TECHNIQUES TO IMPROVE TESTING SCALABILITY ON CONCURRENT PROGRAMS

Combining Static Analysis and Testing for Deadlock Detection

Miguel Isabel Márquez

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Junio 2015

Directores:
Elvira Albert Albiol
Miguel Gómez-Zamalloa Gil

Contents

Agradecimientos

Cuando empecé esta carrera en 2010 pensaba que nunca llegaría el momento de escribir estas palabras. Son muchas las personas que me han apoyado durante estos cinco años.

Me apasiona aprender cosas nuevas y, por ello, quiero agradecer a aquellos profesores que han conseguido inspirarme y motivarme día a día durante la carrera; a los miembros del *grupo COSTA*, por su predisposición y ayuda; a *Miky Gómez-Zamalloa*, por sus consejos y sus trucos informáticos y a *Elvira Albert*, por confiar en mí y darme la oportunidad de descubrir cómo es el maravilloso mundo de la Investigación.

Lo bueno de estar sentado junto a gente brillante, es que cada día se puede aprender algo nuevo de ellos. Por eso quiero expresar mi gratitud a cada uno de mis compañeros. En especial a *Manuel Morán*, por ser uno de los mejores profesores que he tenido y su ayuda incondicional a lo largo de estos años.

También quiero agradecerselo a mi familia, a aquellos que son como mis segundos padres, mis hermanos y mis sobrinos de sangre. A mi madre, por su ánimo y preocupación constante. Y por último, a mi padre. *Mamá siempre dice que estarías orgulloso de mí y a ti te dedico especialmente este trabajo.*

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 un análisis estático y el *testing* dinámico para la detección efectiva de *deadlocks* en programas asíncronos. Las principales contribuciones son: (1) Presentamos una semántica extendida que permite la detección instantánea de bloqueos durante el *testing* y dar al usuario una descripción precisa de la traza de *deadlock*. (2) Además combinamos, nuestra herramienta de *testing* con las descripciones abstractas de posibles ciclos de *deadlock*, inferidos por un análisis estático existente. Tales descripciones son usadas en nuestra semántica extendida para guiar la ejecución hacia posibles caminos de *deadlock*, mientras que el resto son podados. Cuando el programa contiene un bloqueo, el uso combinado del análisis estático y el *testing* nos proporciona una técnica efectiva para encontrar trazas de *deadlock*. En caso de que el programa no contenga ninguno, pero el análisis sí que lo detecte debido a pérdidas de precisión, podríamos llegar a demostrar la ausencia de *deadlocks*.

Palabras Clave

Restricciones de Ciclo de *Deadlock*, Análisis de *Deadlock*, Testing Guiado, Ciclo Abstracto, Detección de *Deadlocks*, Intervalo de Espera, Cadena de *Deadlock*, Tabla de Entrelazados.

Abstract

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

Key Words

Deadlock Cycle Constraint, Deadlock Analysis, Guided Testing, Abstract Cycle, Deadlock Detection, Waiting interval, Deadlock Chain, Interleaving Table.

Chapter 1

Introduction

1.1 Introduction

In order to improve program responsiveness, many modern programming languages and libraries promote a model of actors, in which asynchronous tasks can execute concurrently with their caller tasks, until their callers explicitly wait for their completion. We consider an *asynchronous* language which allows spawning asynchronous tasks at distributed locations, and has two operations for *blocking* and *non-blocking* synchronization with the termination of asynchronous tasks and, thus, it is possible to introduce deadlocks.

Indeed, *deadlock* is one of the most common programming errors and, therefore, a main goal of verification and testing tools for concurrent programs is, respectively, proving *deadlock freedom* and *deadlock detection*. In general, deadlock situations are produced 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.

In our 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. The contribution of this work is a testing framework for deadlock detection in asynchronous systems.

Static analysis and testing are two different ways of detecting deadlocks that often complement each other and, thus, it seems quite natural to combine them. Static analysis evaluates an application by examining its code but without executing it. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks, months or years after releasing the application. This aspect of static analysis is especially important in *security assurance*, because security attacks try to exercise an application in unpredictable and untested ways. However, when a deadlock is found, state-of-the-art analysis tools [11, 12, 9, 18] provide little (and often none) information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and locations), it is essential to know the task interleavings that have occurred and the locations involved in the deadlock, i.e., provide a concrete *deadlock trace* that allows the programmer to identify and fix the problem.

In contrast, testing consists in executing the application: in *dynamic* testing, it is executed for concrete input values, while *static* testing does not make any assumption on the input values and the application is executed symbolically using constraint variables. The primary advantage of testing for deadlock detection is that it can provide the deadlock trace with all information that the user needs in order to fix the problem. There are two shortcomings though: (1) In dynamic testing, since not all inputs can be tried, there is no guarantee of deadlock freedom; and in static testing, one needs to assume some termination criteria and, thus, it is again not possible to ensure deadlock freedom. (2) Besides, although recent research tries to avoid redundant exploration as much as possible [10, 21, 8, 1, 4], the search space (without redundancies) can be huge. This is a threaten to the application of testing in concurrent programming.

When the focus of testing is on a particular property, it might not be necessary to generate the whole search space (even without redundancies). Instead, we aim

at guiding the execution only towards those paths that might lead to deadlock, and prune those that we know certainly that cannot lead to deadlock. This paper proposes a seamless combination of static analysis and testing for effective deadlock detection as follows: an existing static deadlock analysis [11] is first used to obtain *abstract* descriptions of potential deadlock cycles which are then used to guide a testing tool in order to find associated deadlock traces (or discard them). Technically, the main contributions of the paper are:

1. *We extend a standard semantics for asynchronous programs with information about the task interleavings made, and the status of tasks (i.e., awaiting, blocked, or finished).*

   The extended semantics will allow us: (1) to provide deadlock traces when a deadlock is found, (2) an early detection of deadlock states during execution and (3) its combined use with static analysis. In essence, whenever the scheduler changes the executing task, we assign a unique *time identifier* to it, determined by the location and task identifiers; we add a tuple with (i) a time value that is larger than those of the tasks that have been selected so far and (ii) its *status* (e.g., if it finished or is blocked awaiting for the termination of another task). All this information is stored in the *interleavings table* along the execution.

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

   Our characterization is based on the notion of *deadlock chains* which are chains $t_0, \ldots, t_n$ of times whose tasks are basically awaiting for the termination of another one in the next location until t_n that waits for the termination of a task of the (blocked) location in which t_0 executes. A state is deadlock iff it contains this kind of deadlock chains. This notion is useful per se since it is not straightforward to detect a deadlock during the execution. This is because there can be one (or several locations) that keep on executing (maybe even go into an infinite computation) while, due

to a deadlock chain in other locations, we are sure that the execution will eventually lead to a deadlock. Thus, it allows early detection of deadlocks paths.

3. *We present a new methodology to detect deadlocks which combines testing and static analysis.*

   The *deadlock cycles* inferred by static analysis are used by our framework to generate constraints that must be fulfilled by every state in a deadlock derivation. Using these constraints, our extended semantics guides the testing process towards paths that might lead to a deadlock cycle and discard *deadlock-free paths*. The effectiveness of this framework is highly related to the deadlock analysis: the more accurate the analysis is, the more reduced the search space will be. *Deadlock analysis* is a very active research area and whenever new deadlock analyses are developed, our framework will become more effective as well; since it can be easily adapted to use a new *constraints generator*.

4. *We introduce several deadlock-based testing criteria.*

   Our criteria allow us to apply our methodology to find: the first deadlock trace, a representative deadlock trace of each deadlock cycle, or all deadlock traces for each cycle.

5. *The implementation in the aPET system [5] and a thorough experimental evaluation.*

   Our experiments show that we can find *deadlock traces* for the potential deadlock cycles with a significant reduction of the required state exploration. We have used our framework on programs where deadlock analysis loses precision and outcomes false-positives and also show a reduction in time and space to prove these programs are actually deadlock-free.

Chapter 2

Language

In this chapter, we present the language that will be used during the formalization of the framework and an example programmed in this language that illustrates its semantics. Section 2.1 presents the language syntax and semantics and Section 2.2 illustrates the well-known *Sleeping Barber Problem*, which is used as running example.

2.1 Asynchronous Programs: Syntax and Semantics

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

The language uses *future variables* to check if the execution of an asynchronous task has finished. An asynchronous call $\mathsf{m}(\bar{\mathsf{z}})$ spawned at location x is associated with a future variable f as follows $\mathsf{f} = \mathsf{x}\ !\ \mathsf{m}(\bar{\mathsf{z}})$. Instructions $f.\mathsf{block}$ and $f.\mathsf{await}$ allow, respectively, blocking and non-blocking synchronization with the termina-

tion of m. When a task completes, or when it is awaiting with a non-blocking await for a task that has not finished yet, its processor becomes idle again, chooses the next pending task, and so on. The number of distributed locations need not be known a priori (e.g., locations may be virtual).

Syntactically, a location will therefore be similar to a *concurrent object* [15] and can be dynamically created using the instruction **new**. The program consists of a set of methods of the form M::=T $m(\bar{T}\ \bar{x})\{s\}$, where statements s take the form s::=$s; s$ | x=e | **if** e then s **else** s | **while** e **do** s | **return** | b=**new** | f = x ! $m(\bar{z})$ | f.await | f.block. For the sake of generality, the syntax of expressions e and types T is left open.

Figure 2.1 presents the semantics of the language. The information about $\boldsymbol{\rho}$ in bold font is part of the extensions for testing in Section 4 and should be ignored by now. A *state* or *configuration* is a set of locations and future variables $o_0 \cdots o_n \cdot fut_0 \cdots fut_m$. A *location* is a term $loc(o, tk, h, \mathcal{Q})$ where o is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\perp$ if the location's lock is free, h is its local heap, and $\mathcal{Q}$ is the set of tasks in the location. A *future variable* is a term $fut(id, o, tk, m)$ where id is a unique future variable identifier, o is the location identifier that executes the task tk awaiting for the future, and m is the initial program point of tk. A *task* is a term $tsk(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed or ϵ if the task has terminated.

We assume that the execution starts from a main method without parameters. The initial state is St=$\{loc(0, 0, \perp, \{tsk(0, main, l, body(main))\}$ with an initial location with identifier 0 executing task 0. Here, l maps local variables to their initial values (**null** in case of reference variables) and $\perp$ is the empty heap. $body(m)$ is the sequence of instructions in method m, and we can know the program point pp where an instruction s is in the program as follows pp:s.

As locations do not share their states, the semantics can be presented as a

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

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

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

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

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

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

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

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

Figure 2.1: Semantics of Asynchronous Programs

macro-step semantics [20] (defined by means of the transition "$\longrightarrow$") in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to an await or return instruction. In this case, we apply rule MSTEP to select an available task from a location, namely we apply the function $selectLoc(S)$ to select non-deterministically one *active* location in the state (i.e., a location with a non-empty queue) and $selectTask(o)$ to select non-deterministically one task of o's queue.

The transition $\rightsquigarrow$ defines the evaluation within a given location. NEWLOC creates a new location without tasks, with a fresh identifier and heap. ASYNC

spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tk_1, and it adds a new future to the state. $ini(m)$ refers to the first program point of method m. We assume $o \neq o_1$, but the case $o = o_1$ is analogous, the new task tk_1 is added to $\mathcal{Q}$ of o. The rules for sequential execution are standard and are thus omitted. AWAIT1: If the future variable we are awaiting for points to a finished task, the await can be completed. The finished task t_1 is only looked up but it does not disappear from the state as its status may be needed later on. AWAIT2: Otherwise, the task yields the lock so that any other task of the same location can take it. RETURN: When **return** is executed, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction ϵ). BLOCK2: A y.block instruction waits for the future variable but without yielding the lock. Then, when the future is ready, BLOCK1 allows continuing the execution.

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

2.2 Motivating Example

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

The main creates three locations barber, client and chair and spawns two asynchronous tasks to start the wakeup task in the client and sleeps in the barber, both tasks can run in parallel. The execution of sleeps spawns an asynchronous task

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

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

Figure 2.2: Classical Sleeping Barber Problem

on the chair to represent the fact that the client takes the chair, and then blocks at line 11 until the chair is taken. The task taken first adds the task sits on the client, and then awaits on its termination at line 17 without blocking, so that another task on the location chair can execute. On the other hand, the execution of wakeup in the client spawns an asynchronous task cuts on the barber and one on the chair, isClean, to check if the chair is clean. The execution of the client blocks until cuts has finished. We assume that all methods have an implicit return at the end.

Figure 2.3 summarizes the execution tree of the main by showing some of the macro-steps taken. Derivations that contain a dotted node are not deadlock, while those with a gray node are deadlock. A main motivation of our work is to detect as early as possible that the dotted derivations will not lead us to deadlock and prune them. Let us see two selected derivations in detail. In the derivation ending at node 11, the first macro-step executes cl.wakeup and then b.cuts. Now, it is clear that the location cl will not deadlock, since the block at line 24 will succeed and the other two locations will be also able to complete their tasks, namely the await at line 17 of location ch can finish because the client is certainly not blocked, and also the block at line 11 will succeed because the task in taken

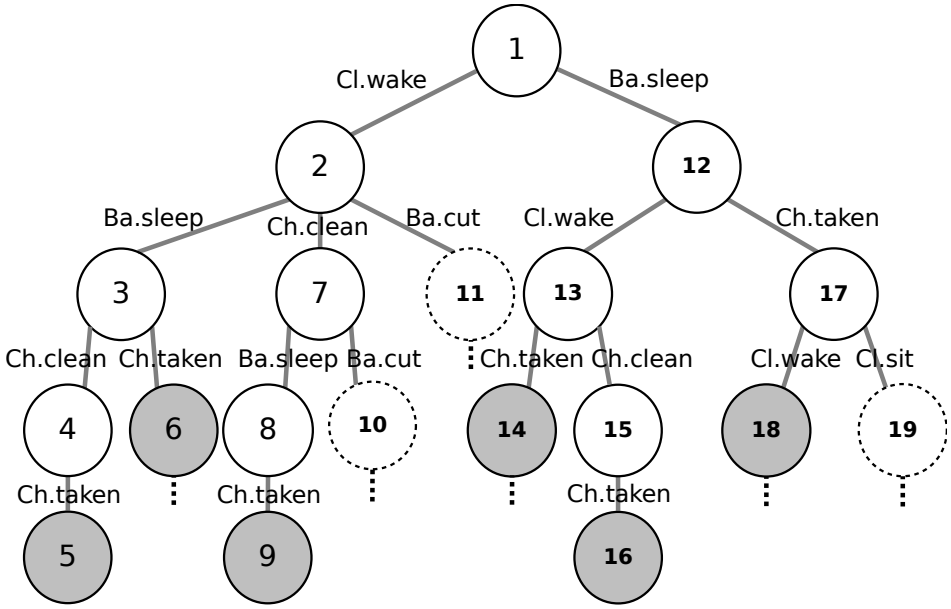


Figure 2.3: Execution Tree of Example 2.2

will eventually finish as its location is not blocked. However, in the branch of node 6, we first select wakeup (and block client), then we select sleeps (and block barber), and then select taken that will remain in the await at line 17 and will never succeed since it is awaiting for the termination of a task of a blocked location. Thus, we certainly have a deadlock. However, location chair can keep on executing an available task isClean. Let us outline five states of this derivation:

$$
\begin{aligned}
St_1 \equiv\ & loc(cl, \bot, h_1, \{tsk(1, wakeup, l_1, \mathsf{body(wakeup)})\})\cdot \\
& \cdot loc(ba, \bot, h_2, \{tsk(2, sleeps, l_2, \mathsf{body(sleeps)})\})\cdot \\
& \cdot loc(ch, \bot, h_3, \emptyset)\cdot \\
& \cdot loc(ini, \bot, h_0, \{tsk(0, main, l_0, \epsilon)\}) \xrightarrow{cl,1} \\
St_2 \equiv\ & loc(cl, 1, h_1, \{tsk(1, wakeup, \mathsf{f_0.block})\})\cdot \\
& \cdot loc(ba, \bot, h_2, \{tsk(2, sleeps, l_2, \mathsf{body(sleeps)}), tsk(3, cuts, l_3, \mathsf{body(cuts)})\})\cdot \\
& \cdot loc(ch, \bot, h_3, \{tsk(4, isClean, l_4, \mathsf{body(isClean)})\})\cdot \\
& \cdot loc(ini, \bot, h_0, \{tsk(0, main, l_0, \epsilon)\})\cdot \\
& \cdot fut(f_0, ba, 3, 12) \xrightarrow{ba,2}
\end{aligned}
$$

$$
\begin{aligned}
St_3 \equiv\ & loc(cl,1,h_1,\{tsk(1,wakeup,\mathsf{f_0.block})\})\cdot \\
& \cdot loc(ba,2,h_2,\{tsk(2,sleeps,l_2,\mathsf{f_1.block}),tsk(3,cuts,l_3,\mathsf{body(cuts)})\})\cdot \\
& \cdot loc(ch,\bot,h_3,\{tsk(5,taken,l_5,\mathsf{body(taken)}),tsk(4,isClean,l_4,\mathsf{body(isClean)})\})\cdot \\
& \cdot loc(ini,\bot,h_0,\{tsk(0,main,l_0,\epsilon)\})\cdot \\
& \cdot fut(f_0,ba,3,12)\cdot fut(f_1,ch,5,15) \xrightarrow{ch,5} \\
St_6 \equiv\ & loc(cl,1,h_1,\{tsk(1,wakeup,l_1,\mathsf{f_0.block}),tsk(6,sits,l_6,\mathsf{body(sits)})\})\cdot \\
& \cdot loc(ba,2,h_2,\{tsk(2,sleeps,l_2,\mathsf{f_1.block}),tsk(3,cuts,l_3,\mathsf{body(cuts)})\})\cdot \\
& \cdot loc(ch,\bot,h_3,\{tsk(5,taken,l_5,\mathsf{f_5.await}),tsk(4,isClean,l_4,\mathsf{body(isClean)})\})\cdot \\
& \cdot loc(ini,\bot,h_0,\{tsk(0,main,l_0,\epsilon)\})\cdot \\
& \cdot fut(f_0,ba,3,12)\cdot fut(f_1,ch,5,15)\cdot fut(f_2,cl,6,25) \xrightarrow{ch,4} \\
St_{6'} \equiv\ & loc(cl,1,h_1,\{tsk(1,wakeup,l_1,\mathsf{f_0.block}),tsk(6,sits,l_6,\mathsf{body(sits)})\})\cdot \\
& \cdot loc(ba,2,h_2,\{tsk(2,sleeps,l_2,\mathsf{f_1.block}),tsk(3,cuts,l_3,\mathsf{body(cuts)})\})\cdot \\
& \cdot loc(ch,\bot,h_3,\{tsk(5,taken,l_5,\mathsf{f_5.await}),tsk(4,isClean,l_4,\epsilon)\})\cdot \\
& \cdot loc(ini,\bot,h_0,\{tsk(0,main,l_0,\epsilon)\})\cdot \\
& \cdot fut(f_0,ba,3,12)\cdot fut(f_1,ch,5,15)\cdot fut(f_2,cl,6,25)
\end{aligned}
$$

The first state is obtained after executing the main where we have the initial location *ini*, three locations created at lines 3, 2 and 4, and two tasks at lines 5 and 6 added to the queues. Note that each location and task is assigned a unique identifier (we use numbers as identifiers for tasks and short names as identifiers for locations). In the next state, the task wakeup has been selected and fully executed. Observe at St_2 the addition of the future variable created at line 22. In St_3 we have executed task sleeps in the barber and added a new future term. In St_6 we execute task taken in the chair (this state is already deadlock as we will see in Section 4.2). The state $St_{6'}$ is obtained after executing task isClean and is not included in Figure 2.3, as it is not relevant at this point. From now on, we use the location and task names instead of numeric identifiers for clarity.

Chapter 3

Preliminaries

This chapter recaps the two techniques that we adopt to develop our framework. The static analysis that we use to infer *potential deadlock cycles* is explained in Section 3.1. Section 3.2 summarizes the main features of aPET, an automated test case generator.

3.1 Deadlock Analysis

In the literature, there is a large number of static analyses that detect deadlocks both in thread-based languages and in actor-based languages [11, 12, 9, 18]. Our choice is the static deadlock analysis in [11] and its implementation *DECO*, a *DEadlock analyzer for Concurrent Objects* whose efficiency and scalability have been proved experimentally on several case studies.

DECO reports that a program is deadlock-free when there are no abstract cycles that could lead to deadlock. On the other hand, when the analyzer reports a potential deadlock, it also provides hints on the program points involved in this deadlock.

The analysis builds a *dependencies graph* from which the deadlock cycles are formed. We now recall the definition of this graph as it is used in Theorem 1.

Definition 1 (Deadlock Dependencies Graph)**.** *Given a program state* $S = \texttt{Loc} \cup \texttt{Fut}$*, where* Loc *and* Fut *are, respectively, the set of locations and futures. We define its dependencies graph* G_S *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* $loc(o, tk, h, \mathcal{Q})$, $loc(o_2, \_, h_2, \mathcal{Q}_2) \in \texttt{Loc}$*, two tasks* $tsk(tk, m, l, \{y.\mathsf{block}; s\}) \in \mathcal{Q}$ $tsk(tk_2, m_2, l_2, s_2) \in \mathcal{Q}_2$ *and a future variable* $fut(y, o_2, tk_2, m_2) \in \texttt{Fut}$ *where* $s_2 \neq \epsilon(v)$*.*

2. ***Task-Task****:* $tk_1 \rightarrow tk_2$ *iff there are two locations* $loc(o, \_, h, \mathcal{Q})$, $loc(o_2, \_, h_2, \mathcal{Q}_2) \in \texttt{Loc}$*, two tasks* $tsk(tk_1, m_1, l_1, \{sync; s\}) \in \mathcal{Q}$ $tsk(tk_2, m_2, l_2, s_2) \in \mathcal{Q}_2$ *and a future variable* $fut(y, o_2, tk_2, m_2) \in \texttt{Fut}$*, where* $sync \in \{y.\mathsf{block}, y.\mathsf{await}\}$ *and* $s_2 \neq \epsilon(v)$*.*

3. ***Task-Location****:* $tk \rightarrow o$ *iff there is a location* $loc(o, tk_2, h, \mathcal{Q}) \in \texttt{Loc}$ *and a task* $tsk(tk, m, l, s) \in \mathcal{Q}$ *with* $tk \in \mathcal{Q}$*,* $tk_2 \neq tk$ *and* $s \neq \epsilon(v)$*.*

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 block instruction will generate two dependencies, whereas an await will generate only a dependency. Besides, every task without the location's lock (which is not finished) has a dependency to its location. *If there is a cycle in the graph, then the program is deadlock.* This definition can be better understood by means of an example.

Example 1. *Let us consider the final (deadlock) state for derivation ending at node* 6 *described in Section 2.2. Here, we denote by* o:m *a task executing method* m *on location* o*. We have the following seven dependencies in this state which form a cycle:*

d1	*cl*	$\rightarrow$	*ba:cuts*	*d4*	*ba*	$\rightarrow$	*ch:taken*	*d7*	*ch:taken*	$\rightarrow$	*cl:sits*
d2	*cl:wakeup*	$\rightarrow$	*ba:cuts*	*d5*	*ba:sleeps*	$\rightarrow$	*ch:taken*				
d3	*cl:sits*	$\rightarrow$	*cl*	*d6*	*ba:cuts*	$\rightarrow$	*ba*				

Observe that in location cl *we have a blocking task* cl:wakeup *executing a* block *which induces dependencies location-task d1 and task-task d2 above, and a waiting task* cl :sits *that induces the dependency task-location d3. In* ba*, we have the blocking task* ba:sleeps *that adds dependencies location-task d4 and task-task d5 and a waiting task* b:empt *that adds the dependency task-location d6. Finally, in* ch*, we have a waiting task* ch:taken *that induces a dependency task-task d7. The cycle involves the locations* ba *and* cl *and the three tasks* ba:cuts*,* ch:taken*,* cl:sit*.*

Similar to other approaches [16, 2], the deadlock analysis consists in constructing an *Abstract Dependencies Graph* that over-approximates the *Deadlock Dependencies Graph* of any state and, then, look for cycles within the graph. It returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points, $tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions.

The abstraction that we 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* [3], i.e., it distinguishes the actions performed by the different location abstractions, (e.g., an abstract task is of the form $o_{pp}.m$ where o_{pp} is the abstract location that executes it).

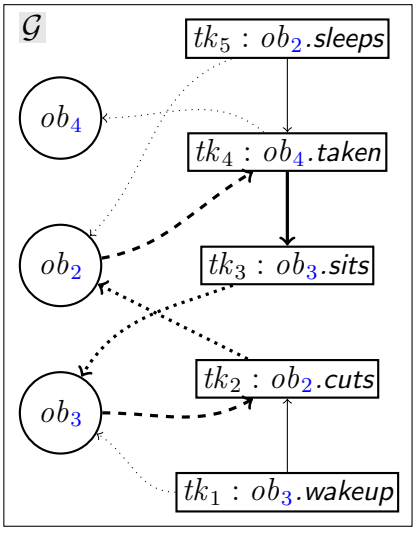
A more precise abstraction for locations can include the program point where its ancestor location is created, e.g., if the instruction client=new Cl(); at line 3 is executed from two different locations (one created at line 30 and another at line 50, the analysis uses two abstractions $client_{30:3}$ and $client_{50:3}$ and treats the two abstract locations separately. The same accuracy improvement can be made for task abstractions.

The length of the ancestors chain k is a parameter of the analysis and any $k \geq 0$ can be used. The same length k adopted by the deadlock analysis should be used in the semantics so that we can take fully advantage of the information

in the abstract chains during testing. Our semantics can be easily extended to keep the k ancestor locations by simply adding an additional parameter in tasks with the list of the ancestor location identifiers (up to k).

The analysis performs the following steps: (1) it generates an *abstract dependencies graph* $\mathcal{G}$ that over-approximates the dependencies graphs of any reachable state S. This graph is obtained by abstracting locations and tasks, which are used as nodes, and its edges are the abstracted version of the ones in Definition 1. Finally, (2) it looks up the cycles contained within the abstract dependencies graph.

Example 2. *Following with the program in Section 2.2, if we perform the first step of the deadlock analysis, its abstract dependencies graph $\mathcal{G}$ is as follows:*



The locations created at lines 2, 3 and 4 are abstracted to ob_2, ob_3, ob_4, respectively. In a similar way, if we perform the Points-to analysis with the parameter $k = 1$, then the tasks spawned at lines 5 and 6 are abstracted to ob_3.wakeup and ob_2.sleeps, respectively. Now, we can see that there exists a cycle within the abstract dependency graph which is marked with bold edges. The cycle found in Example 1 is abstracted to this cycle, which includes the two blocked locations cl (here ob_3) and ba (here ob_2) and the three waiting tasks.

Given a program with a main procedure, the output of the analysis are the potential cycles (if any). But, as it can observed, it is complex to figure out from

them why these dependencies arise, and in particular the interleavings scheduled to lead to this situation. In our framework, we use these abstract cycles to guide the execution towards concrete cycles, representatives of the abstract ones or discard false positives if there is no execution containing a representative.

Now, we recall the theorem that guarantees the soundness of this analysis and whose proof can be found in [11]. We denote as $\overline{\alpha}$, the abstraction function applied over the nodes in a path of *deadlock dependencies graph*.

Definition 2 (deadlock soundness)**.** *Let S be a reachable state. If there is a cycle $\gamma = e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_1$ 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 $\mathcal{G}$.*

Furthermore, the following lemma is used in Theorem 1 and its proof can be also found in [11].

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

Finally, the accuracy of the analysis can be greater improved by means of *May-Happen-In-Parallel information* in order to discard unfeasible cycles whose tasks cannot happen simultaneously. The interested reader is referred to [11] for a detailed explanation.

3.2 Testing

Software testing is one of the techniques most widely used in practice in order to ensure the reliability of concurrent programs. The basic idea is to use some sample of the data that a program is expected to handle in order to test the functional behavior of the program. If the program produces correct results for the sample, it is assumed to be correct. Most current research focuses on the question of how to choose this sample.

We use the framework developed in [5] and its implementation aPET, a *non-random white-box tool*, which generates test cases using *symbolic execution*. In order to perform *testing*, we execute from a main method whose input parameters must be completely instantiated.

Our language adopts a non-deterministic semantics, as rule MSTEP in Figure 2.1 selects any location whose queue is not empty and, then, any pending task inside such queue can be chosen to be executed. As fields can be accessed by all tasks, different behaviors can occur depending on the order in which tasks are scheduled in the location and, thus, during testing all possible orderings must be tried.

Finally, our *testing tool* stops generating new test cases when a *coverage criterion* is achieved by the current test suite. A *coverage adequacy criterion* defines what properties of a program must be tested to constitute an *adequate* test, i.e., one whose successful execution implies no errors in a tested program. In Section 5.2, we will propose several criteria that define how good is a test suite when we are trying to detect deadlocks.

To illustrate this concept, we highlight two very common *criteria*: *statement coverage* that requires executing all the statements in the program under test and, similarly, *branch coverage* requires that all control transfers in the program are exercised during testing. The interested reader can find more examples of *coverage criteria* in [22].

```
27 Int a; // field
28 Int main(){
29   Int r = 0;
30   this.a = 0;
31   Fut<Int> g;
32   this ! toOdd();
33   this ! toEven();
34   g = this ! oddEven();
35   g.await;
36   r = g.block;
37   return r;
38 }
```

```
39 Unit toOdd(){
40   this.a = this.a*2+1;
41 }
42 Unit toEven(){
43   this.a = this.a*2;
44 }
45 Int oddEven(){
46   Int r = 0;
47   if (this.a == 0) r = 0;
48   else r = this.a mod 2;
49   return r;
50 }
```

Figure 3.1: Program with multiple outputs

Example 3. *Let us consider the methods in Figure 3.1 that belong to the same class, where a is a class field. In order to test this program, we execute the* main *method using the rules for the semantics in Figure 2.1 and trying all possible reorderings. Figure 3.2 summarizes the derivation tree, where the derivations that contain a gray node have already got the output, although the execution could be unfinished.*

When the execution arrives to the program point g.await*, the queue of tasks for location* **this** *will contain the three asynchronous calls* toOdd, toEven *and* oddEven*. Now, the current task in which the* g.await *is executing also has to go to the queue since the value of g is not ready.*

First, let us consider the leftmost derivation in Figure 3.2. Then, we suppose that toOdd *is selected to be executed. Then, it changes the value of field a to 1. Next,* toEven *updates the field with the value 2 and, finally,* oddEven *returns 0 as result, and it is stores in the future variable g. Now, the execution of the* await *can proceed and the method returns* $r = 0$ *as result. Once this execution has finished, a new test case is saved with the output and the initial and final states.*

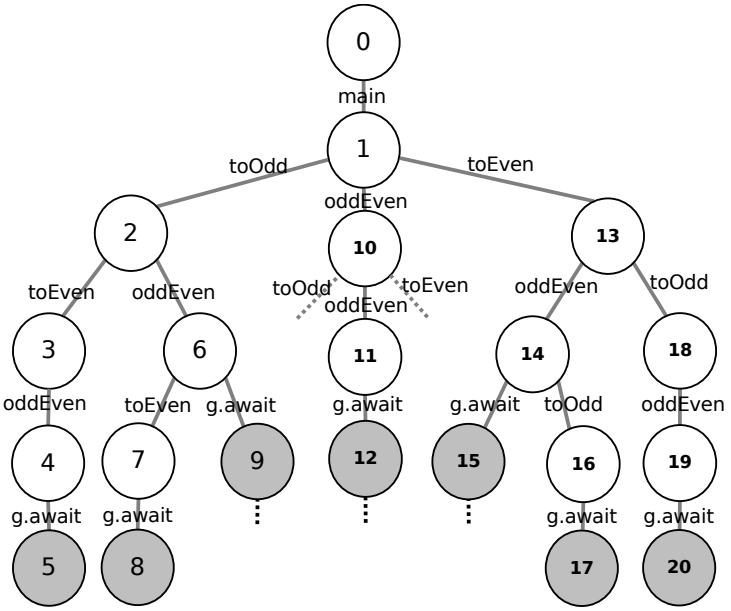


Figure 3.2: Execution Tree of program in Figure 3.1

$$\begin{array}{lcl} R & = & 0 \\ S_i & = & [(\mathsf{O}, \mathsf{loc}('\mathsf{main}', [\mathsf{field}(\mathsf{a}, 0)], []))] \\ S_f & = & [(\mathsf{O}, \mathsf{loc}('\mathsf{main}', [\mathsf{field}(\mathsf{a}, 2)], []))] \end{array}$$

Now, we consider the derivation ending at state 17: let us suppose that the first selected task is toEven *and, thus, the value of field a does not change. Then,* oddEven *returns 0 as result and, finally,* toOdd *updates the field a with the value 1. The following is the test case obtained:*

$$\begin{array}{lcl} R & = & 0 \\ S_i & = & [(\mathsf{O}, \mathsf{loc}('\mathsf{main}', [\mathsf{field}(\mathsf{a}, 0)], []))] \\ S_f & = & [(\mathsf{O}, \mathsf{loc}('\mathsf{main}', [\mathsf{field}(\mathsf{a}, 1)], []))] \end{array}$$

Now, let us consider these two test cases to be evaluated under the two criteria proposed previously. Then, we can observe this set does not achieve the statement coverage, as neither of the two executions tests the line 48. Analogously, both test cases evaluate the condition in line 47 to true *and, thus, a control transfer remains unexercised.*

Finally, let us consider the rightmost derivation, corresponding to the following execution order: $\mathsf{main}, \mathsf{toEven}, \mathsf{toOdd}, \mathsf{oddEven}$ *and* $\mathsf{g.await}$*, then we obtain this test case:*

$$\begin{array}{lcl} R & = & 1 \\ S_i & = & [(\mathsf{O}, \mathsf{loc}('\mathsf{main}', [\mathsf{field}(\mathsf{a}, 0)], [])] \\ S_f & = & [(\mathsf{O}, \mathsf{loc}('\mathsf{main}', [\mathsf{field}(\mathsf{a}, 1)], [])] \end{array}$$

The test suite that contains these three test cases does achieve both statement and branch coverage.

Chapter 4

Testing for Deadlock Detection

The goal of this chapter is to present a framework for early detection of deadlocks during testing. This is done by enhancing the standard semantics for asynchronous programs with information which allows us to easily detect *dependencies* among tasks, in Section 4.1. These dependencies are necessary to detect in a second step *deadlock states*, which is explained in Section 4.2. Chapter 5 also makes use of these dependencies in order to discard derivations that could not achieve them.

4.1 An Enhanced Semantics for Deadlock Detection

In the following we define the *interleavings table* whose role is threefold: (1) It stores all decisions about task interleavings made during the execution. This way, at the end of a concrete execution, the exact ordering of the performed macro-steps can be observed. (2) It will be used to detect deadlocks as early as possible, and, also to detect states from which a deadlock cannot occur, therefore allowing to prune the execution tree when we are looking for deadlocks. (3) Its times will be used to check *time and future constraints* discarding derivations that does not satisfy them. The interleavings table is a mapping with entries of the form $t_{id_o,id_t,pp} \mapsto \langle n, \rho \rangle$, where:

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

Figure 4.1: MSTEP2 rule for combined testing and analysis

- $t_{id_o,id_t,pp}$ is a *macro-step identifier*, or *time identifier*, that includes: the identifiers of the location id_o and task id_t that have been selected in the macro-step, and the program point pp of the first instruction that will be executed;
- n is a (non-negative) integer representing the time when the macro-step starts executing;
- ρ is the status of the task after the macro-step and it can take three values as it can be seen in Figure 2.1: block or await when executing these instructions on a future variable that is not ready (we also annotate in ρ the information on the associated future); **return** that allows us to know that the task finished. This allows reflecting task dependencies which will be necessary later to find deadlock cycles and find out when an execution is blocked.

We use a function ***clock*(*n*)** to represent a clock that starts at 0, is increased by one in every execution of ***clock***, and returns the current value **n**. The initial entry is $t_{0,0,1} \mapsto \langle 0, \rho_0 \rangle$, being 0 the identifier for the initial location and task, and 1 the first program point of *main*. The clock also assigns the value 0 as the first element in the tuple and a fresh variable in the the second element ρ_0. The next macro-step will be assigned clock value 1, next 2, and so on. As notation, we define the relation $t \in table$ if there exists an entry $t \mapsto \langle n, \rho \rangle \in table$, and the function $status(t, table)$ which returns the status ρ_t such that $t \mapsto \langle n, \rho_t \rangle \in table$.

The semantics is extended by changing rule MSTEP as in Figure 4.1. The function deadlock will be defined in Theorem 1 to stop derivations as soon as deadlock is detected. Function $\mathsf{check}_{\mathfrak{C}}$ should be ignored by now, it will be defined in Section 5.1. Essentially, there are two new aspects: (1) The state is extended with the status ρ, namely all rules include a status ρ attached to the state using the symbol $\diamond$. The status is showed in bold font in Figure 2.1 and can get a value in rules block2, await2 and return. The initial value ρ_0 is a fresh variable. (2) The state for the macrostep is extended with the interleavings table *table*, and a new entry $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$ is added to *table* in every macrostep if there has been progress in the execution, i.e., $S' \neq S$, being n the current clock time.

Example 4. *The interleavings table below is computed for the derivation in Section 2.2. It has as many entries as macro-steps in the derivation. We can observe that subsequent time values are assigned to each time identifier so that we can then know the order of execution. The right column shows the future variables in the state that store the location and task they are bound to.*

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

4.2 Formal Characterization of Deadlock State

Our semantics can easily be extended to detect deadlock just by redefining function *selectLoc* so that only locations that can proceed are selected. If, at a given state, no location is selected but there is at least a location with a non-empty queue then there is a deadlock. However, deadlocks can be detected earlier.

We present the notion of *deadlock state* which characterizes states that contain a *deadlock chain* in which one or more tasks are waiting for each other termination and none of them can make any progress. Note that, from a deadlock state, there might be tasks that keep on progressing until the deadlock is finally made explicit.

Even more, if one of those tasks runs into an infinite loop, the deadlock will not be captured using this naive extension. The early detection of deadlocks is crucial to reduce state exploration as our experiments show in Section 6.2.

We first introduce the auxiliary notion of *waiting interval* which captures the period in which a task is waiting for another one to terminate. In particular, it is defined as a tuple $(t_{stop}, t_{async}, t_{resume})$ where t_{stop} is the macro-step at which the location stops executing a task due to some block/await instruction, t_{async} is the macro-step at which the task that is being awaited is selected for execution, and, t_{resume} is the macro-step at which the task will resume its execution. t_{stop}, t_{async} and t_{resume} are time identifiers as defined in Section 4.1. t_{resume} will also be written as $next(t_{stop})$. When the task stops at t_{stop} due to a block instruction, we call it *blocking interval*, as the location remains blocked between t_{stop} and $next(t_{stop})$ until the awaited task, selected in t_{async}, has already finished.

The execution of a task can have several points at which macro-steps are performed (e.g., if it contains several await or block the processor may be lost several times). For this reason, we define the set of successor macro-steps of the same task from a macro-step: $suc(t_{o,tk,pp_0}, table) = \{t_{o,tk,pp_i} : t_{o,tk,pp_i} \in table, t_{o,tk,pp_i} \geq t_{o,tk,pp_0}\}$.

Definition 3 (Waiting/Blocking Intervals)**.** *Let $St = (S, table)$ be a state, $I = (t_{stop}, t_{async}, t_{resume})$ is a waiting interval of St, written as $I \in St$, iff:*

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

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

In condition 3, we can see that if the task starting at t_{async} has finished, then it is not a waiting interval. This is known by checking that this task has not reached return, i.e., $\nexists\ t \in suc(t_{async}, table)$ such that $status(t) = return$. In

condition 1, we see that in ρ_{stop} we have the name of the future we are awaiting (whose corresponding information is stored in fut, condition 2). In order to define t_{resume} in condition 2, we search for the same task tk_0 and same location o that executes the task starting at program point pp_1 of the await/block, since this is the point that the macro-step rule uses to define the macro-step identifier t_{o,tk_0,pp_1} associated to the resumption of the waiting task.

Example 5. *Let us consider again the derivation in Section 2.2. We have the following blocking interval* $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_1$ *with* $St_1 \equiv (S_1, table_1)$, *since* $t_{cl,wakeup,21} \in table_1$, $status(t_{cl,wakeup,21}, table_1) = [24{:}f.\mathsf{block}]$, $(f, ba, cuts, 12) \in St_1$ *and* $t_{ba,cuts,12} \notin table_1$. *This blocking interval captures the fact that the task at* $t_{cl,wakeup,21}$ *is blocked waiting for task cuts to terminate. Similarly, we have the following two intervals in* St_6: $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$ *and* $(t_{ch,taken,15}, t_{cl,sits,25}, t_{ch,taken,17})$, *which intuitively capture that the barber ba is waiting that the chair ch is taken, and in turn the chair ch is waiting that the client cl sits.*

The following notion of *deadlock chain* relies on the waiting/blocking intervals of Definition 3 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 4 (Deadlock Chain)**.** *Let* $St = (S, table)$ *be a state. A chain of time identifiers* $t_0, ..., t_n$ *is a deadlock chain in* St, *written as* $dc(t_0, ..., t_n)$ *iff* $\forall t_i \in \{t_0, ..., t_{n-1}\}$ *s.t.* $(t_i, t'_{i+1}, next(t_i)) \in St$ *one of the following conditions holds:*

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

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

Let us explain the two conditions in the above definition: In condition (1), we check that when a task t_i is waiting for another task to terminate, the waiting interval contains the initial time t'_{i+1} in which the task will be selected. However, we look for any waiting interval for this task t_{i+1} (thus we check that t_{i+1} is a successor of time t'_{i+1}). As in Definition 5, this is because such task may have started its execution and then suspended due to a subsequent await/block instruction.

Abusing terminology, we use the time identifier to refer to the task executing. In condition (2), we capture deadlock chains which occur when a task t_i is waiting on the termination of another task t'_{i+1} which executes on a location $loc(t'_{i+1})$ which is blocked. The fact that is blocked is captured by checking that there is a blocking interval from a task t_{i+1} executing on this location. Finally, note that the circularity of the chain, since we require that $t_{n+1} \equiv t_0$.

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

Derivations ending in a deadlock state are considered complete derivations. We prove that our definition of deadlock is equivalent to the standard definition of deadlock in [11, 9]. To do so, we define a function γ that transforms one-to-one a *deadlock chain* into a cycle in G_S.

Definition 5 (γ)**.** *Given a state St=(S, table) and a sequence of times* $\{t_0, ..., t_n\}$ *in St, satisfying (1) or (2) in Definition 4. The one-to-one function* $\gamma(\{t_0, ...t_n\})$=$e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$ *in* G_S *is defined as follows:*

$$\gamma(\{t_0, ..., t_n\}) = \begin{cases} \{loc(t_0) \rightarrow tsk(t_1)\} \cup \gamma_{tk}(\{t_1, ..., t_n\}) & \text{if } t_0 \text{ holds (1)} \\ \{loc(t_0) \rightarrow tsk(t'_1) \rightarrow loc(t'_1)\} \cup \gamma(\{t_1, ..., t_n\}) & \text{if } t_0 \text{ holds (2)} \wedge \neg(1) \end{cases}$$

where γ_{tk} *is the following auxiliary function:*

$$\gamma_{tk}(\{t_0, ..., t_n\}) = \begin{cases} \{tsk(t_0) \rightarrow tsk(t_1)\} \cup \gamma_{tk}(\{t_1, ..., t_n\}) & \text{if } t_0 \text{ holds (1)} \\ \{tsk(t_0) \rightarrow tsk(t'_1) \rightarrow loc(t'_1)\} \cup \gamma(\{t_1, ..., t_n\}) & \text{if } t_0 \text{ holds (2)} \wedge \neg(1) \end{cases}$$

We need to distinguish between functions γ and γ_{tk}, as in [11] a location blocked in a task could be represented in G_S by both the location identifier and

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

Theorem 2 (Deadlock Equivalence)**.** *Let St be a program state,*

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

Proof.
$\Rightarrow$ Let $dc(\{t_0, ..., t_n\})$ be a deadlock chain, then we could apply the function γ, as $\forall t_i \in \{t_0, ..., t_n\}$, t_i satisfies (1) or (2). So, we obtain a path in G_S and using the last condition in Definition 4, both $\gamma(\{t_n\})$ and $\gamma_{tk}(\{t_n\})$ add the edge $tk(t'_0) \rightarrow loc(t_0)$ and, thus, the path becomes a cycle.
$\Leftarrow$ Given a cycle Γ in G_S, by Lemma 1, it contains at least one location node, which is required by the function γ. As γ is a one-to-one function, $\exists \gamma^{-1}$, which is applied to Γ and, easily, we obtain the result. □

Example 6. *Following Example 4, St_6 is a deadlock state since there exists a deadlock chain $dc(t_{cl,wakeup,21}, t_{ba,sleeps,9}, t_{ch,taken,15})$. For the second element in the chain $t_{ba,sleeps,9}$, condition 1 holds as $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11}) \in St_6$ and $t_{ch,taken,15} \in suc(t_{ch,taken,15}, table_6)$. For the first element $t_{cl,wakeup,21}$, condition 2 holds since $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_6$ and $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$ is blocking. Condition 2 holds analogously for $t_{ch,taken,15}$.*

Chapter 5

Combining Deadlock Analysis and Testing

This chapter proposes a deadlock detection methodology that combines static analysis and testing as follows. First, a state-of-the-art deadlock analysis is run, in particular that of [11], which is presented in Section 3.1. For the sake of simplicity of the presentation, we assume parameter $k = 0$ in the formalization, (our implementation uses $k = 1$). If the set is empty, then the program is deadlock-free. Otherwise, using the inferred set of deadlock cycles, we test the program using our enhanced semantics with two goals: (1) finding concrete deadlock traces associated to the different cycles, and, (2) discarding deadlock cycles, and in case all cycles are discarded, ensure deadlock freedom for the considered input or, in our case, for the main method under test, we present this technique in Section 5.1. At the end, in Section 5.2, we propose several coverage criteria, based on our methodology, which give us different level of information about the deadlocks contained by the program tested.

5.1 Guiding Testing towards Deadlock Cycles

Given an abstract deadlock cycle, we now present a novel technique to guide the execution towards paths that might contain a representative of that abstract

deadlock cycle, by discarding paths that are guaranteed not to contain such a representative.

The main idea is as follows: (1) From the abstract deadlock cycle, we generate *deadlock-cycle constraints*, which must hold in all states of derivations leading to the given deadlock cycle. (2) We extend the execution semantics to support deadlock-cycle constraints, with the aim of stopping derivations as soon as cycle-constraints are not satisfied. Uppercase letters in constraints denote variables to allow representing incomplete information.

Definition 6 (Deadlock-cycle constraints)**.** *Given a state* $St = (S, table)$*, a deadlock-cycle constraint takes one of the following three forms:*

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

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

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

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

Notation $\mathsf{sync}(p_i, F_i)$ is a shortcut for $p_i{:}F_i.\mathsf{block}$ or $p_i{:}F_i.\mathsf{await}$. Uppercase letters appearing for the first time in the constraints are fresh variables. The first case handles location-task and task-task arrows, since e_j is a task abstraction; whereas the second case handles task-location arrows, because e_j is an abstract location.

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

Example 7. *In our working example there are three abstract locations, ob_2, ob_3 and ob_4, corresponding to locations* barber, client *and* chair, *created at lines* 2, 3 *and* 4*; and six abstract tasks, sleeps, cuts, wakeup, sits, taken and isClean, as we have seen in Example 1. Let us observe here that the abstract tasks involved are different to the ones in Example 2 due to the accuracy of the analysis.*
The following cycle is inferred by the deadlock analysis: $o_2 \xrightarrow{11:sleeps} taken \xrightarrow{17:taken} sits \xrightarrow{25:sits} o_3 \xrightarrow{24:wakeup} cuts \xrightarrow{12:cuts} o_2$.
The first arrow captures that the location created at line 2 is blocked waiting for the termination of task taken *because of the synchronization at line 11 of task* sleeps.

Observe that cycles contain dependencies also between tasks, like the second arrow, where we capture that taken *is waiting for* sits. *Also, a dependency between a task (e.g.,* sits*) and a location (e.g., o_3) captures that the task is trying to execute on that (possibly) blocked location.*

Now, the deadlock-cycle constraints computed for this cycle are:
$\{\exists\ t_{O_1,Tk_1,\_} \mapsto \langle \_, 11{:}F_1.block\rangle, \exists fut(F_1, O_2, Tk_2, 15),\ \exists t_{O_2,Tk_2,\_} \mapsto \langle \_, 17{:}F_2.await\rangle,$
$\exists\ fut(F_2, O_3, Tk_3, 25),\ \mathsf{pending}(Tk_3),\ \exists t_{O_3,Tk_4,\_} \mapsto \langle \_, 24{:}F_3.block\rangle$
$\exists fut(F_3, O_4, Tk_5, 12), \mathsf{pending}(Tk_5)\}$. *They are shown in the order in which they*

are computed by ϕ. The first four constraints require table to contain a concrete time in which some barber sleeps waiting at line 11 for a certain chair to be taken at line 15 and, during another concrete time, this one waits at line 17 for a certain client to sit at line 25. The client is not allowed to sit by the 5th constraint. Furthermore, the last three constraints require a concrete time in which this client waits at line 24 to get a haircut by some barber at line 12 and that haircut is never performed.

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

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

Definition 8. *Given a set of deadlock-cycle constraints $\mathfrak{C}$, and a state $St = (S, table)$, check holds, written $\mathsf{check}_{\mathfrak{C}}(St)$, if $\forall t_{O_i,Tk_i,PP} \mapsto \langle N, \mathsf{sync}(p_i, F_i)\rangle \in \mathfrak{C}$, $fut(F_i, O_j, Tk_j, p_j) \in \mathfrak{C}$, one of the following conditions holds:*

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

Function reachable checks whether a given task might arise in subsequent states. We over-approximate it syntactically by computing the transitive call relations from all tasks in the queues of all locations in S. Precision could be improved using more advanced analyses. Function getTskSeq gets from the state

the sequence of instructions to be executed by a task (which is ϵ if the task has terminated).

Intuitively, check does not hold if there is at least a time constraint so that: (i) its time identifier is not reachable, and, (ii) in the case that the interleavings table contains entries matching it, for each one, there is an associated future variable in the state and a pending constraint for its associated task which is violated, i.e., the associated task has finished.

The first condition (i) implies that there cannot be more representatives of the given abstract cycle in subsequent states, therefore if there are potential deadlock cycles, the associated time identifiers must be in the interleavings table.

The second condition (ii) implies that, for each concrete potential cycle in the state, there is no deadlock chain since at least one of the blocking tasks has finished. This means there cannot be derivations from this state leading to the given deadlock cycle, therefore this derivation can be stopped. Function $\mathsf{check}_{\mathfrak{C}}$ is used in the semantics to prune deadlock-free derivations as showed in Figure 4.1.

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

Definition 9 (Deadlock-cycle guided-testing (DCGT))**.** *Consider an abstract deadlock cycle c, and an initial state St_0. Let $\mathfrak{C} = \phi(c, O_{init}, Tk_{init})$ with O_{init}, Tk_{init} fresh variables. We define DCGT, written $exec_c(St_0)$, as the set $\{d : d \in exec(St_0), \mathsf{deadlock}(St_n)\}$, where St_n is the last state in d.*

Example 8. *Let us consider the DCGT of our working example with the deadlock-cycle and the constraints $\mathfrak{C}$ of Example 7. The interleavings table at St_5 contains the entries $t_{ini,main,1} \mapsto \langle 1, return \rangle$, $t_{cl,wakeup,21} \mapsto \langle 2, 24{:}f_0.block \rangle$ and $t_{ba,cuts,12} \mapsto \langle 3, return \rangle\}$. $\mathsf{check}_{\mathfrak{C}}$ does not hold since $t_{O_1,Tk_1,24}$ is not reachable from St_5 and constraint $\mathsf{pending}(Tk_5)$ is violated (task cuts has already finished at this point). The derivation is hence pruned. Similarly, the rightmost derivation is stopped at St_{11}. Also, derivations at St_6, St_{14} and St_{18} are stopped by function deadlock of Theorem 1. Our deadlock guided testing methodology generates* 16 *states instead of the* 181 *generated by the standard exhaustive execution.*

Theorem 3 (Soundness)**.** *Given a program P, a set of abstract cycles C in P and an initial state* St_0, $\forall d \in exec(St_0)$ *if d is a derivation whose last state is deadlock, then* $\exists c \in C$ *such that* $d \in exec_c(St_0)$.

Proof of Theorem 3. If the last state is deadlock, then $\exists dc(\{t_0, ..., t_n\}) \in St_n$, by Theorem 1. Using the soundness of deadlock analysis (by Definition 2) over the cycle $\gamma(\{t_0, ..., t_n\})$, the existence of an abstract cycle $c \in C$ is ensured. Now, by Lemma 2, we obtain the result. □

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

Proof. By contradiction, let us suppose that $\exists d \in exec(St_0)$ and $d \notin exec_c(St_0)$. Hence, $\exists St_i \in d$ such that $\mathsf{check}_{\mathfrak{C}}(St_i)$ returns false and, consequently, the derivation $St_0 \longrightarrow^* St_i$ stops, where $\mathfrak{C} = \phi(c, O, Tk)$ and O, Tk are fresh variables. Therefore, at St_i $\exists\{t_{O_i,Tk_i,PP} \mapsto \langle N, \mathsf{sync}(p_i, F_i)\rangle, fut(F_i, O_j, Tk_j, p_j)\} \subset \mathfrak{C}$ that does not satisfy either (1) or (2) in Definition 8. However, it is not possible, as $\mathfrak{C}$ imposes necessary constraints for the existence of some representative of c and St_n contains a cycle that is a representative of c, then (1) or (2) must be satisfied at every state of d and, in particular, at St_i. As a result, we get a contradiction. □

5.2 Deadlock-based Testing Criteria

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

- first-deadlock, which requires exercising at least one deadlock execution,

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

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

Chapter 6

Implementation and Experiments

The chapter reports on some aspects of the real implementation that were not completely specified in the general description of the framework and summarizes the experiments we have performed. In particular: in Section 6.1.1, we present the handling of the interleavings table; Section 6.1.2 explains the checking of the two prunings: the existence of a *deadlock chain* in the current state and the satisfiability of *Deadlock-cycle constraints*; finally, in Section 6.1.3, we detail how the different coverage criteria have been implemented. Section 6.2 summarizes the experimental results and the main conclusions about the meaning of these experiments.

6.1 Implementation Details

We have implemented our approach within the tool aPET, presented in Section 3.2, which is available at `http://costa.ls.fi.upm.es/apet`. This tool is written in *Prolog* and is based on the *constraint logic programming paradigm*, in which *logic programming* is extended to include concepts from *constraint satisfaction*.

Such webpage also contains the benchmarks in Section 6.2, which are written in ABS language. ABS is a actor-based language [15] that targets *distributed systems* by means of *concurrent object groups* and *asynchronous methods calls* and supports a range of techniques for *model exploration* and *analysis*. The

language in Section 2.1 fully captures the main features of ABS.

6.1.1 Enhanced State and Interleavings Table

In order to make the semantics *object-sensitive* and, thus, get more accurate prunings, we add to the term *loc* in Section 2.1 a new argument *anc* which keeps track its ancestor locations. When a **new** instruction is executed, a new location is created and receives as ancestors list its parent plus the parent's ancestors list. Once the parameter k of the *deadlock analysis* is fixed, the relevant ancestors for the semantics are the first k locations in such list.

In the implementation, the interleavings table in Section 4.1 is represented as a list of $t/4$. Every entry in *table* is now a term $t/4$ that stores the time, the current location and task identifiers and the status. This status stores relevant information by means of two statements: prod(f) that indicates that the current task produces a value collected by the future f and waits(pp,f), that saves the program point pp and the future variable f where the current task stops. Therefore, the futures are not taken explicitly by the state but by the tasks which produce them.

The interleavings table is modified by the relevant instructions as follows:

- When the last instruction executed is an asynchronous call, we add a new term t to the interleavings table right after the table head. The location and task identifiers are the corresponding to the current call, the status is partially instantiated by prod(f), where f is the corresponding future and the time is a variable which will be instantiated at the moment of its execution.
- When a task is selected to be executed, its corresponding term t is brought to the head of the table. Thus, the following accesses to update the status are done in amortized constant time, even when the table is arbitrarily large. Furthermore, the time is instantiated to the next number by solving the constraint imposed over this variable.
- When the current task gets blocked in an await or block instruction, we

modify the status in the table head with waits(pp,f) where pp is the blocking instruction's program point and f is the waited future variable.

As a result of these rules, the interleavings table is always sorted in decreasing order. Furthermore, a complete derivation is deadlock if and only if the last interleavings table contains a $t/4$ whose time remains variable.

Example 9. *Let us see again how the interleavings table progresses along the derivation ending at node 6. For the sake of clarity, we tag the arrows with the line numbers that produce the change in the table.*

$$
\begin{aligned}
&tab_0 \equiv [t(0, ini, ini, [prod(f_0)|X_0])] \xrightarrow{5,6} \\
&tab_1 \equiv [t(0, ini, ini, [prod(f_0)]), t(I_1, cl, wk, [prod(f_1)|X_1]), t(I_2, ba, sl, [prod(f_2)|X_2])] \xrightarrow{21} \\
&tab_2 \equiv [t(1, cl, wk, [prod(f_1)|X_1]), t(0, ini, ini, [prod(f_0)]), t(I_2, ba, sl, [prod(f_2)|X_2])] \xrightarrow{22,23} \\
&tab_3 \equiv [t(1, cl, wk, [prod(f_1)|X_1]), t(I_3, ba, cut, [prod(f_3)|X_3]), t(I_4, ch, cle, [prod(f_4)|X_4]), ...] \xrightarrow{24} \\
&tab_4 \equiv [t(1, cl, wk, [prod(f_1), waits(24, f_3)]), t(I_3, ba, cut, [prod(f_3)|X_3]), t(I_4, ch, cle, ..), ...] \xrightarrow{10,11} \\
&tab_5 \equiv [t(2, ba, sl, [prod(f_2), waits(11, f_5)]), t(I_5, ch, tk, [prod(f_5)|X_5]), t(1, cl, wk, ..), ...] \xrightarrow{16,17} \\
&tab_6 \equiv [t(3, ch, tk, [prod(f_5), waits(17, f_6)]), t(I_6, cl, sit, [prod(f_6)|X_6]), \\
&\qquad t(2, ba, sl, [prod(f_2), waits(11, f_5)]), t(1, cl, wk, [prod(f_1), waits(24, f_3)]), \\
&\qquad t(I_3, ba, cut, [prod(f_3)|X_3]), t(I_4, ch, cle, [prod(f_4)|X_4]), t(0, ini, ini, [prod(f_0)])]
\end{aligned}
$$

We can observe here that tab_6 is sorted decreasingly and those asynchronous calls that have not been executed are always preceded by the time which spawned them. Indeed, we could obtain the execution trace by filtering out from this list those $t/4's$ whose time is variable.

6.1.2 Checking prunings

In order to reduce the exploration of the search space and improve the testing scalability in concurrent programs, we have proposed two prunings that try avoiding useless exploration: (1) *Early Deadlock Detection*, which stops a derivation when the most recent explored state is deadlock, and (2) *Deadlock-Cycle Constraints Solver*, which detects if the current derivation could reach a deadlock state.

Early Deadlock Detection

The goal of the first pruning is the detection of a deadlock state as soon as possible. However, performing this checking at each state can add an important overhead on the execution time. Thus, we just do it on *dangerous states*. We consider a state as *dangerous* if the last time's task in the interleavings table stopped its execution due to an await or block instruction. The meaning of this simple heuristics is that, by Definition 4, the last time could be involved in a deadlock chain if its execution finishes due to one of these two instructions. We perform the following actions:

1. We compute the waiting and blocking intervals contained by the state. To do so, we process in increasing order the interleavings table and filter those times whose index is variable. Now, for each time finished in an await or block instruction, we build an interval, as in Definition 3. The result of this action is a set with every interval contained in the state sorted increasingly. The cost of this computation is linear in the table size.

2. We compute the *blocking chains* by concatenation of the previous intervals. Basically, a blocking chain is a subchain of *deadlock chain* whose times hold the property (1) in Definition 4. For each computed blocking interval, we build an initial blocking chain with its t_{stop}. If the next interval holds that its t'_{stop} is a successor of t_{async}, then we build a new blocking chain as a result of the concatenation of the previous one and t'_{stop}, and so on.

3. We build a graph whose nodes are the locations and there exists a directed edge between two of them if the first one is the initial time 's location in a *blocking chain* which ends in a time whose location is the second node. If this graph contains a cycle, then we have detected a deadlock and, thus, the derivation is stopped saving the exploration of the derivation subtree produced by the pending tasks.

Example 10. *We perform the three previous actions on the* tab_6 *of Example 9. During the first action, we reduce the table:* $[t(0, ini, ini, [prod(f_0)]), t(1, cl, wk, [prod(f_1), waits(24, f_3)]), t(2, ba, sl, [prod(f_2), waits(11, f_5)]), t(3, ch, tk, [prod(f_5), waits(17, f_6)])]$.

Now, we compute the set of intervals, that we shorten as $\mathsf{wait}/\mathsf{block}(t_{stop}, t_{async})$*, because* t_{resume} *is not relevant at this point. Therefore, the result of the first action is:*

$$\{\mathsf{block}(1, I_3), \mathsf{block}(2, 3), \mathsf{wait}(3, I_6)\}$$

During the second action, we obtain three blocking chains that denote as $\mathsf{blchain}(a,b)$ *where* a *and* b *are the initial and final locations, respectively.*

$$\{\mathsf{blchain}(cl, ba), \mathsf{blchain}(ba, ch), \mathsf{blchain}(ba, cl)\}$$

The first element of this set is obtained due to the first interval. In a similar way, we obtain the second one. The last blocking chain generated is result of concatenating the two last intervals, which satisfies the property (1) in Definition 4.

Performing the last action, we obtain a graph with three nodes: cl, ba, ch *and three edges:* $cl \to ba, ba \to ch, ba \to cl$*. Therefore, there exists a cycle between* cl *and* ba*, indicating this is a deadlock state.*

Deadlock-Cycle Constraints Solver

The second pruning is the satisfiability of the *Deadlock-Cycle Constraints*. In order to get a deadlock state, it is necessary that every constraint holds at each state. Therefore, we check if some *time constraint* is unsatisfiable with the current interleavings table to stop the derivation as soon as possible. This fact happens depending on multiple factors like conditions, orderings or, even the input values and, thus, we cannot define a heuristics as simple as the one in the previous section. Therefore, we perform this checking immediately after exploring a new state. This adds an important overhead on the execution time but it can be reduced if we check it periodically.

The *Deadlock-Cycle Constraints* can be understood as a set of *blocking chain constraints*. This concept generalises the previous *blocking chain* as follows: a *blocking chain constraint* concatenates *time constraints* linked by means of *fut*

constraints until that a *pending constraint* is reached. Thus, we will stop the execution if these *chain constraints* cannot happen simultaneously in any state.

In Section 3.1, we comment that both the deadlock analysis and the semantics can be made *object-sensitive* and, indeed, our implementation uses $k = 1$, where k is the precision parameter. To do so, the ϕ-generated *time constraints* (Section 5.1) keep track the list of abstract ancestor locations inside the variable location X. Now, we define a *compatibility relation* between *time constraints* and the times within the interleavings table as follows: a *time constraint* and a concrete time are *compatible* if and only if the program point contained by both times' status is the same and the ancestors of the concrete location have been created in the lines that indicate the abstract ancestors inside the *time constraint*.

Using the previous definitions, let us explain how this second pruning is checked. For each *blocking chain constraint*, we perform the following actions: (1) check if there exists some time in interleavings table that is compatible with the first *time constraint* in the *chain constraint*. (2) If so, then we compute the concrete *blocking chain* starting in such time and (3) check that the *blocking chain constraint* and the concrete *blocking chain* are compatible time to time.

Nevertheless, any *time constraint* could be incompatible with every time within the interleavings table. In order to stop the derivation safely, we need to prove that the interleavings table cannot eventually contain a new time which becomes compatible. To do so, we inspect the locations' queue looking for pending tasks that, by means of transitive calls, could spawn a task that stops at the program point indicated by *time constraint*'s status.

6.1.3 Coverage Criteria

Section 5.2 defines three coverage criteria depending on the user interests. In the following we give details about the implementation of each of them.

First-Deadlock

The first-deadlock criterion requires exercising at least one deadlock execution. To do so, we perform the following steps: (1) we choose one of the abstract cycles and (2) generate its *deadlock-guided constraints*. Then, (3) we perform the *dynamic testing* that follows the enhanced semantics using the rule MSTEP, which is redefined in Section 4.1. Finally, (4) the DCGT stops as soon as a deadlock state is reached, thanks to the *early detection of deadlocks* (Section 4.2) and, thus, (5) we use this execution as the new test suite that achieves the adequacy criterion.

There are alternative implementations of this scheme. In order to take advantage of the independency among DCGTs with different abstract cycles, we could (i) execute in parallel a DCGT for each detected abstract cycle, (ii) when one of the DCGTs finds a deadlock, we send a stop signal to the others and, finally, (iii) we use the deadlock derivation as the new test suite. However, it looks like too extravagant running as much DCGTs as abstract cycles in order to find only the *first* deadlock.

The main disadvantage of these two implementations is that the dynamic testing is focused on finding a representative of a specific cycle. Therefore, the last proposed implementation is running the dynamic testing without $\mathsf{check}_{\mathfrak{C}}$ in rule MSTEP.

The most effective scheme depends on the program to be tested, we highlight the first implementation as it looks like the most well-adjusted and performs better on the benchmarks in Section 6.2.

All-Deadlocks

The all-deadlocks criterion requires exercising all deadlock executions. We perform the following steps: (1) we run in parallel a DCGT for each abstract cycle until they have finished. Then, (2) the final test suite is the join of test cases generated by every DCGT. When we use this criterion, the testing scalability is hardly ever improved, as few branches can be pruned safely. Let us notice here that we cannot use the early detection of deadlocks, because it could prune derivations that can have more than one deadlock.

Deadlock-Per-Cycle

The deadlock-per-cycle criterion requires finding a deadlock representing the given cycle (if exists). Now, let us define the concept of *representation*: a deadlock state represents an abstract cycle if every constraint generated by the cycle satisfies the condition 2 in Definition 8, i.e., every *time constraint* is compatible with some time within the interleavings table.

In order to implement this scheme, (1) we run in parallel for every abstract cycle. (2) When a derivation can be pruned by early detection, we check if the deadlock state is a representative of the abstract cycle. (3) If so, then we stop the execution of this DCGT. (3') On the other hand, this branch is not pruned because a deadlock state is still reachable. Finally, (4) the test suite is the join of the test cases found by the DCGT's.

6.2 Experimental Evaluation

This section summarizes our experimental results which aim at demonstrating the applicability, effectiveness and impact of the proposed techniques during testing. The experiments have been performed using as benchmarks: (i) classical concurrency patterns containing deadlocks, namely *SB* is an extension of the sleeping barber with several clients, *UL* is a loop that creates asynchronous tasks and locations, *PA* the pairing problem, *FA* is a distributed factorial, *WM* making a water molecule, *HB* the hungry birds problem, and, (ii) deadlock free versions of some of the above, named *fX* for the *X* problem, for which deadlock analyzers give false positives. We include here a peer-to-peer system *P2P*.

Table 6.1 shows the results obtained using three different settings: (1) the first set of columns **Exh** corresponds to building the whole search tree, (2) the second to the first-deadlock criterion, and (3) the third to the deadlock-per-cycle criterion. For each setting i, we measure the total time taken (column T_i) and the number of states generated (column S_i). Column *Ans* contains the solutions obtained by the whole execution tree. Column *D/F/C* in the third setting shows "number of deadlock executions" / "number of unfeasible cycles" / "number of abstract cycles" found by the analysis. For instance, for *HB* we have 2/3/5 that shows that the analysis has found five abstract cycles, but we only found a deadlock execution for two of them, therefore 3 of them were unfeasible. Since the DCGTs in setting 3 can be performed in parallel, columns T_{max} and S_{max} show the maximum time and number of states measured among all of them. Columns in **S-up** show the gain of setting 3 w.r.t. 1 computed as $T_{up} = T_1/T_{max}$ (the gain is ∞ when T_1 is ∞ and T_{max} is not, or none "−" when T_{max} is ∞ too), and analogously for states. Times are in milliseconds and are obtained on an Intel(R) Core(TM) i7 CPU at 2.3GHz with 8GB of RAM, running Mac OS X 10.8.5. A timeout of 150.000ms (written 150k) is used. When the timeout is reached we write ∞.

When comparing setting 2 w.r.t. 1, we see that, when we only want to find the first deadlock trace and the program features a deadlock, our guided-testing is very effective, e.g., by just exploring 6 states in 40ms the deadlock is found in

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

Table 6.1: Experimental evaluation

PA. When the program is deadlock free, we need to explore the whole execution also in setting 2. Although the (spurious) information provided by the analysis does not allow much pruning in these cases, still there is a notable gain (e.g., in fPA we explore about one third of the states explored in setting 1 and the time is almost halved). Importantly, we are able to prove deadlock freedom in all examples while exhaustive exploration times out in fUL.

As regards setting 3, we achieve significant gains w.r.t. exhaustive exploration for deadlock-free examples (e.g., by just exploring 23 states in SB we found one representative per cycle in 59ms. while setting 1 times out). The gains are much larger in the examples in which the deadlock analysis does not give false positives (namely, in SB, UL, PA). For WM, we have failed to find a representative of a potential cycle within the timeout. This is because every abstract cycle produces different constraints, some of them allow important pruning during testing as they impose very restrictive conditions, whereas others can hardly guide because most of derivations fulfill the constraints. When this happens, the number of states explored is slightly smaller than with exhaustive execution. However, when we consider that each DCGT is computed in parallel for each cycle (columns **S-up**), we achieve further gains (in SB, UL, HB and PA we decrease the time notably) and in WP we perform slightly better than in setting 1.

Finally, for the examples that are deadlock free, the number of explored states for settings 2 and 3 is the same. This is because in order to ensure that a deadlock representative cannot be found, it is necessary to make exhaustive exploration with every abstract cycle. All in all, we argue that our experiments show that our methodology is very effective for programs that contain deadlock, and it is able also to prove deadlock freedom for some cases in which a static analysis reports false positives.

Chapter 7

Related Work and Conclusions

7.1 Related Work

There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [11, 12, 9] and thread-based programs [17, 19], is based on static analysis techniques. Static analysis can ensure the absence of errors, however it works on approximations (especially for handling iteration and pointer aliasing) which might lead to a "don't know" answer. Deadlock detection has been also studied in the context of dynamic testing and model checking [16, 14, 8, 7], where sometimes has been combined with static information [13, 2].

- The approach in [16] instruments the program's binary at specifics locations such that lock acquisitions, release calls. Then, a *Interference Engine* gathers information about the program's locking behavior by keeping track of the threads and the locks that they hold. They define several algorithms that decide when and how to interfere with scheduling. *Simple Preemption* simply preempts any thread exactly before trying to acquire a lock and exactly after releasing the lock. As a result, this algorithm gets a low overhead. On the other hand, *Component Based Delays* creates a run-time lock order graph and keeps it updated with every execution. The nodes of this graph represents the locks and a directed edge means that a thread that

holds the initial lock attempts to acquire the final one. Each directed loop in the graph corresponds to a potential deadlock. At the beginning of the algorithm, one of these loops is chosen and the *Interference Engine* tries to make threads circularly dependent on each other by delaying any thread holding at least one lock from the loop. To be more accurate, a algorithm variant uses the topological order to decide if the delay may be avoided.

One of the greatest advantages of this algorithm is the usage of the information from previous executions. However, if the *run-time lock order graph* has not been trained enough, then the *Interference Engine* could have imprecise information and the algorithm could miss deadlocks.

- As regards combined approaches, the approach in [14] presents the algorithm *DeadlockFuzzer* that finds real deadlocks in multi-threaded programs. It works in two phases: in the first one, it uses a predictive dynamic algorithm, called *iGoodlock*, which observes an execution and identifies potential deadlock cycles even if the execution does not deadlock. The second phase receives an initial state and a potential deadlock cycle, *DeadlockFuzzer* executes the program using a random scheduler and pauses a thread when is about to acquire a lock l if l is present in the cycle. Between both phases, the potential deadlock cycle detected in the first one has to relate to the initial cycle in the second one by means of an abstraction. To do so, they propose two object abstractions: *k-object-sensitivity*, the same as we do, and *light-wight execution indexing*, by using the call stack.

  The way they use the abstract cycle is completely different to ours. The cycle is used to impose conditions that can cause a deadlock with a high probability, losing the completeness. Whereas we use this information to give necessary conditions discarding those paths that does not hold them. A limitation of *iGoodlock* is that it can give false positives because it does not consider the happens-before relation between the transitions in an execution. We could take into account to our implementation the last abstraction

that they propose in order to compare which prunes more derivations and, thus, performs better on our benchmarks.

- The approach in [13] describes an algorithm, *CheckMate*, that requires users to annotate the predicate with each synchronization variable in a Java program relevant to finding deadlocks. First, *CheckMate* observes a complete execution and, by means of the predicate annotations, creates a *trace program* that only keeps the instructions that are relevant for deadlock. In a second step, *CheckMate* uses an off-the-shelf model checker to explore all possible thread interleavings of the trace program and check if any of them is deadlock.

  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), this method can detect deadlocks that do not exist in the program, while in our case this is not possible since the testing is performed on the original program and the analysis information is only used to drive the execution. Moreover, the trace program is generated by observing an isolated execution, then *CheckMate* does not track all control and data dependencies and it could miss deadlocks. This algorithm is neither complete nor sound.

- In [2], another version of algorithm *GoodLock* is used to create a run-time lock tree for each thread during execution. At the end, it constructs a *run-time lock-graph* using the lock trees and checks if the graph holds a property equivalent to the circular dependency between threads and locks, defined in [16]. In the last stage of the algorithm, a type system is used to infer order relations in order to accelerate the detection of potential cycles. There exist untypable programs that could contain deadlock and, however, this algorithm cannot find it.

  This work shares with our work that information inferred statically is used

to improve the performance of the testing tool, however there are important differences: first, their method developed for Java threads captures deadlocks due to the use of locks and cannot handle wait-notify, while our technique is not developed for specific patterns but rather works on a general characterization of deadlock of asynchronous programs; their underlying static analysis is a type inference algorithm which infers deadlock types and the checking algorithm needs to understand these types to take advantage of them, while we base our method on an analysis which infers descriptions of chains of tasks and a formal semantics is enriched to interpret them.

7.2 Conclusions and Future Work

We have proposed a framework for *guided* testing based on constraints that specify the order in which tasks interleave. Such constraints allow us to drive the execution towards paths that are more likely to lead to deadlock; additional contributions of our work are the deadlock-based testing criteria.

Our work complements static analysis techniques and can be used to look for deadlock paths when static analysis is not able to prove the absence of deadlock. Using our method, if there might be a deadlock, we try to find it by exploring the paths (possibly infinite) given by our deadlock detection algorithm that relies on the static information.

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

Finally, our semantics selects the next task to be executed non-deterministically. Our approach would perform even better if we enhance the semantics presented with a heuristics that chooses tasks that could potentially cause a deadlock. This

extension will require the selector to look at the tasks' instructions list those program points that are required by the *Deadlock-Cycle Constraints*. This subject also remains as future work.

Bibliography

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

[2] R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *C*onf. on Hardware and Software Verification and Testing, *L*NCS 3875, pages 191–207. Springer, 2006.

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

[4] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *P*roc. FORTE'14, LNCS 8461, pages 49-65. Springer, 2014.

[5] E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In *F*SE'13, pp. 595–598. ACM, 2013.

[6] E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. Technical report, 2015. Available at: http://costa.ls.fi.upm.es/papers/costa/AlbertGI15.pdf.

[7] B. D. Bingham, J. D. Bingham, J. Erickson, and M. R. Greenstreet. Distributed Explicit State Model Checking of Deadlock Freedom. In *P*roc. of CAV'13, volume 8044 of *L*ecture Notes in Computer Science, pages 235–241. Springer, 2013.

[8] M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2*013 IEEE Sixth International Conf. on Software Testing, Verification and Validation, pages 154–163. IEEE, 2013.

[9] F. S. de Boer, M. Bravetti, I. Grabe, M. David Lee, M. Steffen, and G. Zavattaro. A Petri Net based Analysis of Deadlocks for Active Objects and Futures. In *P*roc. of FACS 2012, 2012.

[10] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *P*roc. POPL'05, pp. 110-121. ACM, 2005.

[11] A. Flores, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. *F*ORTE'13, LNCS, pp 273–288. Springer, 2013.

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

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

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

[15] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte and Martin Steffen ABS: A Core Language for Abstract Behavioral Specification In *F*MCO'10, pages 142–164, Springer, 2012

[16] A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report http://dslab.epfl.ch/pubs/lockout.pdf, 2013.

[17] S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *P*DD'91, pages 97–107. ACM, 1991.

[18] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *P*roc. of ICSE, pages 386–396. IEEE, 2009.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM* Trans. Comput. Syst., 15(4):391–411, 1997.

[20] K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *P*roc. FASE'06, LNCS 3922, pp. 339-356. Springer, 2006.

[21] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. *F*ORTE, LNCS 7273, pages 219–234. Springer, 2012.

[22] H. Zhu, P. Hall and J. May. Software Unit Test Coverage and Adequacy. *A*CM Comput. Surv., pages 366-427, ACM, 1997.