} \mapsto \langle 2, 25{:}f_0.\textbf{get} \rangle \text{ and}$$
$$t_{w,ping,41} \mapsto \langle 3, return \rangle.$$

Constraints $\{\exists t_{L_1,Tk_1,\_} \mapsto \langle \_, 25{:}F_1.\textbf{get}\rangle, \exists fut(F_1, L_2, Tk_2, 41, \bot)\}$ *are not satisfiable, (task ping has already finished at this point, as we can see in the interleavings table in Figure 4.1).* `check`$_{\mathfrak{C}}$`/1` *does not hold since* $t_{L_1,Tk_1,25}$ *is not reachable anymore and thus, the derivation is pruned because there is no deadlock state reachable from this state. Similarly, the rightmost derivation is stopped at its fifth state. Also, the third and fourth derivations are stopped by function* `deadlock/1` *of Theorem 3 (*early detection*). Since there are no more deadlock cycles, the search for deadlock detection finishes applying DCGT. Our methodology therefore explores* 9 *states instead of the* 25 *explored by the full systematic execution.* □

Theorem 4 (Soundness)**.** *Given a program P, a set of abstract cycles C in P, and an initial state* S_0, $\forall d \in exec(S_0)$ *if d is a derivation whose last state is deadlock, then* $\exists c \in C$ *such that* $d \in exec_c(S_0)$*.*

Proof of Theorem 4 relies on the soundness of the deadlock analysis that we state below.

Definition 10 (Deadlock soundness)**.** *Let S be a reachable state. If there is a cycle* $\gamma = e_1 \to e_2 \to \cdots \to e_1$ *in* G_S*, then* $\overline{\alpha}(\gamma) = \alpha(e_1) \xrightarrow{p_1:tk_1} \alpha(e_2) \xrightarrow{p_2:tk_2} \cdots \xrightarrow{p_n:tk_n} \alpha(e_1)$ *is an abstract cycle of* $\mathcal{G}$*.*

This theorem uses a standard abstraction function α that maps location and task identifiers to the corresponding abstract ones. The formal definition of α can be found in [4]. Let $\overline{\alpha}$ be the extension of α over the paths in G_S that applies the function α in every node contained by the path. Lemma 2 claims an auxiliary result which helps us to prove Theorem 4. It uses function γ, which transforms a sequence of times satisfying Conditions 1 or 2 from Definition 3 into its corresponding cycle in G_S (see proof of Theorem 2).

Lemma 2. *Given an initial state* S_0 *and an abstract cycle c,* $\forall d \in exec(S_0)$*,* $d{\equiv}S_0 \longrightarrow^* S_n$*, if* $\exists\ dc(\{t_0, ..., t_n\}) \in S_n$ *such that* $\overline{\alpha} \circ \gamma(\{t_0, ..., t_n\}) = c$*, then* $d \in exec_c(S_0)$*.*

Proof. By contradiction, let us suppose that $\exists d \in exec(S_0)$ and $d \notin exec_c(S_0)$. Hence, $\exists S_i \in d$ such that `check`$_{\mathfrak{C}}(S_i)$ returns false and consequently, the derivation $S_0 \longrightarrow^* S_i$ stops, where $\mathfrak{C} = \phi(c, L, Tk)$ and L, Tk are fresh variables.
Therefore, at S_i $\exists\{t_{L_i,Tk_i,PP} \mapsto \langle N, pp_i{:}F_i.\textbf{get}\rangle, fut(F_i, L_j, Tk_j, p_j)\} \subset \mathfrak{C}$ does not hold and neither 1 nor 2 in Definition 8. However, this cannot happen, as $\mathfrak{C}$ imposes necessary constraints for the existence of some representative of c and S_n contains a cycle that is a representative of c, then Condition 1 or 2 must be fulfilled in every state of d. As a result, we get a contradiction. □

Proof of Theorem 4. If the last state is deadlock, then, by Theorem 3, $\exists\, dc(\{t_0, ..., t_n\}) \in S_n$. Using the soundness of deadlock analysis on the cycle $\gamma(\{t_0, ..., t_n\})$, the existence of c is ensured. Now, by Lemma 2, we obtain the result. □

4.4 Deadlock-based Testing Criteria

In the application of testing for deadlock detection, and in a general setting where there could occur different potential deadlock cycles, the following practical questions arise: are

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

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

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

Chapter 5

Automatic (Deadlock-Guided) Generation of Distributed Contexts

In the context of static testing at the unit level, that is, the method level, the method under test is generally analyzed from a completely unknown context without any knowledge on its inputs. However, when lifting up to the integration or the system level, an initial (possibly partial) context is usually provided. This is specially relevant in the case of concurrent and distributed programs, where an initial context includes at least the set of locations and their initial tasks. For instance, in our motivating example, such an initial context is provided by the main method, which creates a DB 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 might be the case that it does not expose an error that occurs in other contexts.

A further challenge for our testing framework would be to make it able to automatically and systematically generate distributed contexts at the integration/system level. This would allow our framework to perform static testing, test case generation, and, in particular, deadlock detection, at the integration/system level. However, this would cause a further combinatorial explosion on the different possible distributed contexts that can be generated. Therefore, it is crucial to provide some support so that the process can focus on the *most interesting* contexts, filtering out other *less interesting* ones as soon as possible. This would include focusing on the most general contexts while other less general ones are discarded (since the derivations produced by the most general ones include those produced by the less general ones). E.g., the context generated by our main method should not be considered among the initial contexts of interest to do static testing, in the sense that it can be made more general just by leaving all locations fields uninstantiated.

For the particular context of deadlock detection, an additional challenge would be to only generate initial contexts in which a deadlock can occur. In the case of our motivating example, this would mean generating for instance a context with a database location and some worker location with a scheduled work task and a register task on the database for it, i.e., the context created by the main method. E.g., contexts that do noy include both tasks would be useless for deadlock detection. Interestingly, the deadlock analysis provides the possibly conflicting task interactions that can lead to deadlock. This information could be used to help our framework anticipate this information and discard initial distributed contexts that

cannot lead to deadlock from the beginning.

Section 5.1 introduces the concept of *initial context* and presents a method to automatically and systematically generate initial contexts. Then, Section 5.2 presents a deadlock-guided approach to effectively generate initial contexts for deadlock detection.

5.1 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 the initial tasks in each location queue (with free variables as parameters). 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, generating 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. E.g. 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 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, one could think on using some static analysis to determine a subset of tasks of interest according to some specific property. This is the case of our deadlock-guided approach of Section 5.2.

With all this, the input to our automatic generator of initial contexts is a set of abstract tasks $\mathcal{T}_{ini}$, each with its associated minimum and maximum cardinalities, i.e., a set of tuples $(\mathsf{C.M}, C^{min}, C^{max})$, where C and M are 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}. The goal of starting from this input is hence usefulness and flexibility.

Example 11. *Let us consider the set* $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.getData}, 0, 1)\}$. *The corresponding multisets are* $\{\mathsf{register}\}$ *and* $\{\mathsf{register}, \mathsf{getData}\}$. *All contexts must contain exactly one instance of task* register *and at most one instance of task* getData. *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* getData *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* getData*. E.g., the state corresponding to the latter context would be:*

$$\begin{array}{rcll} \mathtt{S} & = & [\mathtt{loc}(\mathtt{DB1},\mathtt{bot}, & [\mathtt{data}\mapsto \mathtt{D1},\mathtt{clients}\mapsto \mathtt{Cl1},\mathtt{checkOn}\mapsto \mathtt{B1}],\\ & & & [\mathtt{tsk}(1,\mathtt{register},[\mathtt{this}\mapsto \mathtt{r}(\mathtt{DB1}),\mathtt{m}\mapsto \mathtt{W1}],\mathtt{body}(\mathtt{register}))])\\ & & \mathtt{loc}(\mathtt{DB2},\mathtt{bot}, & [\mathtt{data}\mapsto \mathtt{D2},\mathtt{clients}\mapsto \mathtt{Cl2},\mathtt{checkOn}\mapsto \mathtt{B2}],\\ & & & [\mathtt{tsk}(2,\mathtt{getData},[\mathtt{this}\mapsto \mathtt{r}(\mathtt{DB2}),\mathtt{m}\mapsto \mathtt{W2}],\mathtt{body}(\mathtt{getData}))])], \end{array}$$

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.* getData*, 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. Let us recall 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, and a procedure (as a Prolog rule) that generates these contexts as partially instantiated states. 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 $i \neq j$. E.g., the three contexts in Example 11 are written as $\{[\mathsf{register}]_{\mathsf{db}_1}\}, \{[\mathsf{register}, \mathsf{getData}]_{\mathsf{db}_1}\}$ and $\{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{getData}]_{\mathsf{db}_2}\}$, 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 11 (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 a set of abstract tasks with associated cardinalities. Let* $o_{1,1}, \ldots, o_{1,C_1^{max}}, \ldots, o_{n,1}, \ldots, o_{n,C_n^{max}}$ *be* $\sum_{i=1}^{n} C_i^{max}$ *different identifiers: 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 12. *Let us consider the example* $\mathcal{T}_{ini}$=$\{(\mathsf{DB.register}, 0, 1), (\mathsf{DB.getData}, 1, 1)\}$. *The identifiers are* $o_{1,1}$ *and* $o_{2,1}$, *and the variables of the system are* $u^{reg}_{1,1}$, $u^{reg}_{2,1}$, $u^{get}_{1,1}$ *and* $u^{get}_{2,1}$. *Finally, we obtain the next system:*

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

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*

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

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

□

Let us observe that the two last contexts are equivalent since they are both composed of two instances of DB with tasks register and getData 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 12 (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 13. *The superset in Example 12 contains 3 equivalence classes induced by the relation* $\sim$*: (1) the class* $\{\{[\mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{getData}]_{\mathsf{o}_{2,1}}\}\}$, *where both contexts are composed of a location with a task* getData, *(2) the class* $\{\{[\mathsf{register}, \mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}, \mathsf{getData}]_{\mathsf{o}_{2,1}}\}\}$, *whose locations have two tasks* register *and* getData; *and, finally, (3) the class* $\{\{[\mathsf{register}]_{\mathsf{o}_{2,1}}, [\mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}]_{\mathsf{o}_{1,1}}, [\mathsf{getData}]_{\mathsf{o}_{2,1}}\}\}$, *where both contexts have two locations with a task* register *and a task* getData, *respectively.* □

Definition 13 (Minimal set of initial contexts $\mathcal{I}^{C_i}$ (same class Cl_i))**.** *Let* $\mathcal{T}_{ini}$ *be a 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 14. *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):*

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

□

```
121 generate_contexts([(M,MinC,MaxC)|Methods],SOut) :-
122   add_calls(SVar,[(M,0,MinC,MaxC)|Methods],SOut),
123   normal_form(SOut,N),
124   (prev_generated(N) -> fail ; assertz(prev_generated(N))).
125
126 add_calls(SIn,[(PP:M,Instances,MinC,MaxC)|Ms],SOut) :-
127   Instances < MaxC,
128   Call =.. [M,[ref(LocVar)|ArgsIn],ArgsOut],
129   add_task(SIn,LocVar,call,Call,PP,TaskId,S1),
130   fresh_fut(FutId),
131   add_future(S1,FutId,LocVar,TaskId,PP,S2),
132   I2 is Instances + 1,
133   add_calls(S2,[(PP:M,I2,MinC,MaxC)|Methods],SOut).
134
135 add_calls(SIn,[(_,I,Min,_),(M,MinC,MaxC)|Methods],SOut) :-
136   Min <= I,
137   add_calls(SIn,[(M,0,MinC,MaxC)|Methods],SOut).
138
139 add_calls(SIn,[(_,I,Min,_)],SIn) :-
140   Min <= I.
```

Figure 5.1: Prolog predicate to generate initial contexts

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 14 (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 $U : \mathcal{I}^{C_1} \times \ldots \times \mathcal{I}^{C_n} \rightarrow \mathcal{I}$, 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 15. *Let us consider the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.getData}, 1, 1),$ $(\mathsf{Worker.work}, 1, 1)\}$ from which we get the initial contexts $\mathcal{I}^{Worker} = \{\{[\mathsf{work}]_{\mathsf{w}_1}\}\}$ and $\mathcal{I}^{DB} = \{\{[\mathsf{register}, \mathsf{getData}]_{\mathsf{db},1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{getData}]_{\mathsf{db}_2}\}\}$. Then, by Definition 14,*

$$\mathcal{I} = \{\{[\mathsf{register}, \mathsf{getData}]_{\mathsf{db}_1}, [\mathsf{work}]_{\mathsf{w}_1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{getData}]_{\mathsf{db}_2}, [\mathsf{work}]_{\mathsf{w}_1}\}\}$$

□

We now define a Prolog predicate that generates the minimal set of initial contexts as partially instantiated states. Predicate `generate_contexts/2` in Figure 5.1 receives a set of abstract tasks with their associated maximum and minimum cardinalities, and generates on backtracking all generated initial contexts by means of `add_calls/3`. Predicate `normal_form/2` produces a normal form for the new context which is the same for all initial

contexts in the same equivalence class. The new context is therefore only generated if it has not been previously generated (i.e., if the call `prev_generated/1` fails). The first rule of `add_calls/3` checks if the number of instances `Instances` of task `PP:M` is smaller than the maximum cardinality `MaxC`, in which case we add a new instance of `M`, `Instances` is incremented, and `add_calls/3` is recursively invoked. The second rule checks if the number of instances is greater than or equal to `Min`, it initializes the number of instances for the next method (`M`) and makes the recursive call to `add_calls/3`. Finally, the third rule corresponds to the base case when we are processing the last method of the list and the number of instances if greater than or equal to `Min`.

Example 16. *Let us show predicate* `generate_contexts/2` *in action for the set* $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.makesConnection}, 1, 1), \mathsf{Worker.work}, 1, 1)\}$. *The first rule of* `add_calls/3` *is applied, as* `0 = Instances < MaxC = 1`. *Then,* `add_task/7` *is called with variable* `Locs` *and* `M = DB.register`*, at Line 129. As* `Locs` *is a variable, a new location is created. Once this predicate has finished,* `Instances` *is incremented and* `add_calls` *is recursively called (Line 133). Now, the second rule is applied, as* `0 = Min < Instances = 1`*, and* `add_calls` *is called with* `M = DB.makesConnection` *whose number of instances is initialized to 0 (Line 137). Again, at Line129,* `add_task/7` *is called with* `M = DB.makesConnection` *and* `Locs` *containing an instance of DB. Here we get to a branching point which gives rise to the two different initial contexts in Example 15. In the first branch,* `SIn` *contains a location whose class is equal to that of the method* `makesConnection`*, so* `LocVar` *is the existing location and a new instance is added to its queue. Finally,* `add_calls/3` *is called with* `M = Worker.work` *(Line 137), it creates a new instance of class Worker with a task* `work`*, and it finishes correctly at Line 140, and returns an initial context containing an instance of DB with tasks register and makesConnection, and an instance of Worker with task work. Now, it fails and the backtracking goes back to the branching point. Here, the third rule is applied and then, the first location is ignored and task* `makesConnection` *is added to a new location at Line 129. It finishes in a similar way. In this case, the initial context returned contains two instances of DB containing a task register and makesConnection, respectively, and an instance of Worker with task work.* □

5.2 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. In this section we propose to use this information to help our framework to discard initial contexts that cannot lead to deadlock from the beginning. The underlying idea is as follows: we select an abstract cycle detected

$$(\text{AWAIT1})\ \frac{tk = tsk(tk, m, l, \mathsf{x} = \textbf{await}\ y?; \mathsf{s}),\ l(y) = f,\ fut(f, \_, \_, \_, v) \in Futs}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \rightsquigarrow loc(o, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, \mathsf{s})\})}$$

$$(\text{AWAIT2})\ \frac{tk{=}tsk(tk, m, l, \mathsf{x} = \textbf{await}\ y?; \mathsf{s}),\ l(y) = f,\ fut(f, \_, \_, \_, \bot) \in Futs}{loc(o, tk, h, \mathcal{Q} \cup \{tk\}) \rightsquigarrow loc(o, \bot, h, \mathcal{Q} \cup \{tk\})}$$

Figure 5.2: Semantics of instruction **await**

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, such as, the **await** instruction. All the previous results are easily extended with this instruction.

In Figure 5.2, we can see the semantics of **await** instruction. The instruction **await** $y?$ waits for the future variable $\mathsf{y} \in Futs$ (where *Futs* is the future variable set in the state) yielding the lock (rule AWAIT2) and, once the task related to the future variable is finished, it can resume its execution (rule AWAIT1). When a location stops its execution due to an **await** instruction, the task yields the lock and another one can interleave its execution with it, i.e., start to execute and, thus, modify the location state (i.e., the location's *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 rewrite our working example to illustrate this problem and show how to solve it.

Figure 5.3 shows a modification of our working example. We have replaced field checkOn by connected and added a few lines to method register: field connected is bound to 0, then an asynchronous call is made to method empty and we wait until this task has been performed because of non-blocking instruction at Line 171 and then, field connected is checked to be greater than 0, finally it is decreased by one. We also added two new methods: empty that always returns 0 and makesConnection where connected is bound to the maximum number of connections allowed by the system. 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 171 is reached, the location's queue only contains task empty but no makesConnection and, therefore, at the moment task register is resumed, field connected keeps unchanged and the body of 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., 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 makesConnection.

```
141 class Main{
142   main(){
143     this ! simulate(1);
144     return 0;
145   }
146   simulate(int n){
147     DB db = new DB();
148     while (n > 0){
149       Worker w = new Worker();
150       db ! register(w);
151       w ! work(db);
152       n = n−1;
153     }
154     return 0;
155   }
156 }// end of class Main
157
158 class DB{
159   Data data = ...;
160   List<Worker> clients;// Empty list
161   int MAX_CONNECTS = 10;
162   int connected = MAX_CONNECTS;
163   int empty(){ return  0;}
164   int makesConnection(){
165     connected = MAX_CONNECTS;
166     return  connected;}
168   int register(Worker w){
169     connected = 0;
170     Fut<int>g = this! empty();
171     await g?;
172     if(0<connected){
173       connected = connected - 1;
174       Fut⟨int⟩ f = w ! ping(5);
175       if (f.get == 5) add(clients,w);
176     } else add(clients,w);
177     return 0;
178   }
179   Data getData(Worker w){
180     if (contains(w,clients)) return data;
181     else return null;
182   }
183 }// end of class DB
184
185 class Worker{
186   Data data;
187   int work(DB db){
188     Fut⟨Data⟩ f = db ! getData(this);
189     data = f.get;
190     return 0;
191   }
192   int ping(int n){return n;}
193 }// end of class Worker
```

Figure 5.3: Modified working example

Definition 15 (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:

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

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

```
194 calculate_interfering_tasks(Cycle,Tasks) :-
195   init(Cycle,[],Events,[],Ans),
196   process_events(Events,Ans,NoCardinality),
197   findall((Task,1,1),member((Task,_),NoCardinality),Repeated),
198   list_to_set(Repeated,Tasks).
199
200 init([],Evs,Evs,Ans,Ans).
201 init([edge(loc,get(Task,LAsync,LGet),task)|C],Evs,Evs2,Ans,Ans2) :-
202   !, init(C,[(Task,LAsync),(Task,LGet)|Evs],Evs2,[(Task,LGet)|Ans],Ans2).
203 init([edge(task,sync(Task,LAsync,LSync),task)|C],Evs,Evs2,Ans,Ans2) :-
204   !, init(C,[(Task,LAsync),(Task,LSync)|Evs],Evs2,Ans,Ans2).
205 init([_|Cycle],Evs,Evs2,Ans,Ans2) :- init(Cycle,Evs,Evs2,Ans,Ans2).
206
207 process_events([],Ans,Ans).
208 process_events([(Task,Inst)|Evs],Ans,Ans2) :-
209   thereis_await(Task,Inst),
210   accessed_fields(Task,Inst,Fields), !,
211   findall((T,L),(member(F,Fields),
212                  inst(F,write,T,L),
213                  \+ member((T,L),Ans)),Modifiers),
214   append(Modifiers,Evs,Evs2), append(Modifiers,Ans,Ans1),
215   process_events(Evs2,Ans1,Ans2).
216 process_events([_|Evs],Ans,Ans2) :- process_events(Evs,Ans,Ans2).
```

Figure 5.4: Prolog predicate to infer interfering tasks for a given deadlock cycle

the set of instructions that modify field f. We can observe that *initialTasks(C)* is the union of 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 infinite depending on the program we are working with. For instance, if we apply the definition to the abstract cycle C in Example 8, $initialTasks(\mathsf{db.register}, 173, C)$ will be evaluated. It fits well with the conditions on third clause, as there exists an **await** instruction, $\mathsf{fields(173)} = \{\mathsf{connected}\}$ and then again 173 is a modifier instruction of field connected, so $initialTasks(\mathsf{db.register}, 173, C)$ will be evaluated again recursively.

Figure 5.4 presents predicate `calculate_interfering_tasks/2` that finitely infers the interfering-tasks for a given deadlock cycle as defined by Definition 15. First, both the list of events and of answers are initialized (`init/5`) according to the type of edge. For each edge

in the cycle, we take the call and the corresponding synchronization instruction, and we add them to the pending events. Moreover, **get** instructions produce *location-task* edges, so they are also included in the answers list, 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 those instructions. In predicate `process_events/3`, we take a pending event `(Task,Inst)` and we check if there is an **await** instruction between the start of `Task` and `Inst`, using predicate (`thereis_await/2`), where the previously accessed field values (`accessed_fields/3`) could be changed (third clause in Definition 15. In case it does, we need to include in the answer set all tasks which contain instructions modifying such field (`inst/4`). Besides, 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 instructions to the pending events list. This predicate finishes when this list is empty and `Ans` is the list of pairs with all interfering instructions and their container tasks. Finally, we only take the tasks, i.e., the first component of each pair, we set their minimum and maximum cardinalities and remove duplicates (`list_to_set/2`). Finiteness is guaranteed because each instruction is added to the pending events and answers lists at most once, and the number of instructions is finite.

Example 17. *Let us show how predicate* `calculate_interfering_tasks/2` *works for our modified example. For the sake of clarity, instructions are identified by their line numbers. After the* `init/5` *predicate, the value of variables* `Events` *and* `Ans` *is* [(Worker.work, 189), (Worker.work, 188), (DB.register, 175), (DB.register, 174)] *and* [(DB.register, 175), (Worker.work, 189)]*, respectively. Hence, predicate* `process_events/3` *takes* (Worker.work, 189) *first. Since there is not an* **await** *instruction between the beginning of work and Line 189,* `Ans` *stays unchanged. The same happens with* (Worker.work, 188)*. Now, the pending events list is* [(DB.register, 175), (DB.register, 174,)] *and* (DB.register, 175) *is processed. Now, there is an* **await** *between Lines 168 and 175 and, then,* `fields(DB.register,175,Fields)` *is invoked and* `Fields`=[connected]*. We find three instructions modifying the field connected: 165 ∈ DB.makesConnection, 169 ∈ DB.register and 173 ∈ DB.register. None of them is a member of the answer set and hence they are added to both lists. Now,* `Evs` *is [(DB.register,173), (DB.makesConnection,165), (DB.register,174),(DB.register,169)] but again there is no* **await** *between the beginning of tasks* DB.register *and* DB.makesConnection *and Lines 173 and 165, respectively and, thus,* `Ans` *stays unchanged. Finally, both* (DB.register, 169) *and* (DB.register, 174) *are taken and both* `fields(DB.register,174,Fields)` *and* `fields(DB.register,169,Fields)` *hold where* `Fields`=[connected]*, but the modifier instructions have been previously added to* `Ans`*, hence* `Ans` *remains unchanged, and the pending events list becomes empty. Finally, the algorithm projects over the first component of each pair in the list, sets the minimum and maximum cardinalities to* 1 *and removes duplicates, returning the set* $\mathcal{T}_{ini} = \{$*(DB.register,1,1), (Worker.work,1,1), (DB.makesConnection,1,1)*$\}$*. Thus, the generation of initial contexts for this set (see Example 16) produces*

$$\begin{aligned}\mathcal{I} \;=\; & \{\{[\mathsf{register},\mathsf{makesConnection}]_{\mathsf{db}_1}[\mathsf{work}]_{\mathsf{w}_1}\},\\ & \{[\mathsf{register}]_{\mathsf{db}_1},[\mathsf{makesConnection}]_{\mathsf{db}_2},[\mathsf{work}]_{\mathsf{w}_1}\}\}\end{aligned}$$

□

Chapter 6

Implementation

We have implemented all the presented techniques within the prototype tool SYCO/`aPET` [10, 8], a dynamic/static testing tool for the ABS *concurrent objects* language [23], which includes the POR techniques to detect and avoid the redundant executions described in [6, 7]. The tool is available for online use through a user-friendly web interface at `http://costa.ls.fi.upm.es/syco`, where the code of our running examples and the benchmarks used in Section 7 can also be found. ABS concurrent objects communicate via *asynchronous* method calls and use **await** and **get**, resp., as instructions for non-blocking and blocking synchronization. Handling non-blocking synchronization in our framework do not pose any technical complication and has not been included in our formalization for the sake of simplicity. All the components of our implementation however include support for it.

The rest of the chapter is organized as follows: Section 6.1 gives an overview of the SYCO/`aPET` tool. Then, Sections 6.2 and 6.3 present detailed step-by-step separate tutorials of the SYCO and `aPET` sub-tools.

6.1 General Overview

6.1.1 The SYCO Tool

SYCO is a systematic tester for `ABS` concurrent objects. Figure 6.1 shows its main architecture. Boxes with dash lines are internal components of SYCO whereas boxes with regular lines are external components. The user interacts with SYCO through its web interface which has been built using the *EasyInterface* [13] framework. The SYCO engine receives an `ABS` program and a selection of parameters. The `ABS` *compiler* compiles the program into an abstract-syntax-tree (AST) which is then transformed into the SYCO intermediate representation (IR), i.e., the CLP-transformed program. The DPOR *engine* carries out the actual systematic testing process in CLP. It comprises the `ABS` semantics in CLP, the DPOR algorithm of [6] and the *stability* and *dependencies* analyses of [6]. The *output manager* then generates the output in the format which is required by EasyInterface, including an XML file containing all the EasyInterface commands and actions and SVG diagrams. In case deadlock-guided testing is applied, the DECO *deadlock analyzer* [15] is invoked, which returns a set of potential deadlock cycles that are then fed to the DPOR engine to guide the testing process (discarding non-deadlock executions).

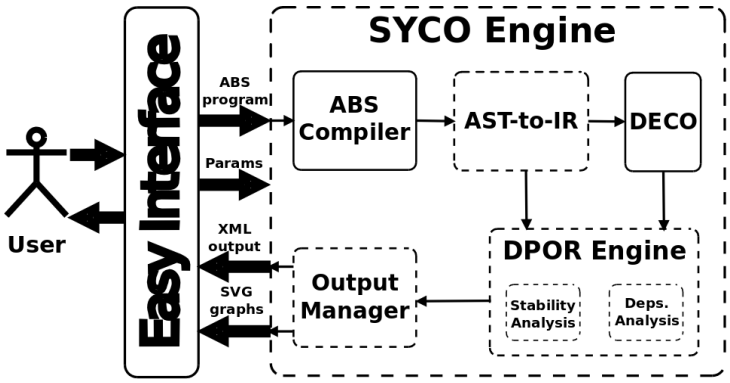

Figure 6.1: SYCO architecture

Section 6.3 details the usage of SYCO. Essentially, once the input program is ready, either selected from the available library of ABS programs or supplied by the user, the SYCO engine is run (with the selected settings) and the output is obtained. As a result, SYCO outputs a set of execution results. For each one, SYCO shows the output state and the sequence of tasks/interleavings and concrete instructions exercised by the corresponding execution (highlighting the source code). SYCO also generates sequence diagrams for each execution. Such sequence diagrams provide graphical and more comprehensive representations of execution traces. Essentially, they show the task/object executing at each time of the simulation, the spawned asynchronous calls (with arrows from caller to callee), and, the waiting and blocking dependencies. See Section 6.2.2 for details.

6.1.2 The aPET Tool

aPET is a static testing tool and test case generator for ABS concurrent objects based on *symbolic execution*. Its architecture is essentially the same as that of SYCO. Indeed, both tools share most of their components, namely the ABS compiler, the AST-to-IR and part of the DPOR Engine and Output Manager. The main differences are that the internal CLP engine of aPET includes support for symbolic execution and its termination criteria, and that the output manager includes support for TCG in different formats.

The usage of aPET is essentially as follows: given an input program and a selection of methods, the aPET symbolic execution engine computes a set of test cases for the selected methods. Test cases can be given as path constraints or, after a constraint solving procedure, as concrete test cases. Each test case includes the input arguments and input state, and the output argument and output state. Section 6.3 details how to use aPET with screenshots and provides information about the different parameters which can be set.

6.2 SYCO: Step by Step

In order to use *SYCO* we select *Systematic testing (SYCO)* from the pull-down menu (see Figure 6.2). example is available here. It is a bit simple than the one presented in Section 2.1, we remove method simulate and clients field becomes a single reference. It can be seen

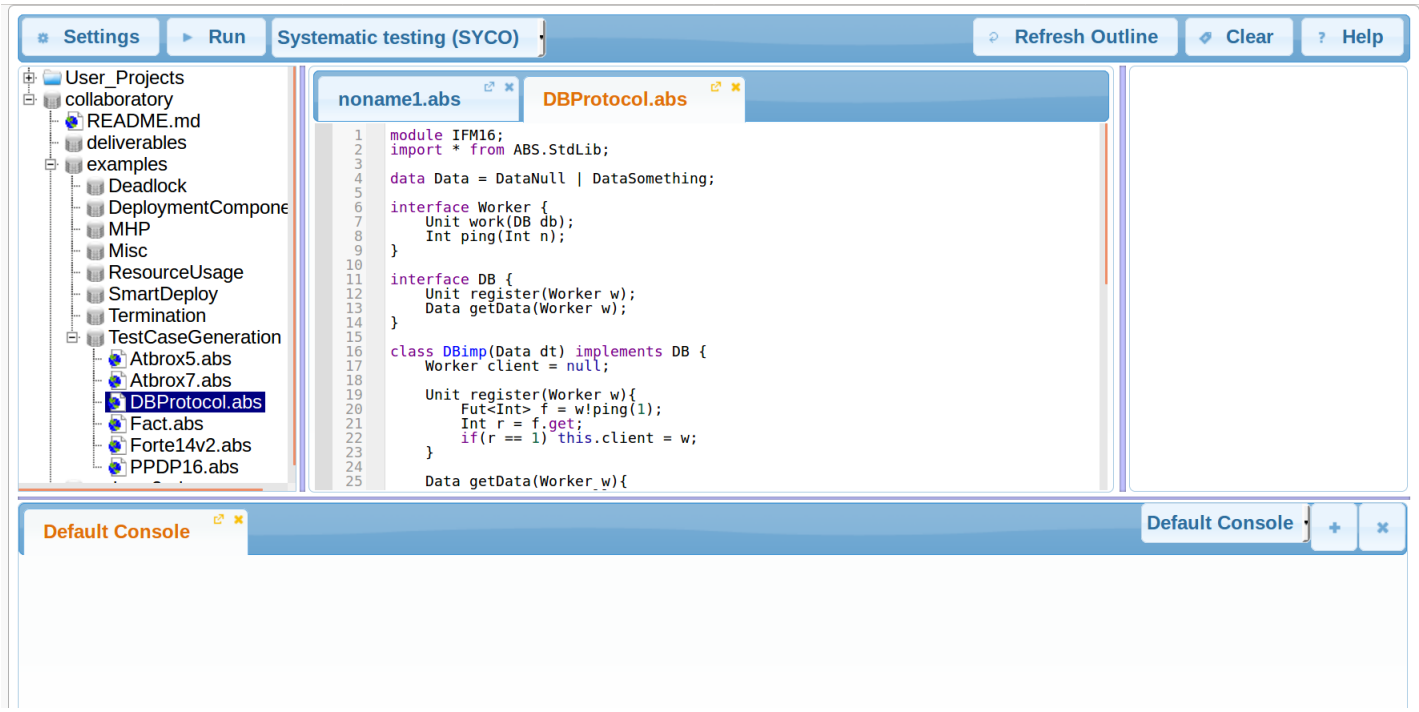

Figure 6.2: SYCO/aPET web interface

in Figures 6.2 and 6.3. If we click over `DBProtocol.abs`, the code appears at the code area. Now, if we press button `Refresh Outline`, the right-hand side with the classes and module information is updated. The `Clear` button cleans the console area. Optionally, the parameters of the selected testing tool can be configured by clicking on `Settings` (details are given in Section 6.2.3). To execute the selected tool it is enough to click `Apply` in the pulldown menu on the tool bar and the results are presented in the console area.

6.2.1 Using SYCO with default parameters

Let us perform a systematic testing of our running example with SYCO using default parameters. We just select SYCO and press `Apply`. Note that systematic testing always targets the `main` block. Therefore, the selection made in the outline view is ignored. The results are printed in the console area.

SYCO first prints the number of complete executions explored (in this case 4 executions). Note that, by default, an aggressive POR is applied. As we will see later, the number of executions without POR is 6. Also, the most recent POR technique included in SYCO is able to obtain just two executions. SYCO then prints the output state and the execution trace. The output state (in blue color) contains all the objects created during the execution. Each object is represented as a term with three arguments: the object identifier, the object type or class, and the final values of the object fields. For instance:

```
- State
|------object(1,'DBimp',[field(dt,'DataSomething'),field(client,ref(2))])
|------object(2,'WorkerImp',[field(datum,'DataNull')])
|------object(main,main,[])
```

means that the final state contains: (1) an object identified by 1 of class `DBimp`, whose fields `dt` and `client` have '`DataSomething`' and `ref`(2) as values; (2) an object identified by 2 of class `WorkerImp`, whose fields `datum` has '`DataNull`' as value and (3) an object identified by `main` that creates the previous objects. Since we have registered the worker as DB's client, its `datum` field should end with value '*DataSomething*', instead of the obtained '`DataNull`'. This execution therefore reveals a bug in the program.

The execution trace (in red color) shows, for each time or macro-step of the execution, the object and task executing at this time. If we click one time of the trace, the corresponding line in the source code is highlighted (in yellow color) in the code area. This is shown in Figure 6.3 where the first time (`|------'Time: 0, Object: main, Task: 0:main'`) of the trace has been clicked.

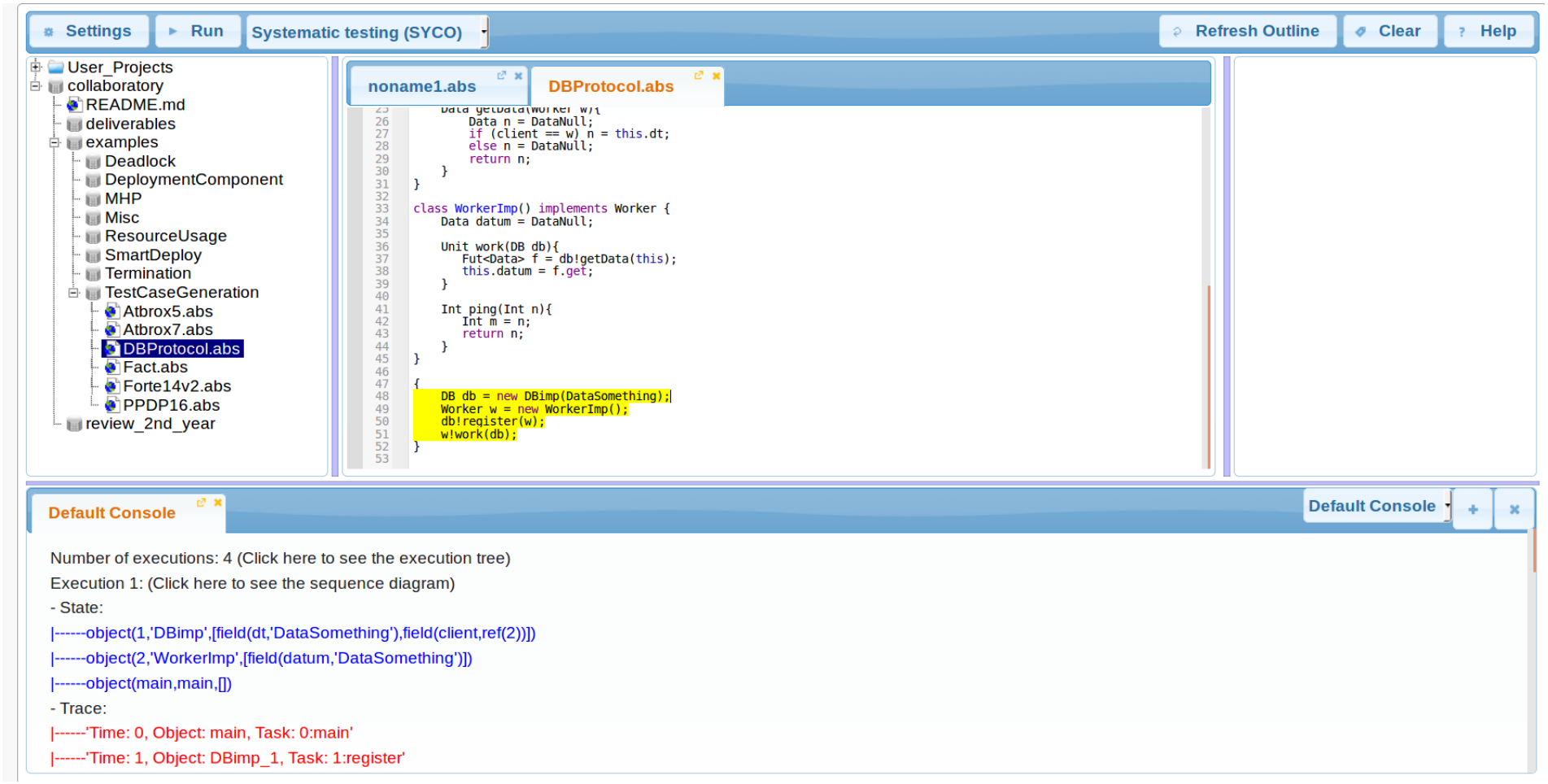

Figure 6.3: Execution of SYCO with default parameters

6.2.2 How to understand the sequence diagrams

To see the sequence diagram of a concrete execution we click the text ‘‘`Click here to see the sequence diagram`’’ (next to the execution number in the console view). Figure 6.4 shows the sequence diagram of the first execution for our running example. At the left-hand side, a timeline is shown with the times of the execution, in this case 7 times $(0-6)$. Each vertical cluster corresponds to the activities performed by each object, and each node corresponds to the task executing at the corresponding object in the corresponding time. Objects are of the form `class_id`, where `class` is the object type and `id` is a unique object identifier. Tasks are of the form `id:method` where `id` is a unique task identifier and `method` is the name of the method. Nodes also indicate why the execution of the associated task stopped. Nodes in green color labeled with `return` correspond to tasks that have finished their executions; nodes in orange color labeled with `waiting for taskId` are tasks which have been suspended waiting for task `taskId`; and nodes in red color labeled with `blocked for taskId` are tasks which block the object waiting for task `taskId`. Finally, arrows from nodes to clusters indicate asynchronous calls or object creations.

In our running example, the trace corresponding to execution 1 is shown in Figure 6.4. Let us briefly explain the diagram and the relations among the diagram, the code of the program (Figure 2.2) and the final state computed for execution 1 (Figure 6.3 below). Time 0 corresponds to the execution of the `main` block within the object identified as `main_0`. It creates two new objects `DBimp_1` (DB DB = **new** DBimp(DataSomething)) and `WorkerImp_2` (Worker w = **new** WorkerImp()). Then, it spawns task `1:register` (`db ! register(w)`) and task `3:work` (`w ! work(db)`). These calls produces in the sequence diagram two arrows tagged with `1:register` and `3:work`, respectively. In the final state, this adds the objects `object(main,main,[])` , `object(1,'DBimp',[field(dt,'DataSomething'), field(client,null)])` and `object(2,'WorkerImp',[field(datum,null)])` respectively.

Then, the block `main_0` finishes its execution and it is marked with `return`. During time 1, object `DBimp_1` executes the task `1:register` which spawns task `2:ping` (an arrow between `DBimp_1` and `WorkerImp_2` appears tagged with `2:ping`) to check if the worker is online and it gets blocked until such task is performed, then, this node is red and it is marked with `blocked for 2`. At time 2, `2:ping` is executed and it finishes in a return, its node is green color. At time 3, the execution of task `3:work` makes a new call (and produces a new arrow between `WorkerImp_2` and `DBimp_1`) (w ! getData()) and it gets blocked until its execution (red node tagged with `blocked for 4`). During time 4, `DBimp_1` resumes the execution of task `1:register` at Line 21 (tagged as `1:register(21)` and finishes correctly. Then, `DBimp_1` becomes `object(1,'DBimp',[field(dt,'DataSomething'),field(client,ref(2))])`. At time 5, `DBimp_1` executes task `4:getData` and it returns `DataSomething`, as `WorkerImp_2` has been previously registered as its client. Finally, during time 6, `WorkerImp_2` resumes the execution of task `3:work` at Line 38 and it finishes correctly, in a green node. `WorkerImp_2` becomes `object(2,'WorkerImp',[field(datum,'DataSomething')])`.

Figures 6.5 and 6.6 show the result and sequence diagram of the third execution, in which we can observe the aforementioned bug. If we make a comparison between the sequence diagrams of executions 1 and 3, we can figure out that the problem in execution 3 originates on time 2, where task `4:getData` finishes before executing task `1:register`, that is, before

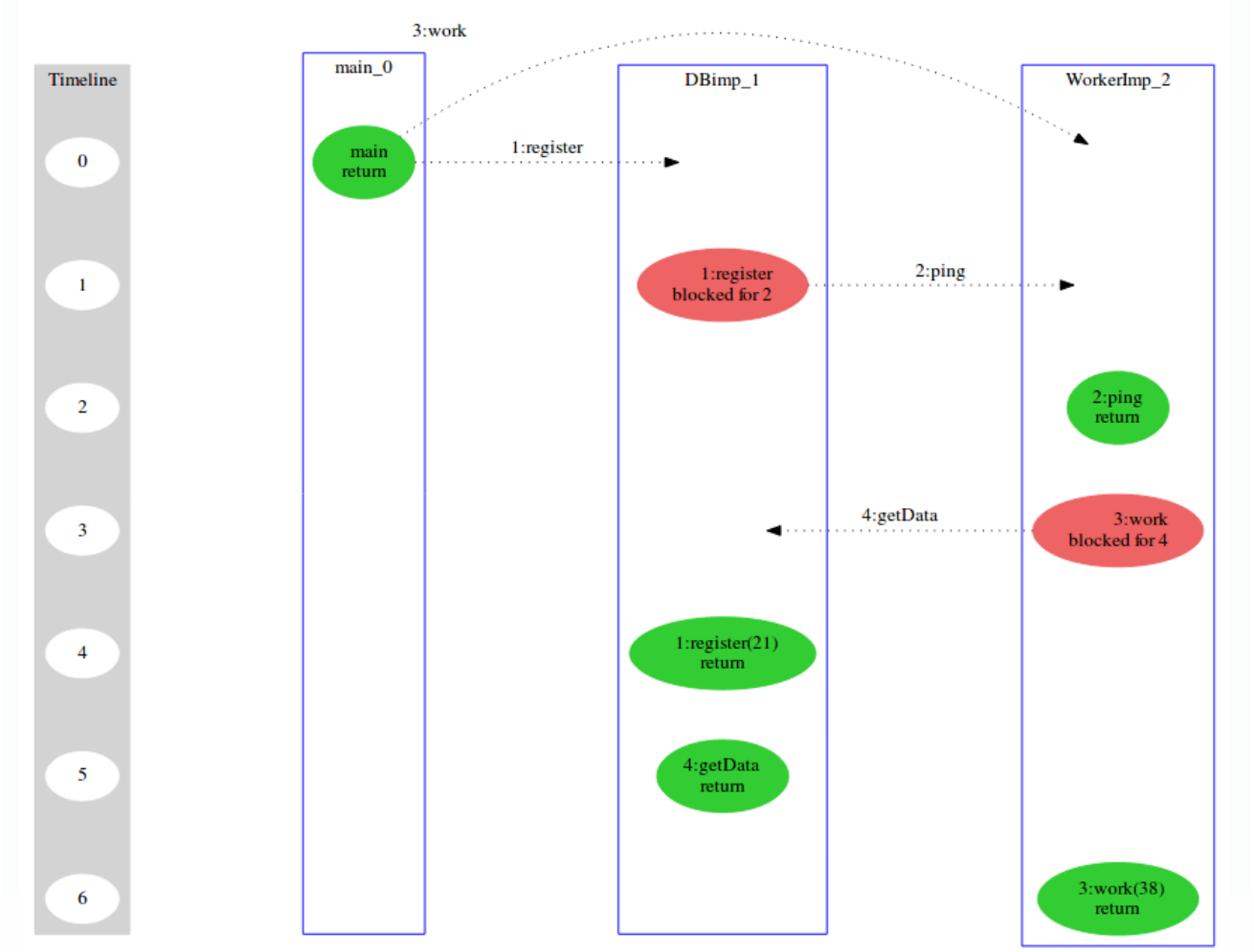

Figure 6.4: A buggy execution trace for the running example

`DBimp_1` had registered `WorkerImp_2` as its client. In the sequence diagram of execution 1 we can observe that object `DBimp_1` executes task `4:getData` after executing task `1 : register` (see times 1 and 15).

6.2.3 Parameters of SYCO

Up to now we have executed SYCO with default parameters. Pressing button `Settings` at the toolbar shows the parameters window, which allows to configure the available parameters for each application. Figure 6.7 shows the parameters of SYCO, with default values. The following parameters can be set:

- *Object selection policy.* By default all objects from a state are selected non-deterministically on backtracking (option `Non-deterministic`). In case parameter *Partial-order reduction* below is enabled, only the required objects are selected according to the POR theory (see [6]). The other value `Round-robin` selects an object deterministically using a round-robin strategy.

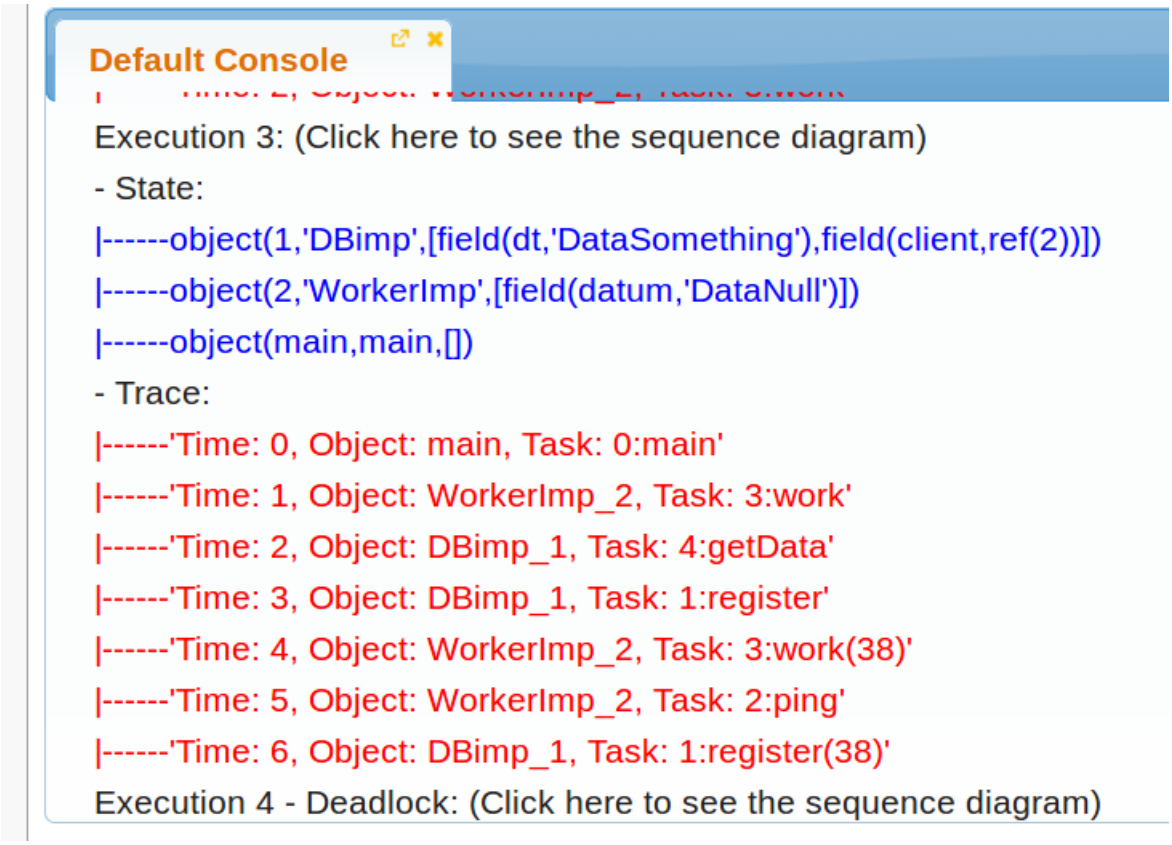

Figure 6.5: A correct execution for the running example

- *Task scheduling policy.* It allows us to set the scheduling policy of objects. Available values are `FIFO`, `LIFO` and `Non-deterministic`. The default value is `Non-deterministic`. Otherwise, SYCO performs a deterministic simulation with the selected strategies.

- *Partial-order reduction.* It allows one to disable POR, by selecting value `None`, or to enable it with one of the following three levels of precision, `Naive dep. approx.`, `Shared memory dep.` (by default) and `Exact dep.`. Option `Naive dep. approx.` only applies the POR object selection in [6] based on stability, whereas option `Shared memory dep.` over-approximates the dependencies based on shared-memory accesses of [6]. Finally, `Exact dep.` applies a recent and yet experimental DPOR technique which detects dynamically context-sensitive and exact dependencies. In the example, 4 executions are obtained with POR based on `shared-memory dependencies`, and 8 if POR is disabled. Using the technique to detect exact dependencies we just get 2 executions. This examples is quite small, but bigger examples illustrate the effectiveness of the available POR techniques.

- *Deadlock-guided testing.* It allows us to enable/disable deadlock-guided testing. By default it is disabled. If it is enabled, the testing process is guided towards deadlocks, discarding non-deadlock executions, with the corresponding state space reduction. This is useful in the context of deadlock detection and debugging. See Section 6.2.5 above.

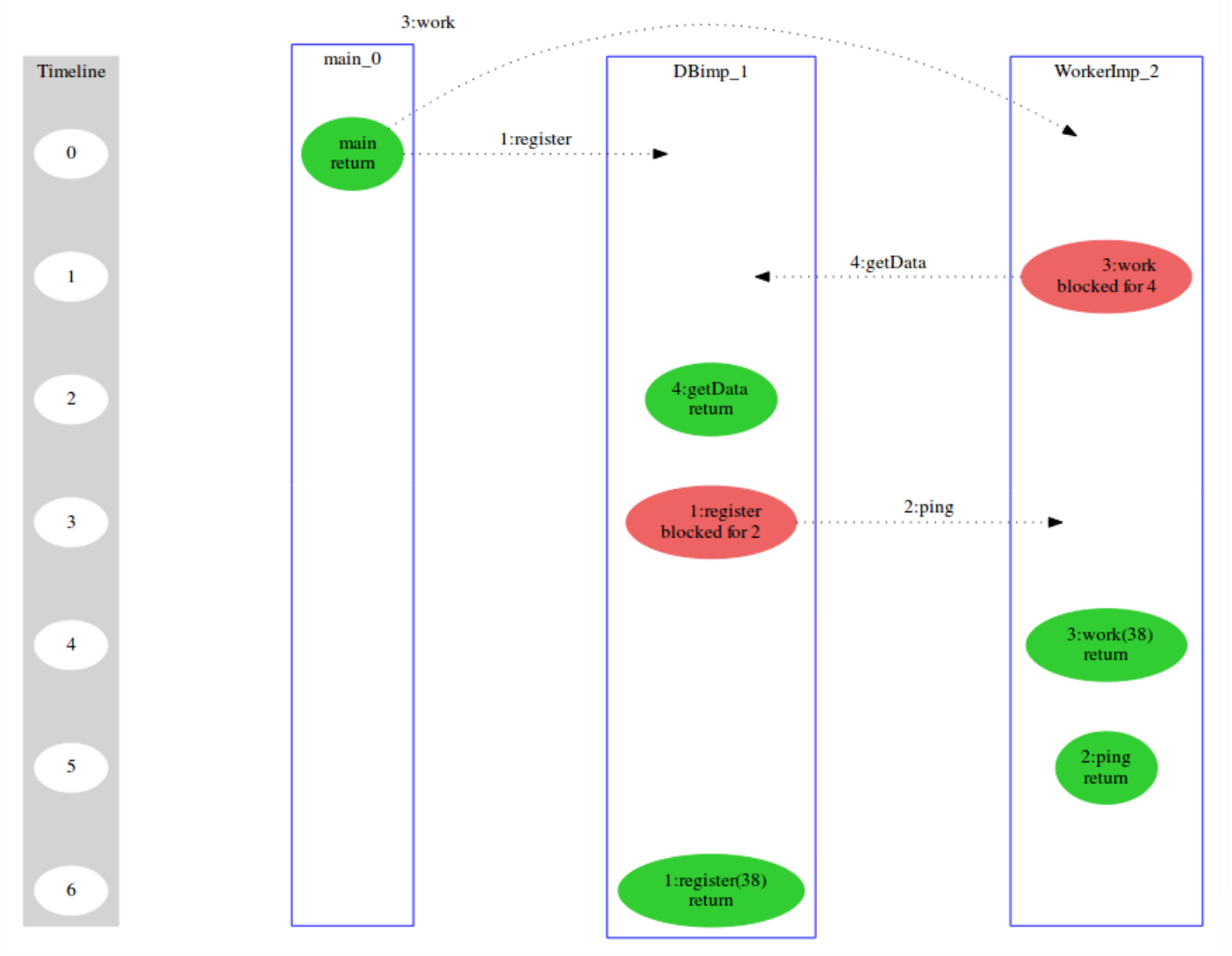

Figure 6.6: Diagram of correct execution for the running example

- *Global timeout.* Measured in seconds. It allows us to stablish a maximum time for the execution of SYCO.

6.2.4 How to understand the execution tree

To see the execution tree of the program we click over the text ‘‘`Click here to see the execution tree`’’ at the first line in the console view. Figure 6.8 shows the execution tree of the program `DBProtocol.abs` (with option `POR` disabled).

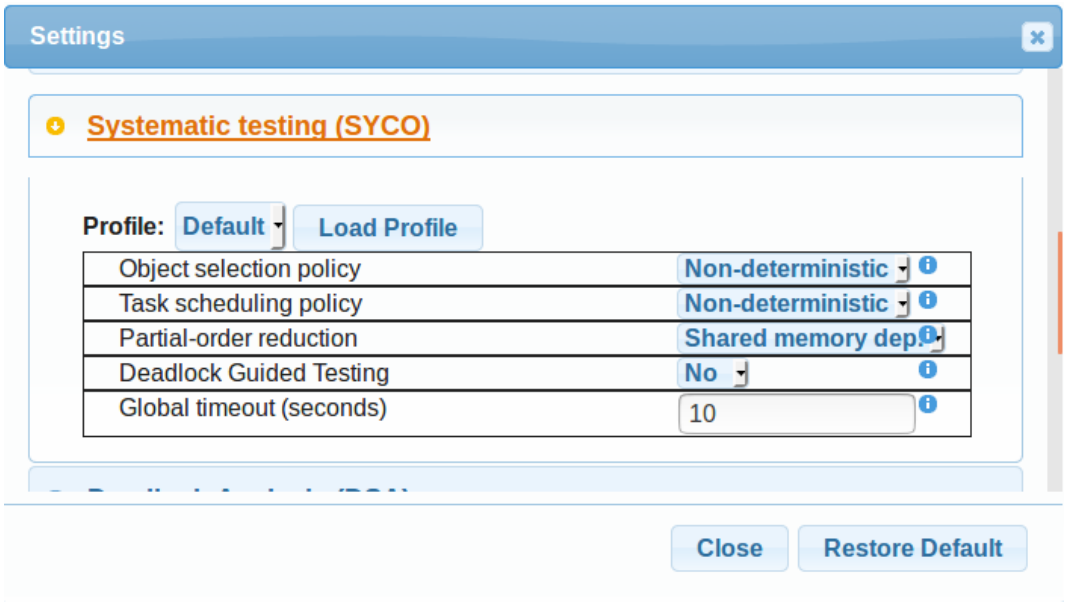

Figure 6.7: The *SYCO* parameters

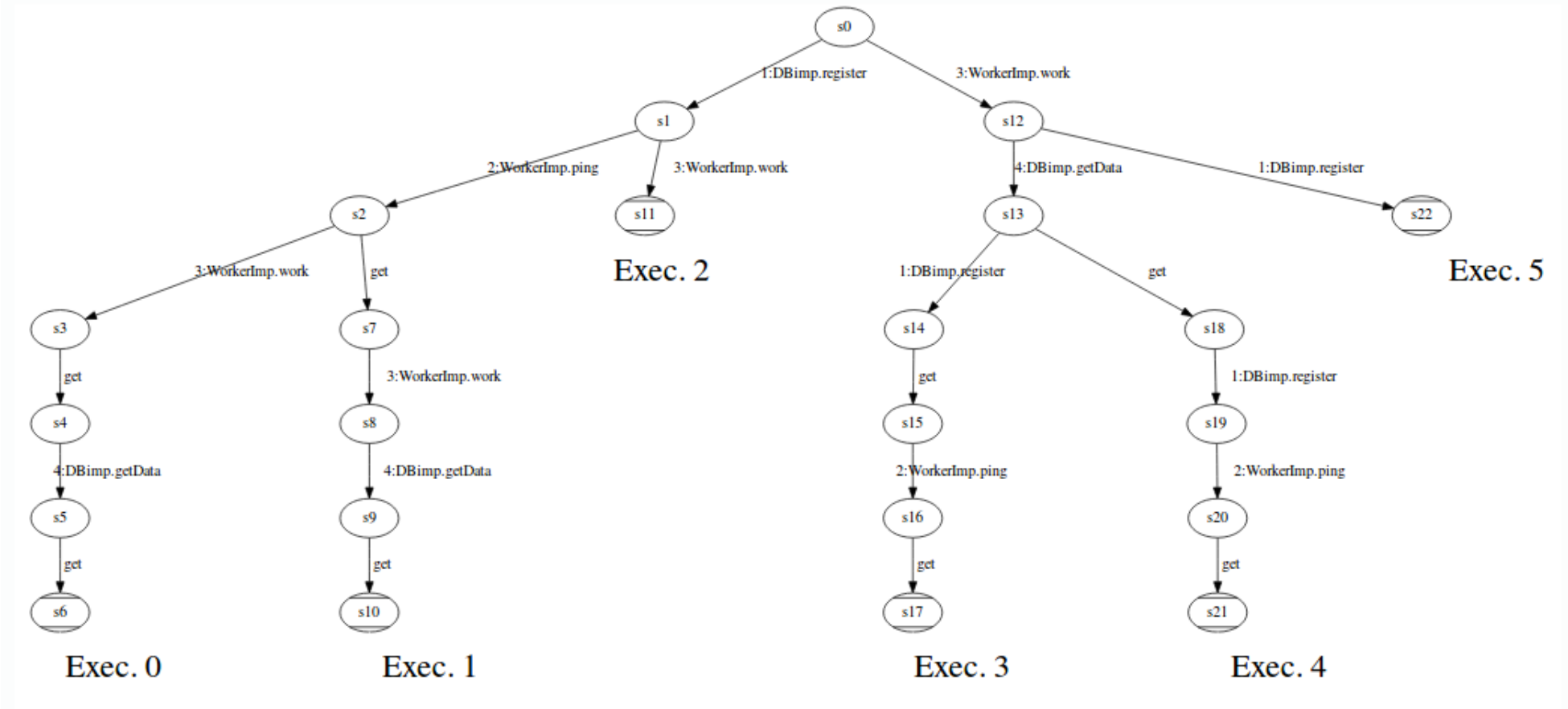

Figure 6.8: Execution tree of `DBProtocol.abs`

The root of the execution tree is the state `s0`. After executing method `main`, the execution can either proceed with task `1:register` of object `DBimp_1` or task`3:work` of object `WorkerImp_2`. If the first one is executed, we reach the state `s1`, where the execution can either proceed with task `2:ping` or `3:work` of object `WorkerImp_2`. If `2:ping` is executed, then executions 0 and 1 are deadlock-free (they would have been pruned if deadlock-guided testing is enabled as we show below). However, if `3:work` is executed, we get a deadlock. The other part of the tree is analogous: executions 3 and 4 are deadlock-free (and would be pruned in deadlock-guided testing) and execution 5 produces a deadlock.

6.2.5 Deadlock-guided testing with SYCO

As we have seen in the previous sections, with POR disabled, *SYCO* produces 6 executions for the working example, which cover all possible task interleavings that may occur. *SYCO* reports that two executions are deadlock executions corresponding to sequences main → register → work and main → work → register. If we enable `Deadlock-guided testing`, we obtain just the two deadlock executions which are shown in Figure 6.9. Looking at the sequence diagram of the first execution (Figure 6.10 up), we can observe a deadlock situation, since both `DBimp_1` and `WorkerImp_2` are blocked and, as we can see, they are squared in red color. During time 1, `DBimp_1` gets blocked waiting for `WorkerImp_2` to execute task `4:ping`. During the next time, object `WorkerImp_2`, instead of executing task `4:ping`, it executes task `5:work`, getting blocked waiting for `DBimp_1` to execute `6:getData`. Therefore, none of the objects can make any progress. Both tasks are highlighted with red solid edges to indicate that these are the ones responsible for the deadlock. The second execution (see Figure 6.10 down) is similar but changing the execution order between tasks `3:register` and `5:work`.

6.3 aPET: Step by Step

This section illustrates the usage of `aPET` using our running example. In this case we select `Test case generation (aPET)` from the pull down menu in the toolbar. In contrast to SYCO, since `aPET` performs symbolic execution, it can be applied over any method, possibly containing input arguments. Symbolic execution produces as a result the conditions over the input arguments and input state, or directly concrete values satisfying those conditions, to execute the different execution paths. Also, for each considered path, the expressions to compute the corresponding outputs, or concrete outputs satisfying them, are generated. Methods to which we want to apply `aPET` are selected in the outline view.

Let us select method `getData` of class `DBimp`, and generate test cases for it with `aPET` using default parameters. For this, we just click over the `Apply` button and in the console area we can observe that 2 test cases have been generated. Let us focus on the first test case which is shown in Figure 6.11.

- In the `Input` section, `Args` stands for the value of the input arguments; in this case `ref(A)` and `null` are the initial values computed for the input parameters this and `w`. `State` shows the input state. It contains only one object (the caller object) of class `DBimp` identified by `A`.

Figure 6.9: Deadlock-guided testing on the working example

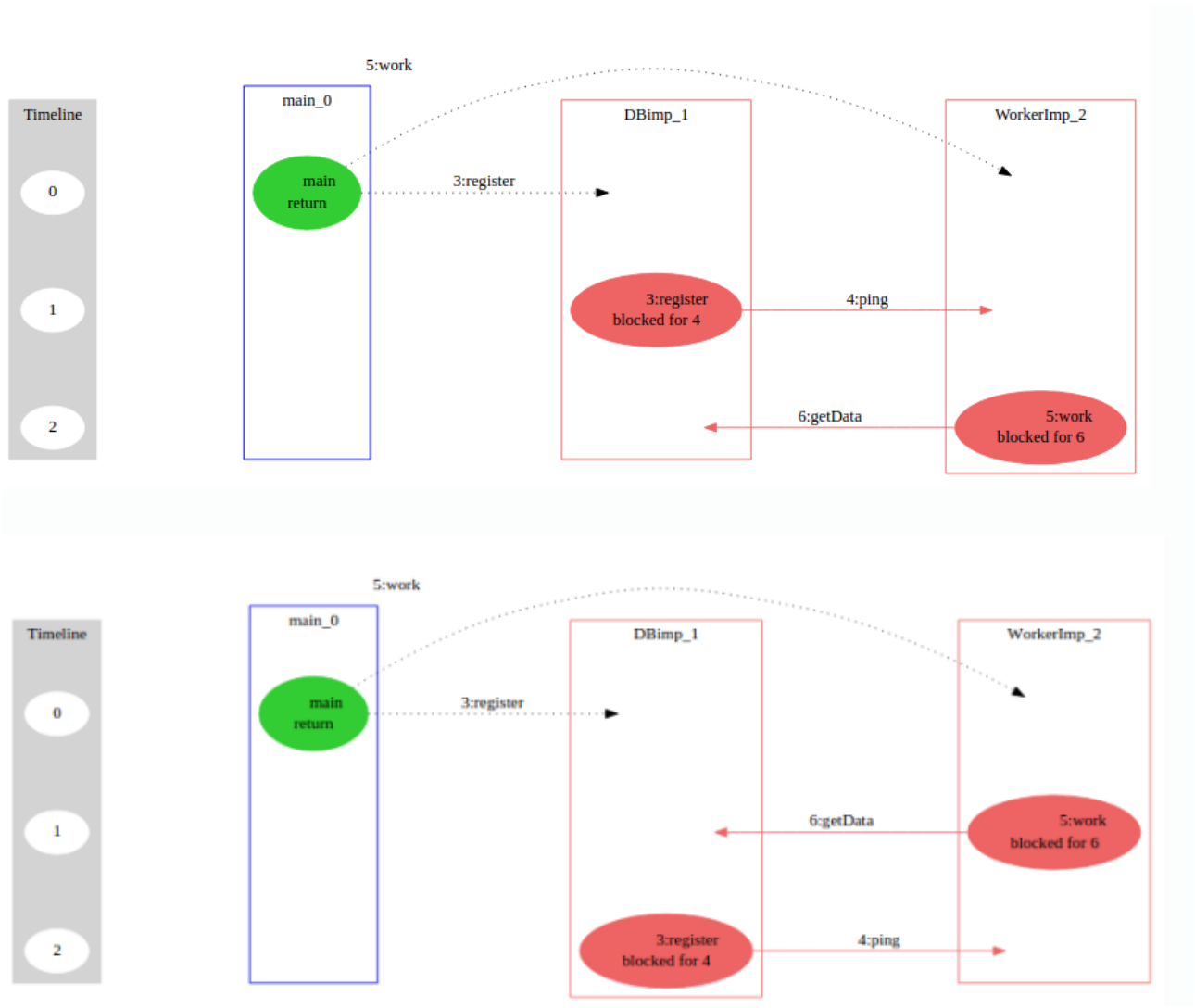

Figure 6.10: Sequence diagrams of deadlock executions

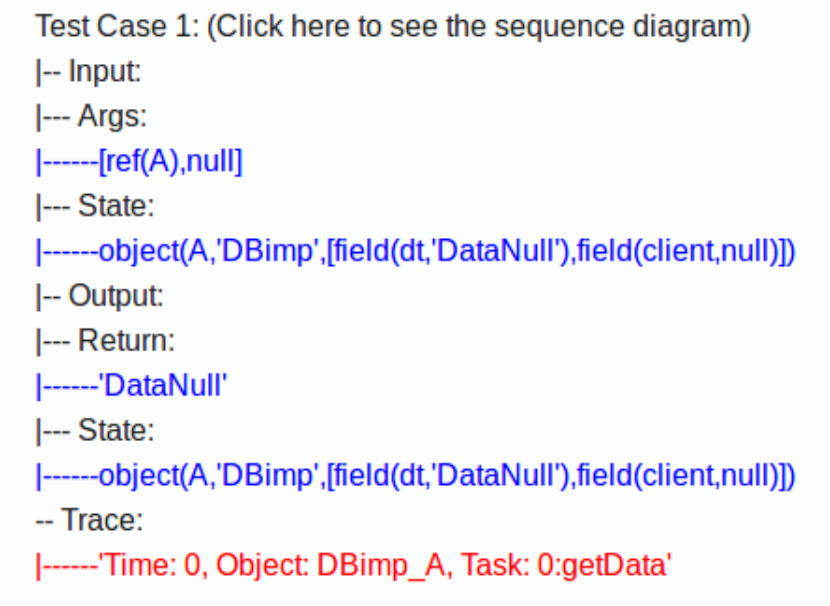

Figure 6.11: TCG with `aPET` for method `getData`

- The `Output` section contains the `Return` value, followed by the final state. The return type of method `getData` is `Data` and it returns `DataNull` (as the value of `client` field is `null` as same as the value of `w`), and, the final state contains only the object `A`.

`aPET` also generates the traces associated with each test case and the corresponding sequence diagrams to graphically visualize the traces. They are displayed by clicking on ``` ``Click here to see the sequence diagram'' ``` in each test case. As in SYCO, if we click over one time point of the trace, the corresponding line in the source code is highlighted (in yellow color) in the code area.

6.3.1 Parameters of `aPET`

The parameters available for `aPET` are shown in Figure 6.12, with their corresponding default values. In the following we describe the meaning and available values for the different parameters:

Concrete test-cases or path-constraints. The result of each feasible execution path in the symbolic execution can be given in the form of (unresolved) path constraints (value `Path constraints`), or in the form of a concrete test case (value `Concrete tests`), where arbitrary concrete values satisfying the constraints are generated. Value `Hybrid` generates concrete data only for functional data, leaving path constraints involving numeric variables. As an example, let us consider again the TCG with `aPET` of method `getData` of class `DBimp` with value `Path constraints`. The two computed test cases are shown in the screenshot below:

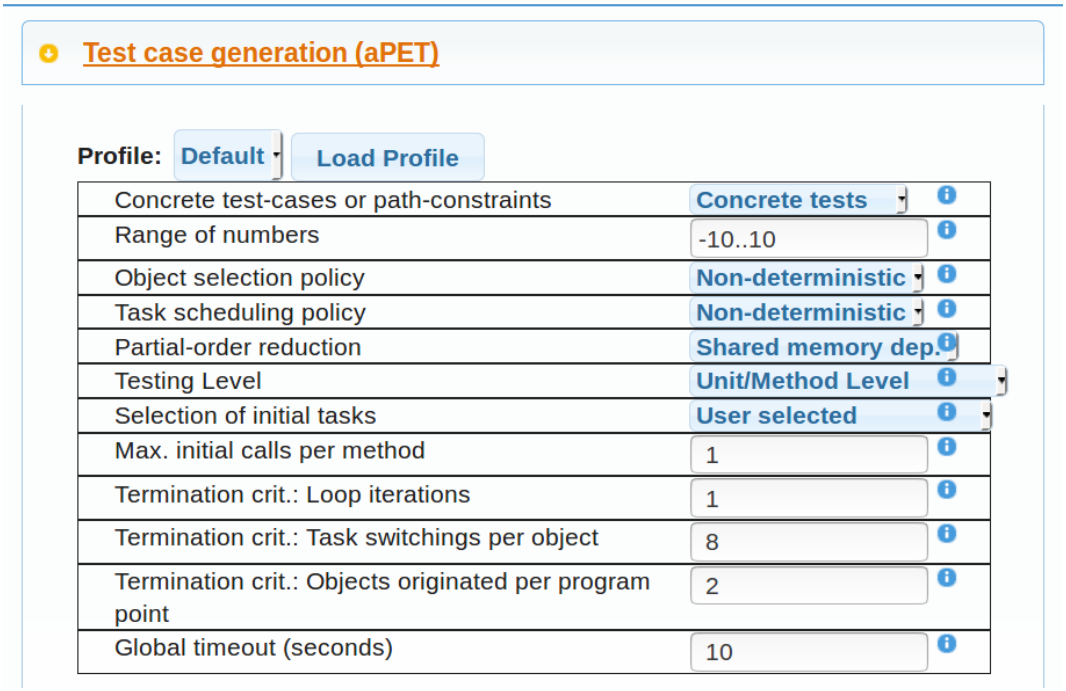

Figure 6.12: The aPET parameters

```
Number of Test Cases: 2 (Click here to see the symbolic execution tree)
Test Case 1: (Click here to see the sequence diagram)
|-- Input:
|--- Args:
|------[ref(A),B]
|--- State:
|------object(A,'DBimp',[field(dt,C),field(client,B)])
|-- Output:
|--- Return:
|------C
|--- State:
|------object(A,'DBimp',[field(dt,C),field(client,B)])
|-- Constraint Store:
:--- The list [B,C] contains the variables unbounded
-- Trace:
|------'Time: 0, Object: DBimp_ref(A), Task: 0:getData'
```

```
Test Case 2: (Click here to see the sequence diagram)
|-- Input:
|--- Args:
|------[ref(A),B]
|--- State:
|------object(A,'DBimp',[field(dt,C),field(client,D)])
|-- Output:
|--- Return:
|------'DataNull'
|--- State:
|------object(A,'DBimp',[field(dt,C),field(client,D)])
|-- Constraint Store:
:--- D is different than B
:--- The list [C,D,B] contains the variables unbounded
-- Trace:
|------'Time: 0, Object: DBimp_ref(A), Task: 0:getData'
```

The first test case can be read as: the initial and final states contain an object `A`, such that the values of fields `dt` and `client` keep unknown (variables C and B) during the path contraint. If we look at the constraint store, we see both variables are unbounded. The return value is `C` (value of field `dt`), because the value of parameter `w` is equal to the value of `client`, (e.g. the worker `B`) was previously registered as its client).

On the other hand, the second test case returns `'DataNull'`, which is different than C (value of field `dt`), and if we look at the constraint store we can see a new constraint `D is different than B`, that is, the value of `client` field is different than the value of `w` (e.g. the worker `B` was not previously registered as its client).

Range of numbers for concrete test cases. It allows specifying the domain for numeric variables and it is given in the format *Min..Max*. This option is only applicable when that concrete test-cases are generated.

Testing Level. It allows us to select between `Unit/Method Level` (by default) and `Integration/System Level`. In this level, selected initial tasks (see the following parameter) are used as it is explained in Chapter 5.1.

Selection of initial tasks. It allows us to select between `User selected` (by default) and `Deadlock interfering tasks`. By default initial tasks are the ones selected by the user in the outline view. If `Deadlock interfering tasks` is chosen, then the system testing is performed from an initial context containing tasks inferred by the predicate in Figure 5.4.

Max. initial calls per method. The specified number (by default 1) is used as a limit on the maximum number of instances of methods in the initial contexts during system testing.

Termination crit.: Loop iterations. The specified number (by default 1) is used as a limit on the maximum number of loop iterations or function recursive calls which are allowed in symbolic execution.

Termination crit.: Task switchings per object. The specified number (by default 8) is used as a limit on the maximum number of task switchings per object which are allowed in symbolic execution.

Termination crit.: Objects originated per program point. The specified number (by default 2) is used as a limit on the maximum number of objects originated per program point which are allowed in symbolic execution.

Parameters *Object selection policy*, *Task scheduling Policy*, *Partial-order reduction* and *Global timeout* have the same meaning as in SYCO (see Section 6.2.3). The first two have however different default values in `aPET`, namely `Round-robin`, and `FIFO` respectively. This is because, in the context of symbolic execution, it is much more likely to run into state explosion problems with non-deterministic schedulings.

6.3.2 System testing with `aPET`

For the sake of simplicity, parameter *Max. initial calls per method* imposes a global maximum cardinality for each initial task considered. This is done to help the user avoiding to define a local cardinality for each of them. Option *Min. initial calls per method* is not available, so the minimum cardinality for each task is 1 by default. Let us select methods `register` and `getData` of class `DBimp`, and we enable the parameter `System Testing` with parameter *Max. initial calls per method* by default. Then, the system testing is performed with $\mathcal{T}_{ini} = \{(\mathsf{DBimp.register}, 1, 1), (\mathsf{DBimp.getData}, 1, 1)\}$. If we click over Button `Run`, 16 test cases are obtained by `aPET`, and none of them are deadlock.

Each of these test cases are achieved from one of these two initial contexts: (1) a context containing two instances of `DBimp` with a task `1:register` and a task `0:getData`, respectively or (2) a context composed by an instance of `DBimp` with tasks `1:register` and `0:getData`. If we press over the first sequence diagram (Figure 6.13), we can observe two edges which indicates tasks and locations inside the initial context during time 0. First arrow means that `DBimp_1` has to perform task `0:getData` and the second one means `DBimp_1` has to perform task `1:register`.

Let us enable now the parameter `Initial tasks` and click over `Run`. 2 test cases are obtained, all of them are deadlock. We can now observe the sequence diagram in Figure 6.14. Here, asynchronous calls performed during time 0 are those tasks inferred by the predicate in Figure 5.4.

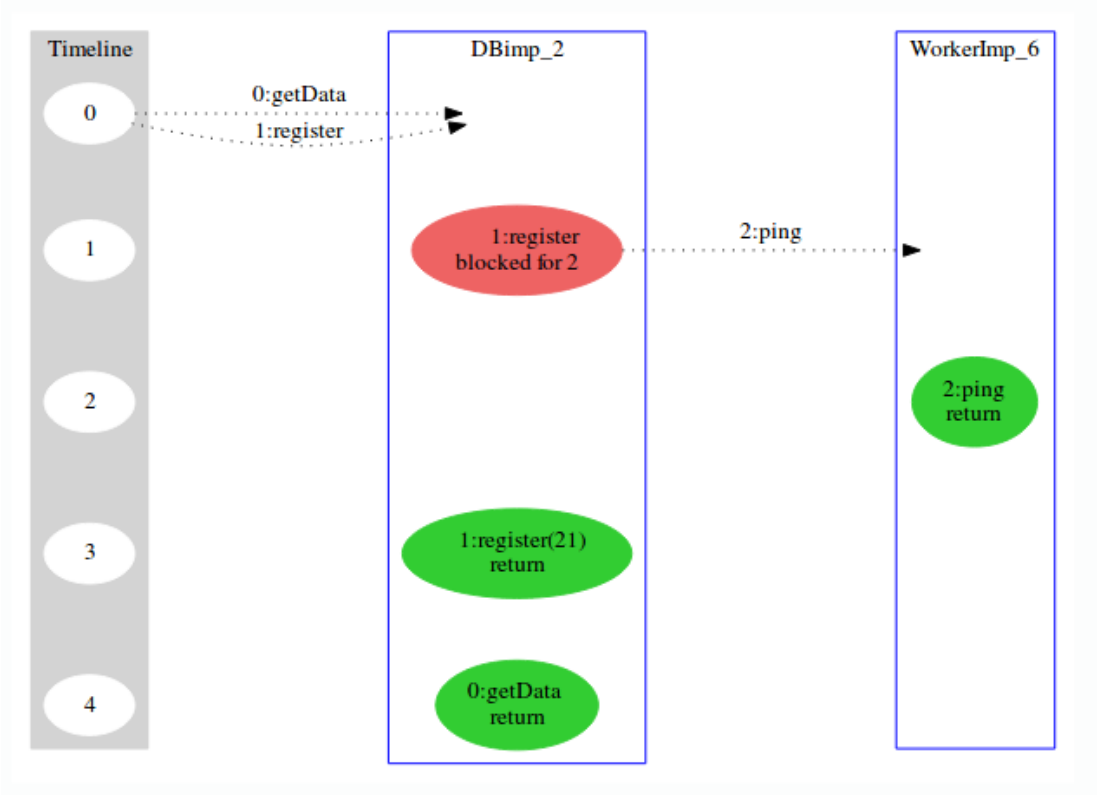

Figure 6.13: Sequence diagram of system testing of methods `register` and `getData`

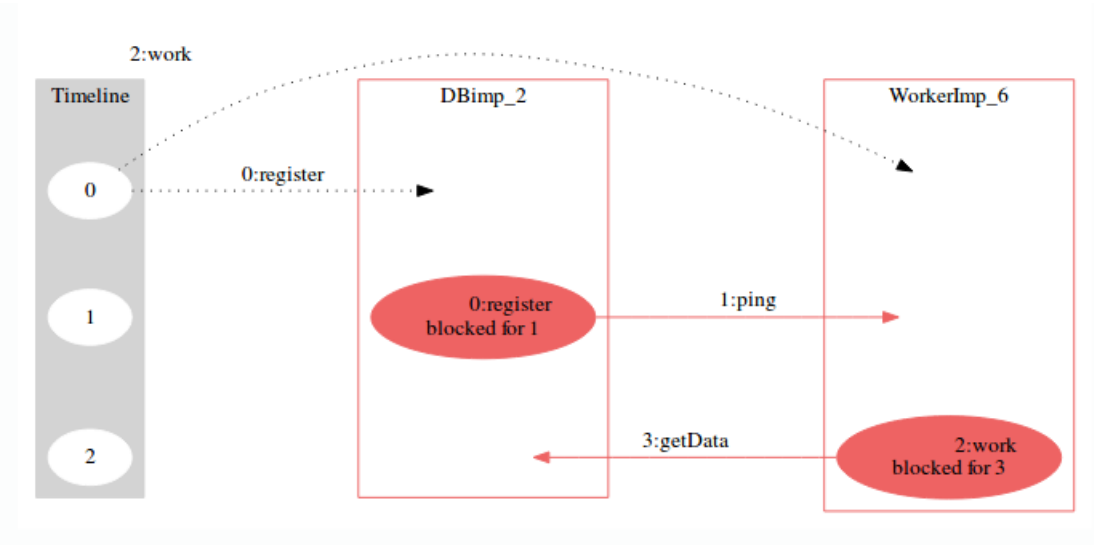

Figure 6.14: Sequence diagram of system testing of methods inferred by the predicate in Figure 5.4

Chapter 7

Experimental Evaluation

This chapter summarizes our experimental results, which aim at demonstrating the applicability, effectiveness, and impact of the proposed techniques, as we can see in Table 7.1.The benchmarks we have used include: (i) classical concurrency patterns containing deadlocks, namely, *DBW* implements a communication protocol between a database and several workers, *F* is a distributed factorial, *PP* is the pairing problem, *HB* the hungry birds problem, *UF* is a loop that creates asynchronous tasks and locations, *SB* is an extension of the sleeping barber; and, (ii) deadlock free versions of some of the above, named *fP* for the *P* program, for which deadlock analyzers give false positives. All of these programs contain a class *Main* which implements a method main with several integer parameters. Code of all of these benchmarks can be found at `http://costa.ls.fi.upm.es/apet`.

Table 7.1 compares the results of our deadlock guided testing (DGT) methodology both for the **deadlock-per-cycle** (**DGT (d-p-c)**) and **all-deadlocks** (**DGT (all)**) criteria, against those obtained using the standard symbolic execution. In all cases, the default POR techniques included in our testing framework are used. Each benchmark is executed with 2 different limits on loop iterations (column **k**).The selected limits for each benchmark are different and have been chosen according to the complexity of the benchmark. For the symbolic execution and the DGT with the **all-deadlock** criterion settings we measure: the number of solutions or complete derivations (columns **Ans**) and the total time taken (columns **T**). For the DGT with the **deadlock-per-cycle** criterion, besides the time (column **T**), we measure the "number of deadlock executions"/"number of unfeasible cycles"/"number of abstract cycles inferred by the deadlock analysis" (column **D/U/C**), and, since the DCGTs for each cycle are independent and can be performed in parallel, we also show the maximum time measured among the different DCGTs (column $\mathbf{T_{max}}$). For instance, for *HB* with $k = 4$ (namely, HB_4), whereas the standard symbolic execution blows-up, our DGT has been able to find all its 1145 deadlock executions in 2847 ms. Also, our DGT for the **deadlock-per-cycle** criterion is telling us that the program has five different abstract deadlock cycles (found by the deadlock analysis), but it only found a feasible deadlock execution for two of them (therefore 3 of them were spurious), being $6912ms$ the total time of the process, and $3237ms$ the time of the longest DCGT (including the time of the deadlock analysis), and, hence the total time assuming an ideal parallel setting with 5 processors.

Columns in the group **Speedup** show the gains in time of DGT both for **deadlock-per-cycle** (columns $\mathbf{T_{gain}}$and $\mathbf{T^{max}_{gain}}$) and **all-deadlocks** (column $\mathbf{T^{all}_{gain}}$) over the standard

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

Table 7.1: Experimental evaluation

symbolic execution. In the case of the **deadlock-per-cycle** criterion we provide the gains both assuming a sequential setting, hence considering value **T** of DGT (column $\mathbf{T_{gain}}$), and an ideal parallel setting, therefore considering $\mathbf{T_{max}}$(column $\mathbf{T^{max}_{gain}}$). The gains are computed as X/Y, X being the measure of standard symbolic execution and Y that of the corresponding DGT. Times are in milliseconds and are obtained on an Intel(R) Core(TM) i7 CPU at 2.3GHz with 8GB of RAM, running Mac OS X 10.8.5. A timeout of 180s is used. When the timeout is reached, we write $>X$ to indicate that for the corresponding measure we have got X units in the timeout. In the case of the speedups, $>X$ indicates that the speedup would be X if the process finishes right in the timeout, and hence it is guaranteed to be greater than X.

Our experiments confirm our claim that systematic testing complements deadlock analysis. In the case of programs with deadlock, we have been able to provide concrete test cases (including full traces and scheduling decisions) for feasible deadlock cycles and to discard unfeasible cycles. For deadlock-free programs, we have been able to discard all potential cycles and therefore prove deadlock freedom (modulo the termination limit used). More importantly, the experiments demonstrate that our DGT methodology is effective and that it can achieve a notable reduction of the search space over standard (symbolic) systematic testing. The gains of DGT both in time and in number of explored states are enormous (more than three orders of magnitude in many cases). It can be observed that the gains are much larger in the examples in which the deadlock analysis does not give false positives (namely, in DBW_3, F_4, F_{10}, PP_9, UF_3). An explanation for this is that, in general, the generated constraints for unfeasible cycles are often not able to guide the exploration effectively (e.g. in HB_4). Indeed, if we consider HB_5 we cannot find a representative of one of the abstract cycles but neither we are able to prove it is a false positive. Even in these

cases, DGT outperforms symbolic execution in terms of scalability and flexibility. Let us also observe that the gains are less notable in deadlock-free examples. That is because, on one hand, all cycles are unfeasible in this case, and, on the other, each DCGT cannot stop until all potential deadlock paths have been considered. As expected, when we consider a parallel setting, the gains are much larger.

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

Chapter 8

Related Work

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

8.1 Deadlock Analysis

There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [15, 16] and thread-based programs [27, 29], is based on static analysis techniques. Static analysis can ensure the absence of errors, however it works on approximations (especially for pointer aliasing) which might lead to a "don't know" answer.

Our work complements static analysis techniques and can be used to look for deadlock paths when static analysis is not able to prove deadlock freedom. Using our method, we try to find a deadlock by exploring the paths (possibly infinite) given by a deadlock detection algorithm that relies on the static information. Although we have used the output given by the deadlock analyzer of [15], our combined approach could use the output of other static analyzers (e.g., [16]) without requiring any conceptual change to the combined framework.

8.2 Symbolic Execution, Verification, Model Checking and Testing

The core of our CLP-based framework is the symbolic execution engine presented in Chapter 3. By relying only on this component, one can clearly do (non-guided) deadlock detection already, and besides other types of errors can also be captured (e.g., find critical states that can cause the system to crash). This is the approach taken in model checking and other verification techniques which are based on symbolic execution to automatically verify correctness properties. Indeed, deadlock detection has been intensively studied in the context of model checking (see, e.g., [28]).

Both static and dynamic testing aim at finding bugs, among them deadlocks (see, e.g., [11, 25, 26, 21]). Indeed, symbolic execution is at the core of static testing systems and our

symbolic execution engine is the basis for the `aPET` testing system [5].

Current research on testing for concurrent systems has focused on avoiding the generation of redundant executions which result from interleaving independent processes (e.g., processes that operate on disjoint areas of the memory). Dynamic Partial Order Reduction (DPOR) [6, 14, 31] is a successful technique to avoid such redundancies. Our work is orthogonal to such line of research, in the sense that we can use DPOR techniques within our framework and the combined approach is still valid. Indeed, our implementation uses the techniques described in [6] to eliminate redundancies, as we have mentioned in Chapter 7.

8.3 Hybrid Approaches

We now relate our work to hybrid approaches that use static information during testing for deadlock detection, namely [24] and [2]. As regards [24], it first performs a transformation of the program into a trace program that only keeps the instructions that are relevant for deadlock and then dynamic testing is performed on such program. The approach is fundamentally different from ours: in their case, since model checking is performed on the trace program (that over-approximates the deadlock behaviour), the method can detect deadlocks that do not exist in the program, while in our case this is not possible since the testing is performed on the original program and the analysis information is only used to drive the execution. Besides, our work is based on static testing that generalizes dynamic testing to allow any input data.

As regards [2], the information inferred from a type system is used to accelerate the detection of potential cycles. This work shares with our work that information inferred statically is used to improve the performance of the testing tool, however there are important differences: first, their method developed for Java threads captures deadlocks due to the use of locks and cannot handle wait-notify, while our technique is not developed for specific patterns but works on a general characterization of deadlock of asynchronous programs; their underlying static analysis is a type inference algorithm which infers deadlock types and the checking algorithm needs to understand these types to take advantage of them, while we base our method on an analysis which infers descriptions of chains of tasks and a formal semantics is enriched to interpret them.

Chapter 9

Conclusions and Future Work

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

Secondly, we have presented a new technique to perform system or integration testing by means of a set of initial tasks provided by the user. Furthermore, we reduce the combinatorial explosion by inferring the *deadlock-interfering* tasks which could lead the execution to deadlock. The experimental evaluation of this approach remains as future work.

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

Finally, we have implemented these techniques within the prototype tool SYCO/`aPET`, a dynamic/static testing tool, which is available online and it appears in the proceedings of the *25th International Conference on Compilers Constructions*. Improvements and extensions of the tool remain also as future work.

Bibliography

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

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

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

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

[5] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *Practical Aspects of Declarative Languages (PADL'12)*, Lecture Notes in Computer Science, pages 123–137. Springer, January 2012.

[6] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In Erika Ábrahám and Catuscia Palamidessi, editors, *34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems (FORTE 2014)*, volume 8461 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2014.

[7] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Test Case Generation of Actor Systems. In *13th International Symposium on Automated Technology for Verification and Analysis, ATVA 2015. Proceedings*, volume 9364 of *Lecture Notes in Computer Science*, pages 259–275. Springer, 2015.

[8] E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint P.sburg, Russian Federation, August 18-26, 2013*, pages 595–598. ACM, 2013.

[9] E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 409–424. Springer, 2016.

[10] E. Albert, M. Gómez-Zamalloa, and M. Isabel. SYCO: A Systematic Testing Tool for Concurrent Objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 269–270. ACM, 2016.

[11] M. Christakis, A. Gotovos, and K. 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*, pages 154–163. IEEE Computer Society, 2013.

[12] Frank S. de Boer, Dave Clarke, and E. Johnsen. A Complete Guide to the Future. In Rocco de Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, March 2007.

[13] J. Doménech, S. Genaim, E. Johnsen, and Rudolf Schlatte. EasyInterface: A toolkit for rapid development of GUIs for research prototype tools. In *Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, Lecture Notes in Computer Science, pages 379–383. Springer, 2017.

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

[15] A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems (FMOODS/FORTE 2013)*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, June 2013.

[16] E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. DeadLock Analysis of Concurrent Objects – Theory and Practice, 2013.

[17] N. Kobayashi, E. Giachino and C. Laneve Deadlock Analysis of Unbounded Process Networks. In *CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 63–77, 2014.

[18] C. Laneve, E. Giachino and M. Lienhardt. A framework for deadlock detection in core ABS. *Software and System Modeling*, 15(4):1013–1048, 2016.

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

[20] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue*, 10(4-6):659–674, July 2010.

[21] Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, pages 245–264, 2000.

[22] M. Isabel. Supervisers: E. Albert and M. Gómez-Zamalloa. Techniques to improve testing scalability on concurrent programs.

[23] E. Johnsen, Reiner Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

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

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

[26] 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`.

[27] 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.

[28] I. Rabinovitz and O. Grumberg. Bounded Model Checking of Concurrent Programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.

[29] 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.

[30] K. Sen and G. 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, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[31] S.a 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 Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.