Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Matematica

# A technique for detecting wait-notify deadlocks in Java

**Relatore:**
**Chiar.mo Prof.**
**Cosimo Laneve**

**Correlatore:**
**Chiar.ma Prof.ssa**
**Elena Loli Piccolomini**

**Presentata da:**
**Adele Veschetti**

**Sessione Unica**
**Anno Accademico 2016/2017**

# Sommario

L'analisi dei deadlock nella programmazione orientata a oggetti può risultare molto complicata, in quanto i programmi possono avere infiniti stati.
In questa tesi, presenterò una nuova tecnica per lo studio di deadlock causati dai metodi `wait` - `notify` in `Java`. A tal fine, ogni processo è stato modellizzato attraverso una Rete di Petri. Questo modello permette di determinare la presenza di deadlock analizzando il *reachability tree*.
La tecnica presentata nella mia tesi è una parte di un progetto molto più ampio e complesso, in quanto nel mio lavoro ho considerato solamente programmi con un oggetto.

# Abstract

Deadlock analysis of object-oriented programs that dynamically create threads and objects is complex, because these programs may have an infinite number of states.
In this thesis, I analyze the correctness of `wait - notify` patterns (e.g. deadlock freedom) by using a newly introduced technique that consists in an analysis model that is a basic concurrent language with a formal semantic. I detect deadlocks by associating a Petri Net graph to each process of the input program. This model allows to check if a deadlock occur by analysing the reachability tree. The technique presented is a basic step of a more complex and complete project, since in my work I only consider programs with one object.

# Contents

# Chapter 1

# Introduction

Concurrent object-oriented programming is a common model of programming born in the 80-ies [2, 18] and is now largely used by the mainstream programming languages as `Java`, `C#`, `C++`, etc. Usually concurrent languages feature parallel computing by means of threads. For example, if two codes, `code1` and `code2`, must be executed in parallel, then the languages allow programmers to specify the following execution structure:

```
1 thread1: code1
2 thread2: code2
```

This composition is safe as long as `code1` and `code2`:

- either do not access to common variables;
- they don't modify (write) common variables, they may only read variables.

For example the following code are safe:

**Example 1.**

```
1 thread1: {y = x+3}
2 thread2: {z = 3*z+4}
```

```
1 thread1: {y = x+3}
2 thread2: {z = x+1}
```

Most of programming languages guarantee exclusive access to objects by synchronization methods. Synchronization prevents threads from accessing

the shared data at the same time. The importance of synchronization is explained in the following example.

**Example 2.**

```
1 x = 0
2 thread1: sync(x){x = x+1}
3 thread2: sync(x){x = x-1}
```

At the beginning `x` is 0. If threads 1 and 2 can access to `x` in a mutually exclusive way the outcome value is always 0. In fact, if thread 1 accesses first to `x`, the variable becomes 1. Then, thread 2 subtracts 1 from `x` and the result is 0. Equivalently, if thread 2 access first to `x` and then thread 1. Without synchronization, it is impossible to predict the result. In fact, threads can access simultaneously to `x`.

A problem that can be encountered with threads is *deadlock*. When one thread depends on another for its execution, a deadlock may occur. When two threads are holding locks on two different resources, one thread would like to have other's source, a deadlock occurs. Deadlock is the result of poor programming code and is not shown by a compiler or execution environment as an exception. The *dining philosophers* problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. The problem is usually described as follows:

- a given number of philosophers are seated at a round table and they must alternately think and eat;
- between every pair of adjacent plates there is only one fork;
- each philosopher can only eat when there are both left and right forks;
- after a philosopher finishes eating, he needs to put down both forks so that the forks become available to others.

Thus, it is impossible for a philosopher to eat at the same time as one of his neighbors: the forks are a shared resource for which the philosophers are competing as shown in Figure 1.1. The problem is how to design a

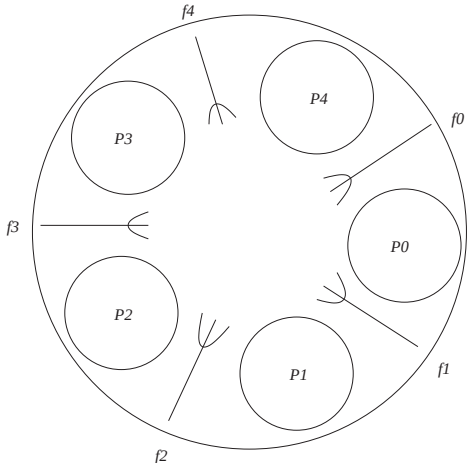



Figure 1.1: The dining philosophers problem.

discipline of behavior (a concurrent algorithm) such that every philosophers will eat. This example was designed to illustrate the challenges of avoiding deadlock and find a proper solution is not obvious. In fact, there are a lot of proposal in which his attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. Assume that each philosopher grabs the fork on his left and then the one on his right, then he releases them in reverse order. The attempted solution fails because it allows the system to reach a deadlock state, in which the philosophers will eternally wait for each other to release a fork.

A relevant feature of object-oriented calculi is thread coordination, which is usually expressed by the methods `wait` and `notify` and the relation between them can easily lead to deadlocks. In my thesis I will focus on detecting `Java` deadlocks caused by those methods.

## 1.1 Related Works

Deadlock detection in concurrent programs that create networks with arbitrary numbers of nodes is extremely complex and solutions either give imprecise answers. Deadlock-freedom of concurrent programs has been largely investigated in the literature [1, 4, 5, 10, 16, 17]. The proposed algorithms automatically detect deadlocks by building graphs of dependencies $(a, b)$ between resources, meaning that the release of a resource referenced by $a$ depends on the release of the resource referenced by $b$. The absence of cycles in the graphs entails deadlock freedom. When programs have infinite states, in order to ensure termination, current algorithms use finite approximate models that are excerpted from the dependency graphs.
In [8, 11] the problem has been solved for value-passing CSS [13] and pi calculus in [14]. In that case there are two formal models: Petri Nets and deadlock Analysis models - lams [9]. Lams are basic recursive models that collect dependencies and dynamic name creation. In [12] it is demonstrated that is possible to define a deadlock analyzer for object-oriented programs by only using an extension of lams. The algorithm developed in [12] has been prototyped in `JaDA` [7], which is a tool that detects deadlocks of `Java` programs at static time. While the type system in [6] simply checks static information, `JaDA` infers the behavioural types from the bytecode.

## 1.2 Thesis Structure

The aim of my thesis is to design and implement a technique capable of detecting `Java` deadlocks caused by `wait - notify` methods with one object. The analysis model is a basic concurrent language with a formal semantic where each process $P$ is generated by the syntax described in Chapter 2. In this chapter, I also show some examples of programs with two objects, analysing their behaviours and if a deadlock may occur or not.
I modeled very process of the language has been modeled into a Petri Net graph. Models are presented in Chapter 3. I designed a compiler that takes

in input a program and returns the corresponding Petri Net graph for each process. By means of an off-the-shelf solver for Petri Net (PIPE [15]), I have analyzed the reachability tree in order to check if a deadlock occurs or not. In Chapter 4, I describe the implementation choices of my work and I report some examples and results in Chapter 5.

# Chapter 2

# Concurrency in Java

Java is an object-oriented programming language that is concurrent. Concurrency refers to the ability of different parts of a program to be executed out-of-order without affecting the final outcome. In other words, concurrency is the ability to run several programs in parallel and parallel execution is when two tasks start at the same time. Thus, threads are generated in parallel in order to get access to common resources locks are used. Locks grant access to objects in a mutually exclusive way. Moreover, Java provides three methods *wait*, *notify* and *notify all* to improve the efficiency communication between threads.
In this thesis, I will focus on the concurrency model of Java and the correctness of concurrent programming patterns.

## 2.1 The Java Thread Model

Java provides built-in support for *multithreaded programming*. The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses and all other threads continue to run.

There exist several threads states: a thread can be running or it can be ready to run. A running thread can also be suspended and it can then be resumed. Moreover, a thread can be blocked when waiting for a resource. At any time, a thread can be terminated and its execution is stopped immediately. Once terminated, a thread cannot be resumed.

### 2.1.1 Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization and Java provides a language-level support for it. A monitor is an object that is used as a mutually exclusive lock and the rule is that there should be only a thread which can own a monitor at a given period of time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it wants to. Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

### 2.1.2 Wait, Notify and NotifyAll

Basically, in `Java`:

- every object has a lock associated with it;
- a thread must acquire the lock, before it can enter a synchronized block or a method;

- the lock is automatically released when the thread exits the synchronized block;

- a thread that cannot acquire the lock is suspended until the lock is available.

If a thread tries to enter a synchronized block that is locked by another thread, it waits until the lock is released. In this case, the thread is in the *entry-set* and is called runnable. The Object class in `Java` contains three final methods that allows threads to communicate about the lock status of a resource, these methods are `wait()`, `notify()`, and `notifyAll()`. Being final means that every subclass of Object (i.e. every `Java` class) inherits them and they cannot be modified. Those three methods can be called only in a synchronized context. The rules for using these methods are actually quite simple:

- `wait()` tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`. In other word, the `wait()` method releases the lock and suspends the thread;

- `notify()` wakes up only one thread between the waiting threads;

- `notifyAll()` wakes up all the threads that called `wait()` on the same object.

Every object has a *wait-set*, that is a thread set which contains all the threads that have called the `wait()` and have not been notifies by the `notify()` or `notifyAll()` methods. The mechanism of inter-thread communication is illustrated in figure 2.1.

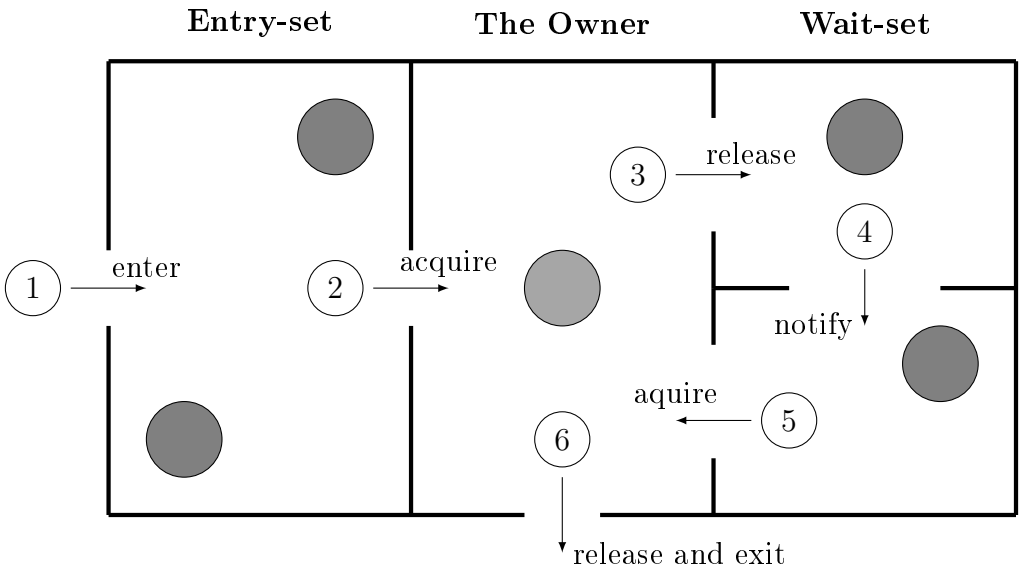



Figure 2.1: Diagram of inter-thread communication.

### 2.1.3 Deadlocks

A special type of error that needs to be avoided that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlocks are difficult to avoid or anticipate since they may not happen during every execution and may involve more than two threads. They may have catastrophic effects for the overall functionality of the software system. The following example shows how the relation between `wait()` and `notify()` can easily lead to deadlock.

**Example 3.** The code below will terminate successfully because the `notify()` operation on $x$ by $t$ will be always performed after the `wait()` operation on

$x$ by the current thread because it owns the lock on $x$ when `t.start()` is executed.

```
1   Thread t = new Thread(){
2      public void run(){
3         synchronized(x) {
4            x.notify();
5         } ;
6      }
7   };
8   synchronized(x){
9   t.start();
10     try {
11        x.wait();
12     } catch (InterruptedException e) {
13     };
14  }
```

If the scope of synchronize is a `x.wait()` invocation, a deadlock may occur because the thread `t` may be performed before the x.wait() instruction.

Thus, the relation between `wait()` and `notify()` can easily lead to deadlocks. In fact, it may happen when:

- the `x.wait()` operation in `t` does not happen before a matching `x.notify()` in `t'`;
- a lock on a object held by `t` is blocking the execution of `t'`.

The following example is the dining philosophers problem discussed in Chapter 1. It was designed to illustrate the challenges of avoiding deadlock.

**Example 4.**

```
1   public void buildTable(int n, Object x, Object y){
2      if(n==0){
3         synchronized(y){synchronized(x){}};
4      }
5      else{
6         final Object z = new Object();
7         Thread t = new Thread(){
8            public void run(){
9               synchronized(x){synchronized(z){}};
10           }
11        };
12        t.start();
13        this.buildNetwork(n-1, z, y);
14     }
15  }
```

`new Object z` creates a new Object, `(new Thread P)Q` creates a new thread with body P that runs in parallel with thread Q and `synchronized(x){P}` is the method that locks x and runs P. This method creates $n+1$ threads ($n+1$ philosophers) and each one shares an object (a fork) with the closest one. Every philosopher, except one, grabs the fork on his left and on his right then releases them, in this order. The exceptional case is the branch `n == 0` because the strategy is opposite. When the method is called by `buildTable(n,x,x)` a deadlock never occurs. If I change the branch `n == 0` with
`synchronized(y){synchronized(x){}}` a deadlock may occur because every philosopher has the same strategy.

# Chapter 3

# The Analysis Model

My analysis model is a basic concurrent language with a formal semantic. I use two countable sets of names: $x, y, \ldots$ are the object names and $A, B, C, \ldots$ the method names. A program is a pair $(\mathscr{D}, P)$ where $\mathscr{D}$ is a finite set of method definitions and $P$ is the main process.
The processes $P$ are the terms generated by the following syntax:

$$P ::= \quad \mathsf{0} \mid (\nu x)P \mid (\nu P)P \mid f(\bar{x}) \mid \mathit{notifyAll}(x).P \mid \mathit{sync}(x)\{P\}.P$$

$$\mid \mathit{wait}(x).P \mid \mathit{notify}(x).P$$

A process can be the inert process 0 or a restriction $(\nu x)P$ that behaves like $P$, but the external environment cannot access to $x$ (the object), in this thesis only one object processes are analyzed. $P$ can also be $(\nu P)P$, i.e. the creation of a new thread, or $f(\bar{x})$ an invocation to a method. Moreover the process can be $\mathit{wait}(x).P$, $\mathit{notify}(x).P$, $\mathit{notifyAll}(x).P$, they modify the states of threads. The thread that executes `wait()` is suspended and the lock $x$ is released; `notify()` wakes up one thread and the lock $x$ is acquired by the thread; `notifyAll()` wakes up all the threads in the wait-set. In the end the process $P$ can be $\mathit{sync}(x)\{P\}.P$, that executes the first $P$ with the exclusive access to $x$ and then executes the second process.

## 3.1 Semantics

The semantics of my programming language are defined operationally by means of a transition system. The formal definition is below.

1. There are terms $\mathbb{P} ::= P \mid P \overset{x}{\bullet} \mathbb{P}$ that are called *threads*. The term $P \overset{x}{\bullet} \mathbb{P}$ corresponds to a thread that is performing $P$ in a critical section for $x$; when $P$ terminates the lock of $x$ must be released (if $\overset{x}{\bullet} \notin \mathbb{P}$) and $\mathbb{P}$ may start.

2. Configurations are multisets of threads, written $\mathbb{P}_1 \mid \cdots \mid \mathbb{P}_n$, or sometimes with the shorter form $\prod_{i \in 1,\ldots,n} \mathbb{P}_i$; configurations are ranged over by $\mathscr{T}$;

3. Let $\equiv$ be the congruence relation that includes commutativity and associativity of $\mid$, with $\mathsf{0}$ being the identity, and $f(\overline{z}) \equiv P_f\{{}^{\overline{z}}/_{\overline{x}}\}$, assuming that $f(\overline{x}) = P_f$;

4. We write $x \in \mathscr{T}$ if there is $\mathbb{P} \in \mathscr{T}$ such that $\mathbb{P} = \mathbb{P}' \overset{x}{\bullet} \mathbb{P}''$ and $\mathbb{P}'$ is not prefixed by $wait(x)$.;

5. Let $\mathbb{P}_1, \cdots, \mathbb{P}_n, \mathbb{P}_{n+1}$ be terms that do not contain $\overset{x}{\bullet}$ (they may contain $\overset{y}{\bullet}$, $\overset{z}{\bullet}$, $\cdots$, with $y, z, \cdots \neq x$). We define

   $$sync(x)^n\{\, \mathbb{P}_1 \overset{x}{\bullet} \cdots \overset{x}{\bullet} \mathbb{P}_n \overset{x}{\bullet} \mathbb{P}_{n+1} \,\} \overset{def}{=} \underbrace{sync(x)\{\cdots sync(x)\{}_{n\,\text{times}} \mathbb{P}_1\} \cdots .\mathbb{P}_n\}.\, \mathbb{P}_{n+1}$$

   With the foregoing assumptions, we define $\#_x(\mathbb{P}_1 \overset{x}{\bullet} \cdots \overset{x}{\bullet} \mathbb{P}_n \overset{x}{\bullet} \mathbb{P}_{n+1}) = n$

6. The transition relation is $\mathscr{T} \longrightarrow \mathscr{T}'$ defined as follows

$$\text{(Zero)}\quad 0 \overset{x}{\bullet} \mathbb{P} \mid \mathscr{T} \longrightarrow \mathbb{P} \mid \mathscr{T}$$

$$\text{(NewO)}\quad \frac{z \text{ fresh}}{(\nu\, x)\mathbb{P} \mid \mathscr{T} \longrightarrow \mathbb{P}\{{}^{z}/_{x}\} \mid \mathscr{T}} \qquad \text{(NewT)}\quad (\nu\, P)\mathbb{P} \mid \mathscr{T} \longrightarrow P \mid \mathbb{P} \mid \mathscr{T}$$

$$\text{(Sync)}\quad \frac{x \notin \mathscr{T}}{sync(x)\{\ P\ \}.\ \mathbb{P} \mid \mathscr{T} \longrightarrow P \overset{x}{\bullet} \mathbb{P} \mid \mathscr{T}}$$

$$\text{(NtfT)}\quad \frac{\mathbb{P}' = \mathbb{P}'_1 \overbrace{\overset{x}{\bullet} \cdots \overset{x}{\bullet}}^{n \text{ times}} \mathbb{P}'_n \overset{x}{\bullet} \mathbb{P}'_{n+1}}{\texttt{notify}()x.\mathbb{P} \mid \texttt{wait}()x.\mathbb{P}' \mid \mathscr{T} \longrightarrow \mathbb{P} \mid sync(x)^n\{\ \mathbb{P}'\ \} \mid \mathscr{T}} \qquad \text{(NtfF)}\quad \frac{no\ wait(x).\mathbb{P}'\ in\ \mathscr{T}}{\texttt{notify}()x.\mathbb{P} \mid \mathscr{T} \longrightarrow \mathbb{P} \mid \mathscr{T}}$$

$$\text{(Cong)}\quad \frac{\mathscr{T}_1 \equiv \mathscr{T}'_1 \quad \mathscr{T}'_1 \longrightarrow \mathscr{T}'_2 \quad \mathscr{T}'_2 \equiv \mathscr{T}_2}{\mathscr{T}_1 \longrightarrow \mathscr{T}_2}$$

**Definition 1.** ***Deadlock-freedom*** *A program $(\mathscr{D}, P)$ is deadlock-free if the following conditions hold:*

$$\begin{array}{ll} \textit{whenever } P \longrightarrow^* \mathscr{T} & \textit{and } \mathscr{T} = (\nu x_1) \ldots (\nu x_n)(sync(x)\{\ P\ \}.\ \mathbb{P} \mid \mathscr{T}') \\ \textit{then there exists } \mathscr{T}'' & \textit{such that } \mathscr{T} \longrightarrow \mathscr{T}''. \end{array}$$

In other words, a program is deadlock-free when there exists a thread such that it can be reduced to the identity 0.

## 3.2 Examples

Some examples are discussed below, highlighting their behaviours and if deadlock occur or not.

1.

$$
\begin{aligned}
sync(x)\{\ (\nu\, sync(x)\{\ notify(x).\ \}.\ )wait(x).\ \}. \\
&\rightarrow (\nu\, sync(x)\{\ notify(x).\ \}.\ )wait(x).\overset{x}{\bullet}\ 0 \\
&\rightarrow sync(x)\{\ notify(x).\ \}.\ \mid wait(x).\overset{x}{\bullet}\ 0 \\
&\rightarrow notify(x).\overset{x}{\bullet}\ 0 \mid wait(x).\overset{x}{\bullet}\ 0 \\
&\rightarrow 0\overset{x}{\bullet}\ 0 \mid 0\overset{x}{\bullet}\ 0
\end{aligned}
$$

Lock is acquired by the main thread; with the `wait()` method the thread releases the lock that is acquired by the second thread. When `notify()` is called, it wakes up the thread and moves it from the sleeping-queue to the ready-queue for it to be executed.

2.

$$
\begin{aligned}
sync(x)\{\ (\nu\, sync(x)\{\ notify(x).wait(x).\ \}.\ )\ wait(x).notify(x).\ \}. \\
&\rightarrow (\nu\, sync(x)\{\ notify(x).wait(x).\ \}.\ )\ wait(x).notify(x).\overset{x}{\bullet}\ 0 \\
&\rightarrow sync(x)\{\ notify(x).wait(x).\ \}.\ \mid wait(x).notify(x).\overset{x}{\bullet}\ 0 \\
&\rightarrow notify(x).wait(x).\overset{x}{\bullet}\ 0 \mid wait(x).notify(x).\overset{x}{\bullet}\ 0 \\
&\rightarrow wait(x).\overset{x}{\bullet}\ 0 \mid notify(x).\overset{x}{\bullet}\ 0 \\
&\overset{(1)}{\longrightarrow} 0\overset{x}{\bullet}\ 0 \mid 0\overset{x}{\bullet}\ 0
\end{aligned}
$$

Where in (1) we used the commutativity of $\mid$. Main thread has the lock that is released by the `wait()` method: the thread is in the wait-set. Now the inner thread can acquire the lock: the `notify()` method wakes up the thread that is waiting on this object's monitor. Right after the lock is released by calling the `wait()`, the lock is acquired by the main thread and woken up by the `notify()`.

3. We report an example where deadlock may occur depending on which

process we choose to wake up.

$(\nu\, sync(x)\{\ notify(x).wait(x).\ \}.\ )\ sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow (\nu\, sync(x)\{\ notify(x).wait(x).\ \}.\ ) \mid sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow sync(x)\{\ notify(x).wait(x).\ \}.\ \mid sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow notify(x).wait(x).\overset{x}{\bullet}\ 0 \mid sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow wait(x).\overset{x}{\bullet}\ 0 \mid sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow wait(x).\overset{x}{\bullet}\ 0 \mid wait(x).notify(x).\overset{x}{\bullet}\ 0$ ***Deadlock***

According to the previous rules, there is not a transition for the `wait()`method. Thus, in this case a deadlock occurs.

$(\nu\, sync(x)\{\ notify(x).wait(x).\ \}.\ )\ sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow (\nu\, sync(x)\{\ notify(x).wait(x).\ \}.\ ) \mid sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow sync(x)\{\ notify(x).wait(x).\ \}.\ \mid sync(x)\{\ wait(x).notify(x).\ \}.$

$\rightarrow sync(x)\{\ notify(x).wait(x).\ \}.\ \mid wait(x).notify(x).\overset{x}{\bullet}\ 0$

$\rightarrow notify(x).wait(x).\overset{x}{\bullet}\ 0 \mid wait(x).notify(x).\overset{x}{\bullet}\ 0$

$\rightarrow wait(x).\overset{x}{\bullet}\ 0 \mid notify(x).\overset{x}{\bullet}\ 0$

$\rightarrow 0\overset{x}{\bullet}0 \mid 0\overset{x}{\bullet}0$

In this case, the process is deadlock-free. The following transition system underlines the fact that deadlock may occur or not. In fact, in this transition system it is clear that the process may finish or not depending on which transition we decide to make: two times on three a deadlock does not occur, but not every choice leads to the same final state.

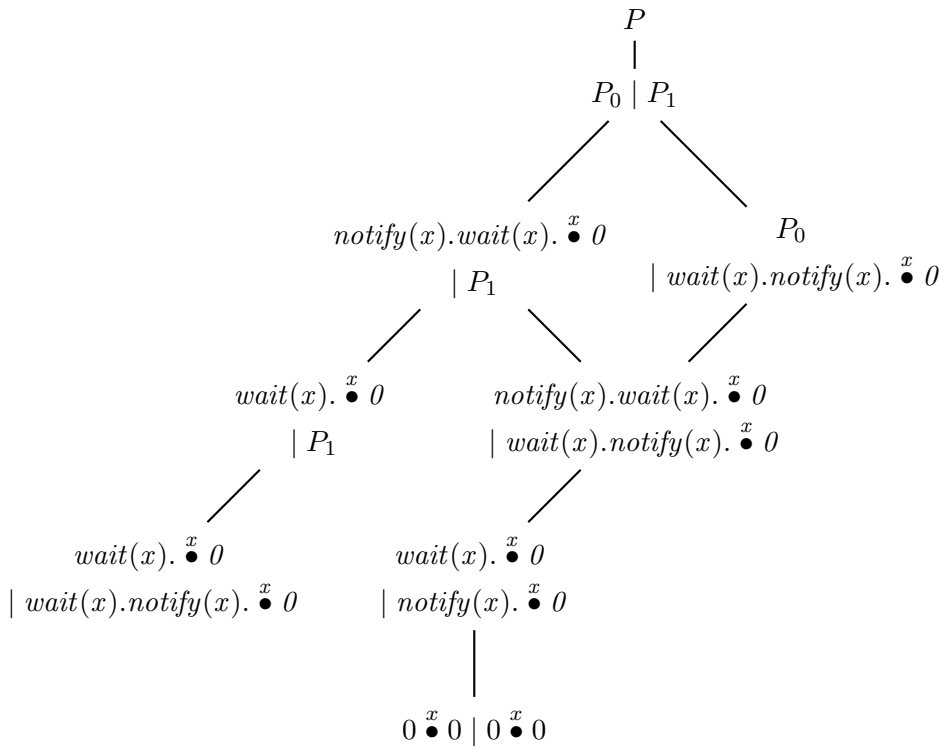



Figure 3.1: Transition system of the deadlock-free program of Example 3.

4. This code will terminate successfully because the `notify()` operation will be always performed after the `wait()`. In this case for any reduction a deadlock never occur. In fact, the transition system ends always with the node $0 \overset{x}{\bullet} 0$.

$(\nu\, sync(x)\{\ notify(x).wait(x).notify(x).\ \}.\ )\ sync(x)\{\ notify(x).wait(x).notify(x).\ \}.$

$\rightarrow sync(x)\{\ notify(x).wait(x).notify(x).\ \}.\ \mid sync(x)\{\ notify(x).wait(x).notify(x).\ \}.$

$\rightarrow notify(x).wait(x).notify(x).\ \overset{x}{\bullet}\ 0 \mid sync(x)\{\ notify(x).wait(x).notify(x).\ \}.$

$\rightarrow wait(x).notify(x).\ \overset{x}{\bullet}\ 0 \mid sync(x)\{\ notify(x).wait(x).notify(x).\ \}.$

$\rightarrow wait(x).notify(x).\ \overset{x}{\bullet}\ 0 \mid notify(x).wait(x).notify(x).\ \overset{x}{\bullet}\ 0$

$\rightarrow notify(x).\ \overset{x}{\bullet}\ 0 \mid wait(x).notify(x).\ \overset{x}{\bullet}\ 0$

$\rightarrow 0 \overset{x}{\bullet} 0 \mid notify(x).\ \overset{x}{\bullet}\ 0$

$\rightarrow 0 \overset{x}{\bullet} 0 \mid 0 \overset{x}{\bullet} 0$

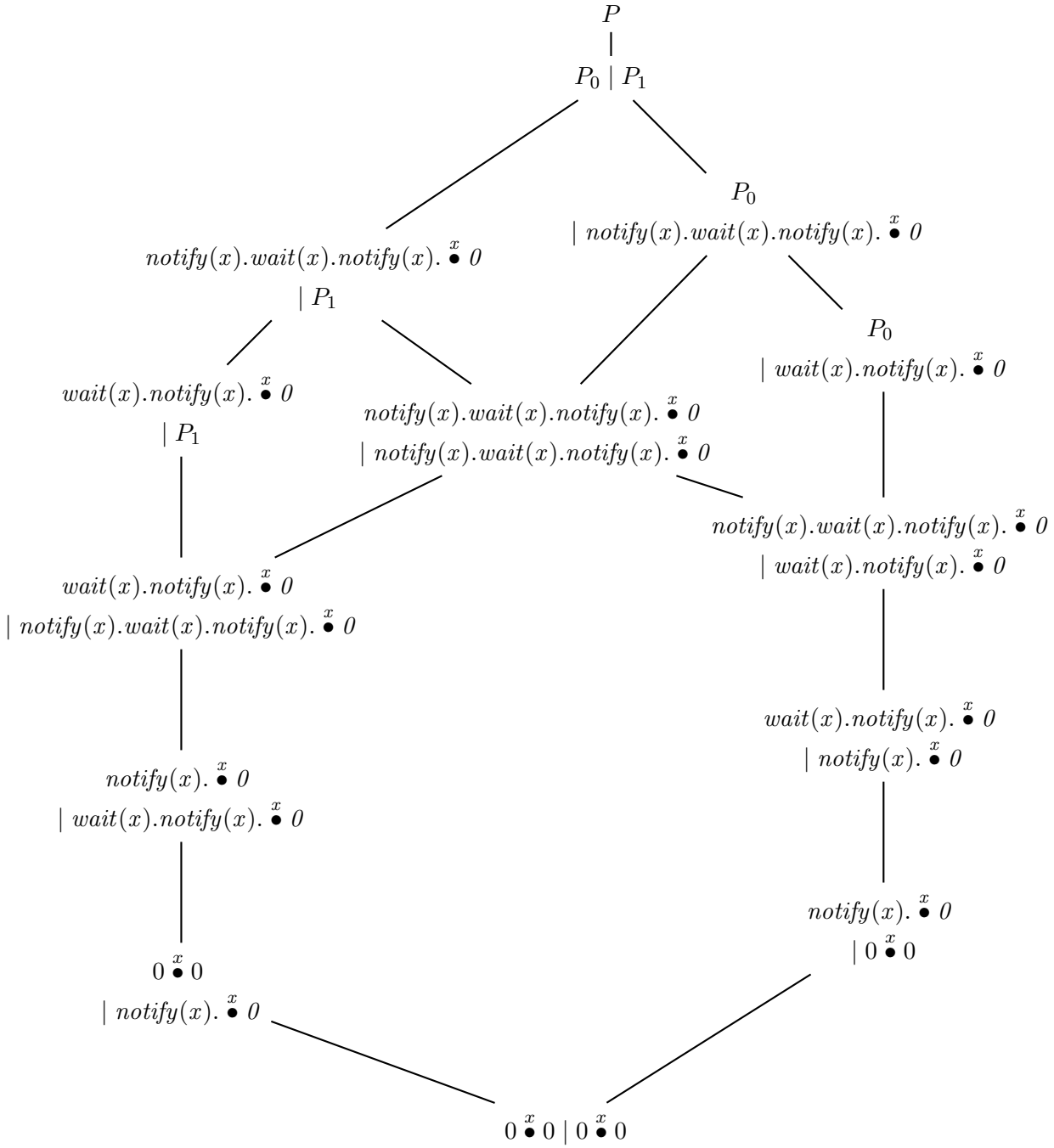



Figure 3.2: The transition system of the program described in Example 4.

Now I will show some examples of code with two objects.

5. The following code is an example of a deadlock-free program.

$$sync(x)\{\ (\nu\, sync(y)\{\ sync(x)\{\ notify(x).\ \}.\ wait(y).\ \}.\ )wait(x).sync(y)\{\ notify(y).\ \}.\ \}.$$
$$\rightarrow (\nu\, sync(y)\{\ sync(x)\{\ notify(x).\ \}.\ wait(y).\ \}.\ )wait(x).sync(y)\{\ notify(y).\ \}.\ \overset{x}{\bullet}\ 0$$
$$\rightarrow sync(y)\{\ sync(x)\{\ notify(x).\ \}.\ wait(y).\ \}.\ \mid wait(x).sync(y)\{\ notify(y).\ \}.\ \overset{x}{\bullet}\ 0$$
$$\rightarrow sync(x)\{\ notify(x).\ \}.\ wait(y).\ \overset{y}{\bullet}\ 0 \mid wait(x).sync(y)\{\ notify(y).\ \}.\ \overset{x}{\bullet}\ 0$$
$$\rightarrow notify(x).wait(y).\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0 \mid wait(x).sync(y)\{\ notify(y).\ \}.\ \overset{x}{\bullet}\ 0$$
$$\rightarrow wait(y).\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0 \mid sync(y)\{\ notify(y).\ \}.\ \overset{x}{\bullet}\ 0$$
$$\rightarrow wait(y).\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0 \mid notify(y).\ \overset{x}{\bullet}\ 0\ \overset{y}{\bullet}\ 0$$
$$\rightarrow 0\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0 \mid 0\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0$$

6. This last example gives a deadlock. In fact, during the analysis we can choose between two or more reductions and, as shown in the following transition system, sometimes a deadlock can occur.

$(\nu\, sync(x)\{\ sync(y)\{\ wait(y).\ \}.\ \}.)sync(x)\{\ sync(y)\{\ notify(y).\ \}.\ \}.$

$\rightarrow sync(x)\{\ sync(y)\{\ wait(y).\ \}.\ \}.\ |\ sync(x)\{\ sync(y)\{\ notify(y).\ \}.\ \}.$

$\rightarrow^{*} sync(y)\{\ wait(y).\ \}.\ \overset{x}{\bullet}\ 0\ |\ sync(x)\{\ sync(y)\{\ notify(y).\ \}.\ \}.$

$\rightarrow wait(y).\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0\ |\ sync(x)\{\ sync(y)\{\ notify(y).\ \}.\ \}.$

$\rightarrow wait(y).\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0\ |\ sync(y)\{\ notify(y).\ \}.\ \overset{x}{\bullet}\ 0$

$\rightarrow wait(y).\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0\ |\ notify(y).\ \overset{x}{\bullet}\ 0\ \overset{y}{\bullet}\ 0$

$\rightarrow 0\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0\ |\ 0\ \overset{y}{\bullet}\ 0\ \overset{x}{\bullet}\ 0$

$(\nu\, sync(x)\{\ sync(y)\{\ wait(y).\ \}.\ \}.)sync(x)\{\ sync(y)\{\ notify(y).\ \}.\ \}.$

$\rightarrow sync(x)\{\ sync(y)\{\ wait(y).\ \}.\ \}.\ |\ sync(x)\{\ sync(y)\{\ notify(y).\ \}.\ \}.$

$\rightarrow sync(x)\{\ sync(y)\{\ wait(y).\ \}.\ \}.\ |\ sync(y)\{\ notify(y).\ \}.\ \overset{x}{\bullet}\ 0$

$\rightarrow sync(x)\{\ sync(y)\{\ wait(y).\ \}.\ \}.\ |\ notify(y).\ \overset{x}{\bullet}\ 0\ \overset{y}{\bullet}\ 0$

$\rightarrow sync(y)\{\ wait(y).\ \}.\ \overset{x}{\bullet}\ 0\ |\ 0\ \overset{x}{\bullet}\ 0\ \overset{y}{\bullet}\ 0$

$\rightarrow wait(y).\ \overset{x}{\bullet}\ 0\ \overset{y}{\bullet}\ 0\ |\ 0\ \overset{x}{\bullet}\ 0\ \overset{y}{\bullet}\ 0$ ***Deadlock***

We can compare the two reductions. In the first one, with * transition I analyze before the $P_0$ thread that ends with a `wait()`. Then, I make the $P_1$ reductions and get a `notify()`. So we can reduce both with the rule of `notify()`- `wait()`. On the other hand, in the second case I decided to reduce before the main thread, obtaining a `notify()`and so an identity 0. But the other thread ends with a `wait()`and it cannot be reduced without a parallel `notify()`, so it gives a deadlock.

The transition system in Figure 3.3 shows the same result: some reductions lead to the final state 0, but there are some other that produce deadlock.

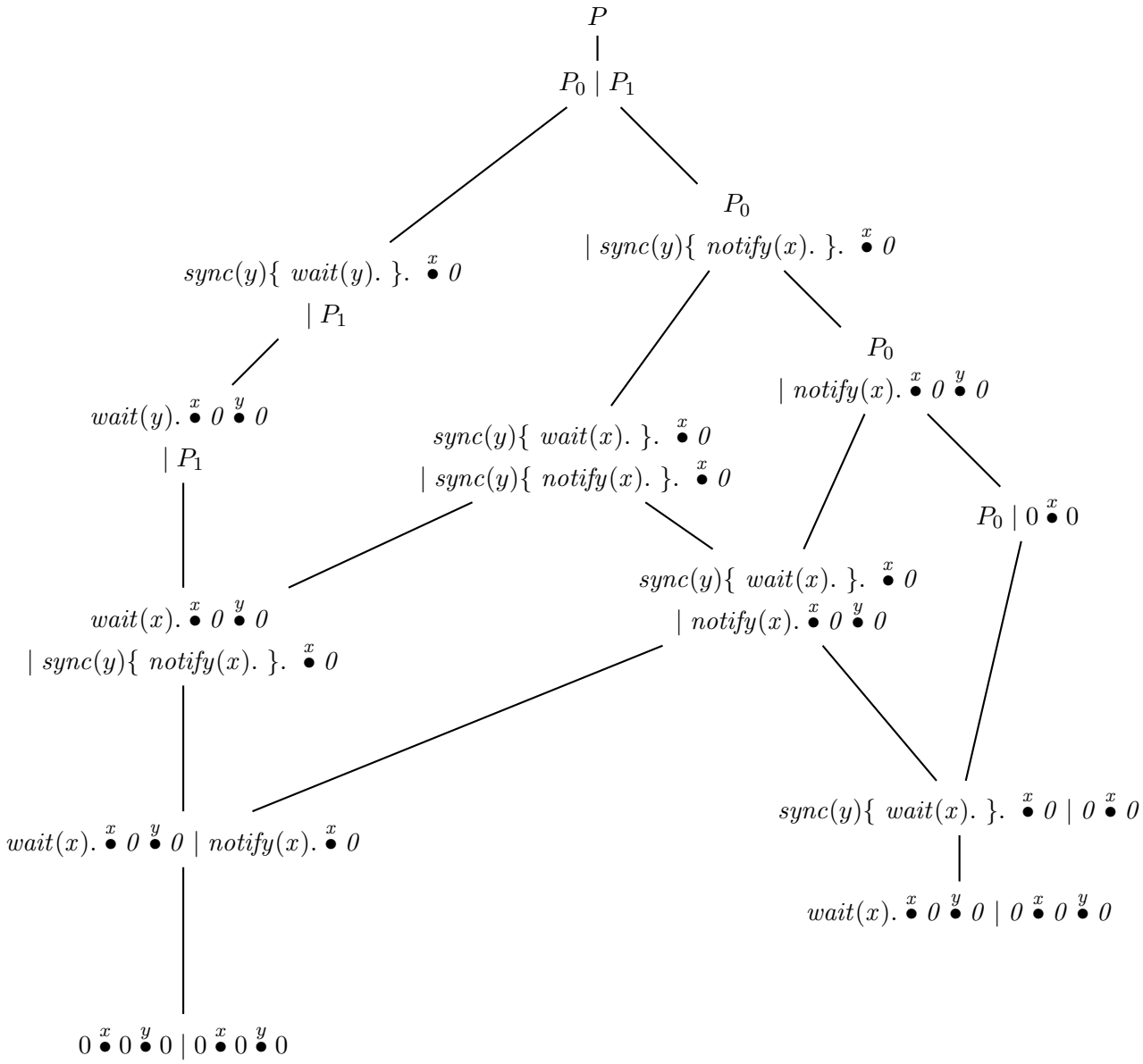



Figure 3.3: The transition system of the program described in Example 6.

# Chapter 4

# Petri Nets

Petri Nets are a mathematical tool for modeling systems. Analysis of Petri Net can reveal important information about the structure and then this information can be used to evaluate the modeled system and suggest changes. The practical application of Petri Nets considers them as an auxiliary analysis tool. For this approach, the system is modeled as a Petri Net and then it is analyzed. Some problems can occur during the analysis, then it is necessary to modify the design of the model until the system is error-free.

## 4.1 Petri Net Structure

A Petri Net is a tuple $(N, T, I, O)$: a finite set of *places* $N = \{n_1, \ldots, n_k\}$, a finite set of *transitions* $T = \{t_1, \ldots, t_m\}$, an *input function* $I$ and an *output function* $O$. Where $N \cap T = \emptyset$ and:

$$I : T \to N^{\infty}$$

that identifies the places leading *into* a transition and

$$O : T \to N^{\infty}$$

that identifies the places leading *out* a transition. The cardinality of $N$ is $k \geq 0$, the cardinality of set $T$ is $m \geq 0$. An arbitrary element of the first

set is denoted by $n_i$, $i = 1, \ldots, k$, and one element of the set of transitions by $t_j$, $j = 1, \ldots, m$.

Petri Net can be graphically represented by a bipartite directed multigraph. Thus, a Petri Net is defined by places and transitions and a Petri Net graph has two types of nodes: a circle  that corresponds to a place, and a bar  that is the graphical representation of a transition. Directed arcs connect places and transitions, some arcs are directed from places to transitions and other arcs from transitions to places. A Petri Net graph is a multigraph because it allows multiple arcs from one node to another. Petri Net graphs are usually associated with an initial marking. A marking is an assignment of tokens to the places of a Petri Net and the number of tokens can change during the execution. Formally, a marking $\mu$ can be defined as a function from the set of places to a set of nonnegative integers: $\mu : N \to \mathbb{N}$. In Petri Net graph a marking is represented by a dot $\bullet$.

### 4.1.1 Petri Net Execution

The execution of a Petri Net depends by the number and the distribution of tokens. A Petri Net executes by *firing* transitions: a transition fires by removing tokens from its input places and creating new tokens to its output ones. A transition may fire if it is enabled, it means that each of its input places has at least as many tokens in it as arcs from the place to the transition. So multiple tokens are needed for multiple input arcs. A transition fires by removing all its enabling tokens from its input places and then locating into each of its output places one token for each arc from the transition to the place. An example of Petri Net graph and its execution is shown below.

**Example 5.** An example of Petri Net graph and its execution. At the beginning, the place $N_0$ has the initial token, red transitions are the enabled ones.

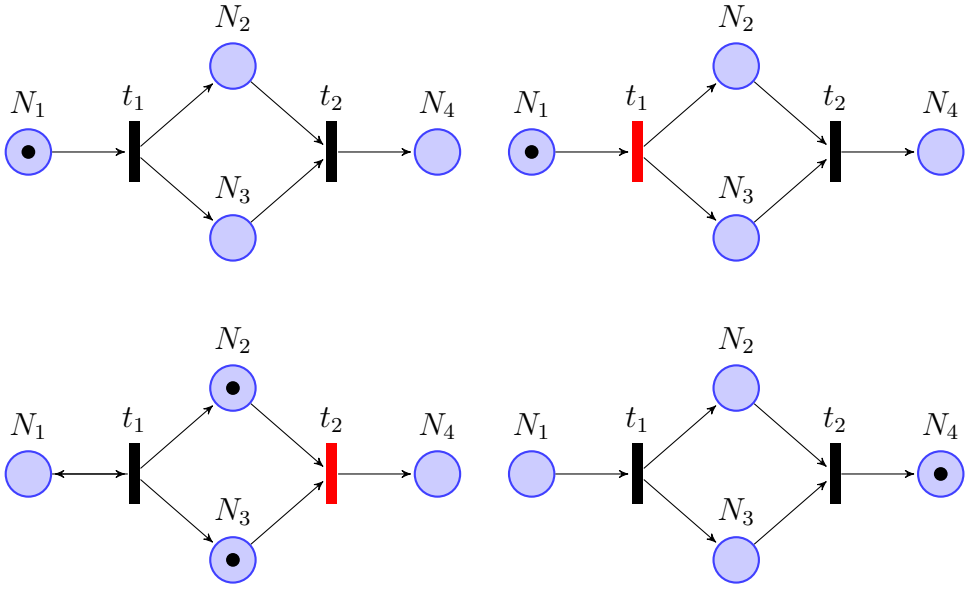
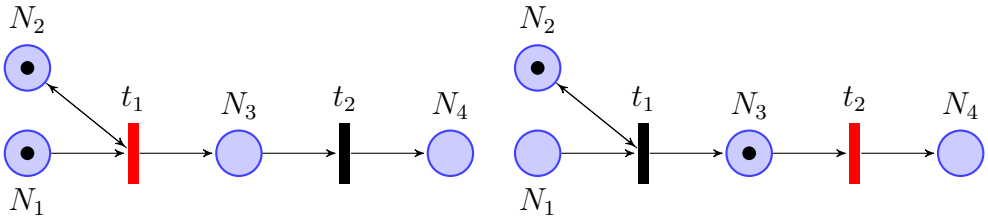



Figure 4.1: A Petri Net graph and its execution.

**Example 6.** I report another example a Petri Net graph that explicates when a transition is not enabled. In the second step of the execution the




transition $t_1$ is not enabled because in place $N_1$ there is not a token, that has been deleted and put in $N_2$ with the first transition.

### 4.1.2 Reachability

Determining the reachability of a marking is one of the most interesting problems of Petri Net graph. When a marking is reachable from the initial

one? Formally, I can describe this problem as follows.

**Definition 2** (Reachability problem)**.** *Giving a Petri Net $C$ with marking $\mu$ and a marking $\mu'$, is $\mu' \in R(C, \mu)$?*

An important tool to determine the reachability set of a Petri Net is the *reachability tree*. I will describe it with an example.
As shown in Figure 4.2, the initial marking has two possible transitions: $t_1$ and $t_2$. Thus, in the reachability tree I add two nodes which result from both transitions. Now, I consider all markings reachable from this configuration: from $(1, 1, 0)$ I can fire $t_1$, giving $(1, 2, 0)$ and $t_2$, giving $(0, 2, 1)$; from $(0, 1, 1)$ it is possible to reach the $(0, 0, 1)$ marking.

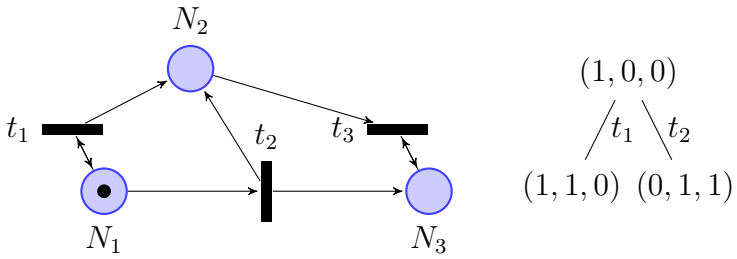



Figure 4.2: A Petri Net graph with the first step of its reachability tree.

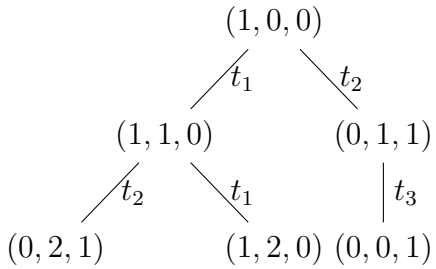



Figure 4.3: Second step of reachability tree.

With the new three markings I can repeat the process, noticing that the marking $(1, 0, 0)$ is dead. So I obtain the third step of the reachability tree, shown in Figure 4.4. If this procedure is repeated over and over, every reachable marking will be produced. Sometimes, the reachability set might

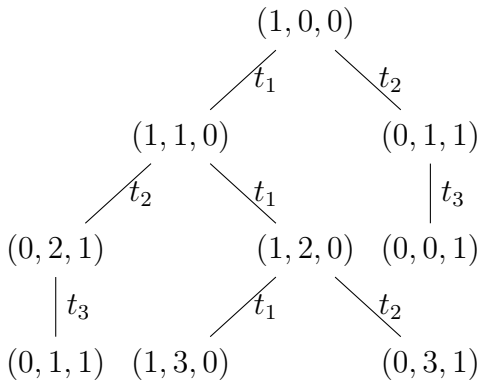



Figure 4.4: Third step of reachability tree.

be infinite and also the corresponding reachability tree. However, even a Petri Net with a finite reachability set, can have an infinite tree: it represents all possibile sequences of transitions and if they are always enabled, the tree would be infinite. I show a sample graph with a infinite reachability tree in Figure 4.5. Since the reachability tree is an important tool for Petri Net analysis, there are some techniques that allow us to limit the tree to a finite size. It is necessary to classify the markings: there are the dead markings, the one in which no transition is enabled, known as *terminal nodes*; the *duplicate nodes*, those markings which have previously appeared in the tree and no successors of a duplicate node need to be considered.

## 4.2 Modeling with Petri Nets

A performance model of Java execution has been developed by using Petri Net graph. Each process has been modeled by a Petri Net graph and saved as an .xml file. It allows us to analyze multi-threaded Java applications. During the execution of a program, we combine the corresponding Petri Net models and, using the Petri Net tool PIPE [15], we can analyze the process. The reachability tree analysis allows us to check if a deadlock occur. This analysis will be examine in depth in the following chapters.

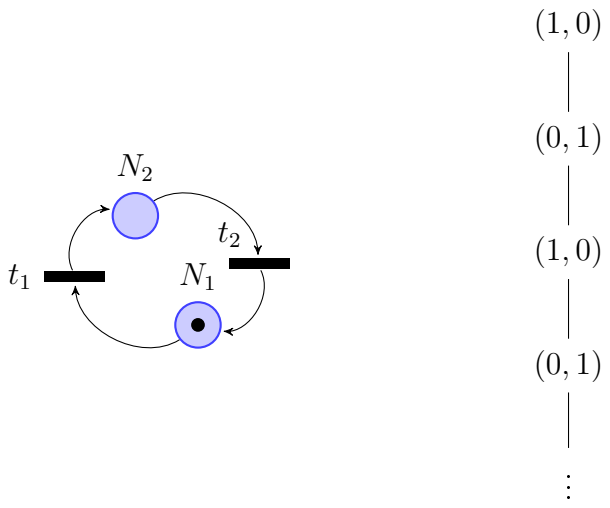



Figure 4.5: A finite Petri Net graph with an infinite reachability tree.

### 4.2.1 The Model

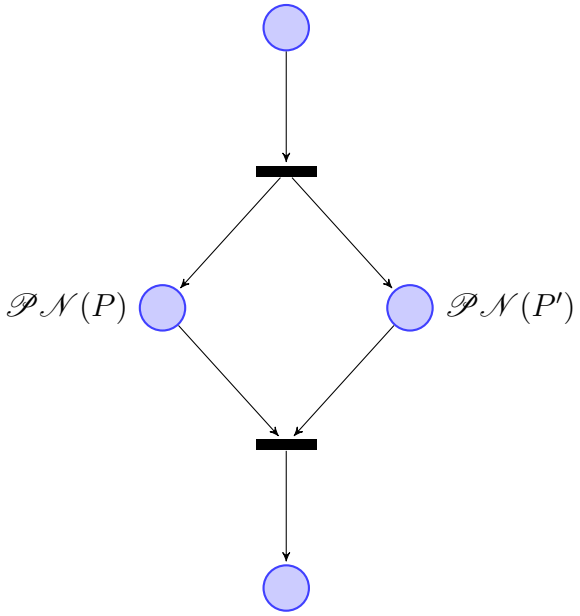
Now, I will describe the model for each instruction. The syntax used is described in Chapter 2. We indicate with the notation $\mathscr{PN}(P)$ the corresponding model of $P$.

- The inert process 0 is represented by the empty place:

  $\mathscr{PN}(0)=$ 

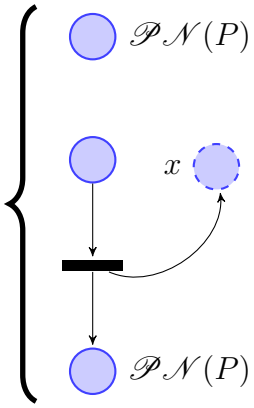
- The creation of a new thread $(\nu P)P'$ has been modeled by the Petri Net graph is shown below:

$\mathscr{PN}((\nu P)P')=$

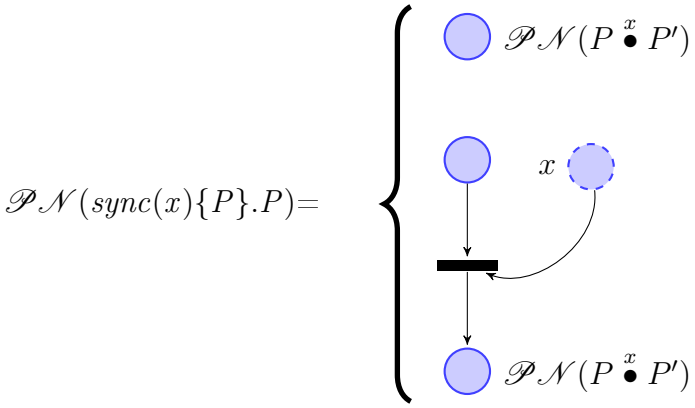



- We also use $P \overset{x}{\bullet} P'$ that corresponds to a thread that is performing $P$ in a critical section for $x$. The first model is if $\overset{x}{\bullet}\notin P$, the second otherwise.

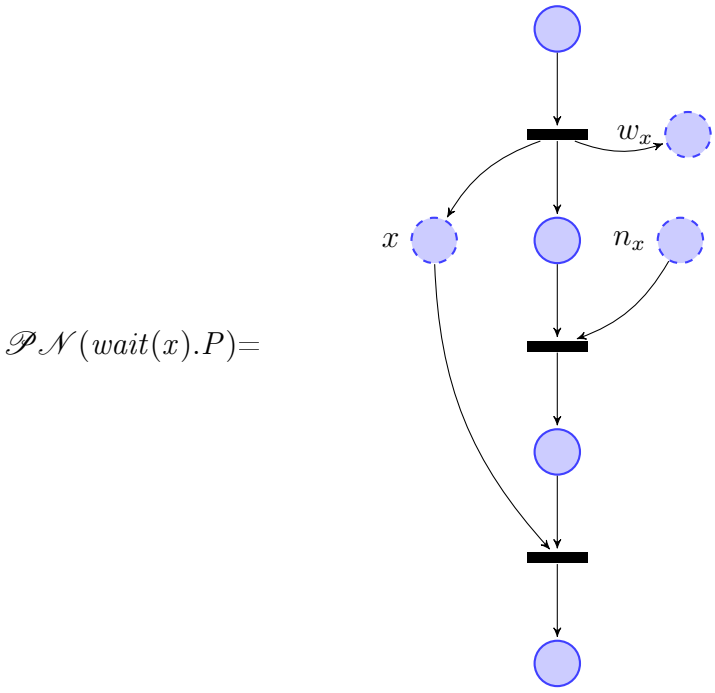
$\mathscr{PN}(0 \overset{x}{\bullet} P')=$




- We modeled the synchronization method $sync(x)\{P\}.P$ as shown below, in order of appearance for $\overset{x}{\bullet}\in P'$ and for $\overset{x}{\bullet}\notin P'$.

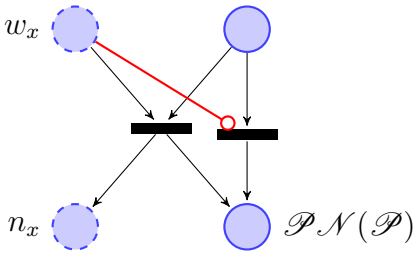



- For the `wait()` method we chose the following model:

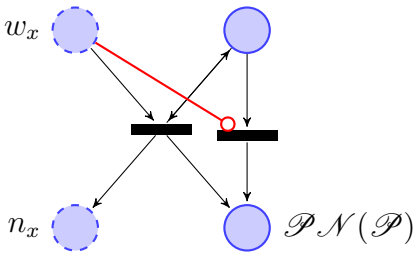



- Regarding the `notify()` and `notifyAll()` methods, they have been modeled using inhibitor arcs too.

$\mathscr{PN}(notify(x).P)=$




$\mathscr{PN}(notifyAll(x).P)=$



# Chapter 5

# The Compiler

In order to test our analyzer and obtain some results related to the problem of deadlock prevention of wait-notify methods in Java, I designed a compiler, using `ANTLR` and its API, able to transform a program written in the syntax described in Chapter 2 into `Java`. Then, it translates our language in a Petri Net graph, according to the models presented in Chapter 3. In this chapter, I will firstly present how `ANTLR` works, then my implementations choices.

## 5.1 ANTLR

`ANTLR`, ANother Tool for Language Recognition, [3], is a parser generator that uses $LL(*)$ for parsing. It takes in input a context-free grammar that specifies a language and produces a source code as output. `ANTLR` can automatically generate parsers (which can generate parse tree and abstract parse tree), lexers and tree parsers.
More specifically, `ANTLR` reads a grammar and generates a program that reads an input stream and if the input stream is not conform to the syntax of language defined in the grammar. If the syntax analysis does not generate an error (i.e. the program is syntactically correct), then the default action is simply to exit. Moreover, `ANTLR` also provides to check lexical correctness.

## 5.2 Implementation

### 5.2.1 The Grammar

In Chapter 2 I described the language to be analyzed, in this section I will present the corresponding grammar.

```
1  program     : term (SEMIC term)*  ;
2  term        : (left=statement (( op=operator ) right=statement )?)  ;
3  statement   : zero | new_object |new_thread | method | sync | waiting |
       ↪ notifying | notifyingAll ;
4  new_object     : LPAR NU ID RPAR (statement)*;
5  new_thread     : LPAR NU statement RPAR (statement)*;
6  method         : CHAR LPAR ( ID ( COMMA ID)* )? RPAR (ASM (ID | statement)
       ↪ )*  ;
7  sync           : SYNC LPAR ID RPAR LBRA (statement)+ RBRA (DOT statement)
       ↪ *;
8  waiting        : WAIT LPAR ID RPAR (DOT statement)* ;
9  notifying      : NOTIFY LPAR ID RPAR (DOT statement)*;
10 notifyingAll   : NOTIFYALL LPAR ID RPAR (DOT statement)*;
11 operator       : PLUS  ;
12 zero           : DIGIT ;
13
14 // token
15 SYNC      : 'sync';
16 WAIT      : 'wait';
17 NOTIFY    : 'notify';
18 NOTIFYALL : 'notifyAll';
19 NU        : 'nu';
20 DIGIT     : '0';
21 CHAR      : 'A'..'Z' ;
22 ID        : [a-z]+ ;
23 DOT       : '.';
24 COMMA     : ',' ;
25 SEMIC     : ';' ;
```

```
26 LPAR      : '(' ;
27 RPAR      : ')' ;
28 LBRA      : '{';
29 RBRA      : '}';
30 PLUS      : '+' ;
31 ASM       : '=';
32 WS        : (' '|'\t'|'\n'|'\r')+  {skip();};
```

I created some node classes, one for each rule of the grammar. Each class has two main methods: `toVisit()`, which generates the `Java` code and `petriNetGeneration()` that matches each method to its Petri net model. Before describe each of above functions in more details, it is necessary to introduce the semantic analysis.

### 5.2.2 Semantic Analysis

Semantic analysis is a process in compiler construction. This phase should guarantee that the syntax structure of the source program has meaning or not.
In my case, semantic analysis is expected to recognize:

- Methods `x.wait()`, `x.notify()` and `x.synchronized()` can be called only if object `x` has been defined before;
- `x.wait()`, `x.notify()` must always be in a `synchronized()` block.

I solved the first problem in the code generation part. In fact, when a `wait`, a `notify` or a `synchronized` method is called on an object `x`, the main compiler class `GrammarVisitorImpl.java`, that I will later discuss thoroughly, saves all objects names in a list and defines each one while creating output `Java` code. Concerning the second case, I avoided this problem with the `GrammarWalker.java` class. This class can be automatically generated by `ANTLR` and provides two types of methods for each rule of the grammar: `exitRule` and `enterRule`. I used a temporary array list of variables, `temp`, in which I added the `synchronized`

method when called (when a `enterSynch` is called) and remove it when the block ends. I also used an integer variable, `int counter`, that counts the level of the nested `synchronized`. Thus, when a `wait` or a `notify` is called, the program controls if the corresponding element in the `temp` array is null, if yes an error message is reported and the compiler does not terminate the code generation. Otherwise the code execution continues as before.

### 5.2.3 Java Code Generation

If the code is syntactically and semantically correct, the compiler can proceed with the `Java` code generation. For this purpose, the method `toVisit()` is called on the root of the abstract syntax tree. Each instruction of the grammar has a specific node in which all the needed information is saved, thanks to `GrammarVisitorImpl.java` class and its method that I describe deeply below.

#### `GrammarVisitorImpl` class

In this class I implemented the `visitStatement` method. It returns the visited rule-corresponding node with all the informations according on the type of the node I am visiting. To better understand what this method does, I report below the part of the code that returns a waiting node. An important variable of this class is `ArrayList<String> names` in which all the object names are saved.

```
1   if (ctx.waiting()!=null){
2         names.add(ctx.waiting().ID().getText());
3         if(ctx.waiting().statement(0)!=null){
4            return new WaitingNode(ctx.waiting().ID().getText(),
                 ↪ visit(ctx.waiting().statement(0)));
5         }
6         else{
7            return new WaitingNode(ctx.waiting().ID().getText());}}
```

In the if condition, the program verifies if the node is a waiting one, using the method `waiting()` automatically generated by `ANTLR`. Then, I add in the array `names` the name of the object in which the program calls the `wait()` method. At this point, the function controls if after the `wait()`there is another method call. If yes, it returns the wait-node and its children nodes after visiting them. Otherwise, the function returns only the current node and its execution ends.

### `toVisit()` method

In each `node` class, I implemented the `toVisit()` method. It returns the corresponding `Java` code as a string and call the `toVisit()` method for the next instruction. The `toVisit()` method code for the waiting node is reported below.

```
1 public String toVisit() {
2       if (stat!=null){
3          return "try {\r\n"+id+".wait();\r\n} catch (
               ↪ InterruptedException e) {\r\n};\r\n"+stat.toVisit()
               ↪ ;
4       }
5       else{
6          return "try {\r\n"+id+".wait();\r\n} catch (
               ↪ InterruptedException e) {\r\n};\r\n";
7       }
8 }
```

This method returns a string and calls itself for node `stat` which represents the next instruction of the input program. The method is similar for all other nodes.

### 5.2.4 Petri Net Graph Generation

After the `Java` code is generated, the compiler goes on with the Petri Net graph generation. The Petri Net graph models shown in Chapter 3 are saved in a .xml file. Thus, in this section, I'm going to describe how the compiler can merge multiple files in order to obtain a Petri Net graph for the whole input program.

#### `petriNetGeneration()` method

This method is implemented also in each `node` class. It adds in `ArrayList<Tuple> inputFiles` the corresponding Petri Net graph file according to models presented in Chapter 3. Now, I will briefly present some of the most useful classes and variables, that I implemented, in order to better understand what this method does.

- `filesNode` class: this class contains all the informations about the Petri Net files. Main class fields are:
  - **`Integer`** `whereToPut`: this integer can have values from 0 to 3. This number represents the "level" of the current instruction, in other words it indicates if there exists a paralell process or not. It will be used in the `ReadXmlFiles` class of which I will later discuss.
  - **`Integer`** `recursion`: 0 if there is not recursion, 1 otherwise.
  - `ArrayList<Tuple> inputFiles` described above.
- class `Tuple`: I created this class in order to have all files informations in one element of the array.
- `arrayFilesNode fileMethod` is a node field that contains all Petri Net files of methods declared in the input program with all their informations about recursion, the existence of a new thread and so on. In fact, it is an ArrayList of `filesNode`.

Here below is shown the code of the node notify `petriNetGeneration()` method.

```
1  public void petriNetGeneration() {
2     if(fileMethod.ArrayFiles.size()!=0 AND fileMethod.ArrayFiles.get
          ↪ (fileMethod.ArrayFiles.size()-1).whereToPutMethod == 1){
3        Tuple tuple = new Tuple("sixthCase.xml", file.whereToPut,
             ↪ fileMethod.ArrayFiles.get(fileMethod.ArrayFiles.size()
             ↪ -1).recursion);
4        fileMethod.ArrayFiles.get(fileMethod.ArrayFiles.size()-1).
             ↪ inputFiles.add(tuple);
5     }
6     else{
7        Tuple tuple = new Tuple("sixthCase.xml", file.whereToPut);
8        file.inputFiles.add(tuple);
9     }
10    if(stat!=null){
11       stat.petriNetGeneration();
12    }
13 }
```

At line 2, it checks if there exist some methods declared in the input program. If yes, the methods puts the corresponding Petri Net file in the `fileMethod` array or otherwise in the main array of files. Then, it goes on with the visit of all the parts of the program, if there are some other instructions in the body of the notify. Others `petriNetGeneration()` methods are very similar.

### `ReadXmlFiles` class

`ReadXmlFiles` is the main class of the compiler concerning the Petri Net graph generation. In fact, I merged all the xml Petri Net files using this class and its main method `readXMLfile(ArrayList<Tuple> inputFiles, int len)`. As an overview we can consider this function as a function that takes in input

the ArrayList of files and its length. At the beginning of the code, there is a for loop over all the input files, the function creates the new .xml file, called `mergedXml`, in the first iteration of the loop and then updates it with the new merged file. Thus, the function works with two .xml files at a time: the `mergedXml` (after the first iteration) and the one to add. It merges them deleting the last place of the first Petri Net graph and replacing it with the first place of the second file. Moreover, at each iteration the function updates the names of places, transitions and archs of the second file concatenating a letter taken from the array of string `LETTERS` which contains all the letters of the alphabet. That is because the names of the places shall be unique in order to properly execute a Petri Net graph and obtain the final markings. The $x$, $w_x$ and $n_x$ places are not changed, because there must be just one in common for each file. Now, I will discuss what this method does in more details.

```
1   for (int i=0; i<noDouble1.size(); i++){
2      if(nodeNames.contains(noDouble1.get(i))){
3         find1 = true;
4         temp1 = noDouble1.get(i);
5      }
6      if(nodeNames.contains(noDouble2.get(i))){
7         find2 = true;
8         temp2 = noDouble2.get(i);
9      }
10     if(nodeNames.contains(noDouble3.get(i))){
11        find3 = true;
12        temp3 = noDouble3.get(i);
13     }
14  }
15  if(nodeNames1.get(j).equals("x") AND find1){
16     Element toErase = (Element) nodes1.item(index);
17     toErase.getParentNode().removeChild(toErase);
```

```
18      flag1 = true;
19   }
20      else if(nodeNames1.get(j).equals("nx") AND find2){
21      Element toErase = (Element) nodes1.item(index);
22      toErase.getParentNode().removeChild(toErase);
23      flag2 = true;
24   }
25   else if(nodeNames1.get(j).equals("wx") AND find3){
26      Element toErase = (Element) nodes1.item(index);
27      toErase.getParentNode().removeChild(toErase);
28      flag3 = true;
29   }
30   else{
31      index ++;
32   }
```

The code above is the part of the function that works with $x$, $n_x$ and $w_x$ places. In particular, it checks if in the `mergedXml` file there are these places. If yes, the boolean flag `find` is set true and it looks for the same place in the current file. If it finds it, then it removes the place because they shall be unique and set boolean `flag` as true. This is the same for all the $x$, $n_x$ and $w_x$ places. In details:

- `nodeNames1` is the array list of string that contains all the names of the places of the second .xml file;
- `noDouble1`, `noDouble2` and `noDouble3` contain the strings $x$, $n_x$ and $w_x$, respectively and their concatenations with the elements of `LETTERS`;
- `find1`, `find2` and `find3` are the boolean variables that indicate that in `mergedXml` file there are $x$, $n_x$ and $w_x$ places;
- `find1`, `find2` and `find3` are the boolean variables set true if $x$, $n_x$ and $w_x$ places are also in the second file.

```
1  if(nodeNames.get(j).contains("nuPN") || j == indexToDel){
2     Node varNode =  nodes.item(j);
3     String name2 = ((Element) varNode).getAttribute("id");
4     Element tNode = null;
5     for (int k=0; k<nodes1.getLength() ; k++){
6           if(((Element) (nodes1.item(k))).getAttribute("id").
              ↪ contains("P0")){
7              tNode = (Element) nodes1.item(k);
8              var = k;
9           }
10    }
11    varNode.getParentNode().insertBefore(doc.adoptNode(nodes1.item(
          ↪ var).cloneNode(true)), varNode);;
12    varNode.getParentNode().removeChild(varNode);
13    tNode = (Element) nodes.item(j);
14    String name = tNode.getAttribute("id");
15    for(int k1 = 0; k1<arcs.getLength(); k1++){
16       if(arcNames.get(k1).contains(name2)){
17          Element temp = (Element) (arcs.item(k1));
18          String tempName = temp.getAttribute("id");
19          temp.removeAttribute(name2);
20          if(temp.getAttribute("source").equals(name2)){
21             String tempName2 = temp.getAttribute("source");
22             temp.setAttribute("source",tempName2.replace(name2,name
                  ↪ ));
23          }
24          if(temp.getAttribute("target").equals(name2)){
25             String tempName2 = temp.getAttribute("target");
26             temp.setAttribute("target",tempName2.replace(name2,name
                  ↪ ));
27          }
```

```
28          name2 = temp.getAttribute("source").concat(" to ").concat(
                ↪ temp.getAttribute("target"));
29          temp.setAttribute("id",tempName.replace(tempName,name2));
30          name2 = ((Element) varNode).getAttribute("id");
31       }
32    }
33 }
```

The previous code is a part of the `readXMLfile` function in which I deleted the last place of the `mergedXml` file and replace it with the first node of the next file. The integer variable `indexToDel` is the index of place that will be deleted. It is stored in the previous iteration of the for loop. In the code above, I check if the current node (index j) is the node to be deleted. If yes, I look for the first place of the current file, that is the place named $P_0$. When found, I store its index in `var`. Then, I add at `mergedXml` file this place and delete the old one. In the second part of the code, I change the arcs names: in fact, in order to merge the files, it is necessary to replace the target and/or the source of those transitions which goes from/to the deleted place. After that, the function insert all the remaining nodes, arcs and transitions in `mergedXml` file and saves the modified file. Thus, we obtain a Petri Net graph for the input program ready to be analyzed. At the end, I show what this function does with the following example.

**Example 7.** The aim of this example is show how the `readXMLfile` works. We want the Petri Net graph of the following sample program:

$$sync(x)\{\ notify(x).\ \}.$$

As shown in a previous chapter, the Petri Net model for `syncronized` and `notify` are the ones in Figure 5.1:

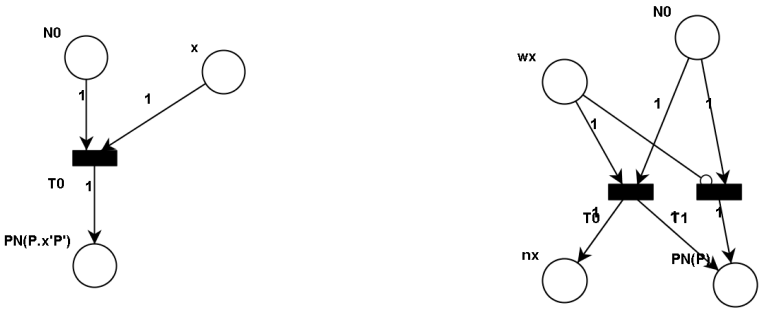



Figure 5.1: Petri Net graph for the `synchronized` and `notify` methods.

Thus, with the `petriNetGeneration()` method the compiler stores in an array the files corresponding to the Petri Net graph models, in this case they are the files of `synchronized` and `notify`, but also the one which is the representation of $\overset{x}{\bullet}$ 0 according to the transition relations described in Chapter 2. In Figure 5.2 is shown the resulting Petri Net graph. We notice

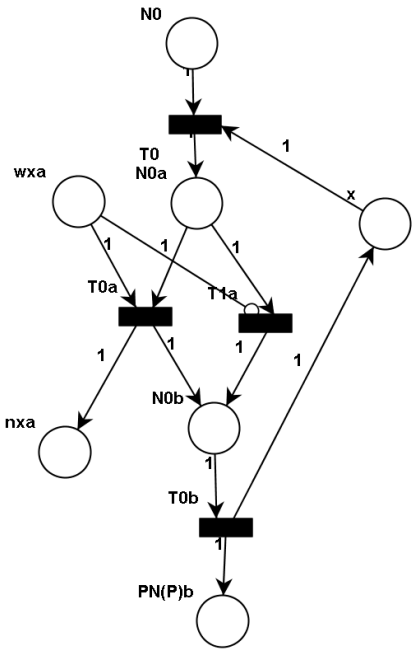



Figure 5.2: Example of Petri Net graph generation.

that there is only one place named $x$ and the other places and transitions

names are concatenated with a letter. In fact, the elments of the second file (the one about notify) are concatenated with the b and the ones of the third file with the c. This guarantees us the correctness of graph execution, because there are not places or transitions with the same name. Moreover, as described before, the last place of the `synchronized` model has been replaced with the first of the `notify` graph.

# Chapter 6

# Evaluation

In this chapter I will describe the results achieved for the analysis of deadlocks with Petri Net graph.

## 6.1 Results

I will describe how compiler works for examples presented in Chapter 2. Thus, they will be translated into `Java` and represented as Petri Net graphs thanks to models presented in Chapter 3.
Firstly, I'll show a sample program deadlock-free with an easy Petri Net graph. In the second example presented a deadlock may occur, depending on which process I decide to wake up. It the end, I will describe an example with recursion. Even if it is deadlock-free, this example is interesting to show because the deadlock-free condition in Petri Net graph is different from the first case.

### 6.1.1 A Deadlock-free Example

The first example is a sample program with only one object.

$$sync(x)\{\ (\nu\ sync(x)\{\ notify(x).wait(x).\ \}.\ )\ wait(x).notify(x).\ \}.$$

This code is deadlock-free as seen in Chapter 2. Our analysis gives the same result. The compiler has translated the program in `Java` code reported below. Moreover, it is represented by the Petri Net graph in Figure 6.1 obtained thanks to model reported in Chapter 3. The PIPE animation mode let us follow the graph execution. At the end of this phase, I obtain the final markings represented in the right image in 6.1. This configuration means that the program is deadlock-free because tokens are in the `x` and final places.

```
1  public static void main(String[] args) {
2     Object x = new Object();
3     synchronized(x){
4        Thread t1 = new Thread(){
5           public void run(){
6              synchronized(x){
7                 x.notify();
8                 try {
9                    x.wait();
10                } catch (InterruptedException e) {
11                };
12             }
13          }
14       };
15       t1.start();
16       try {
17          x.wait();
18       } catch (InterruptedException e) {
19       };
20       x.notify();
21    }
22 }
```

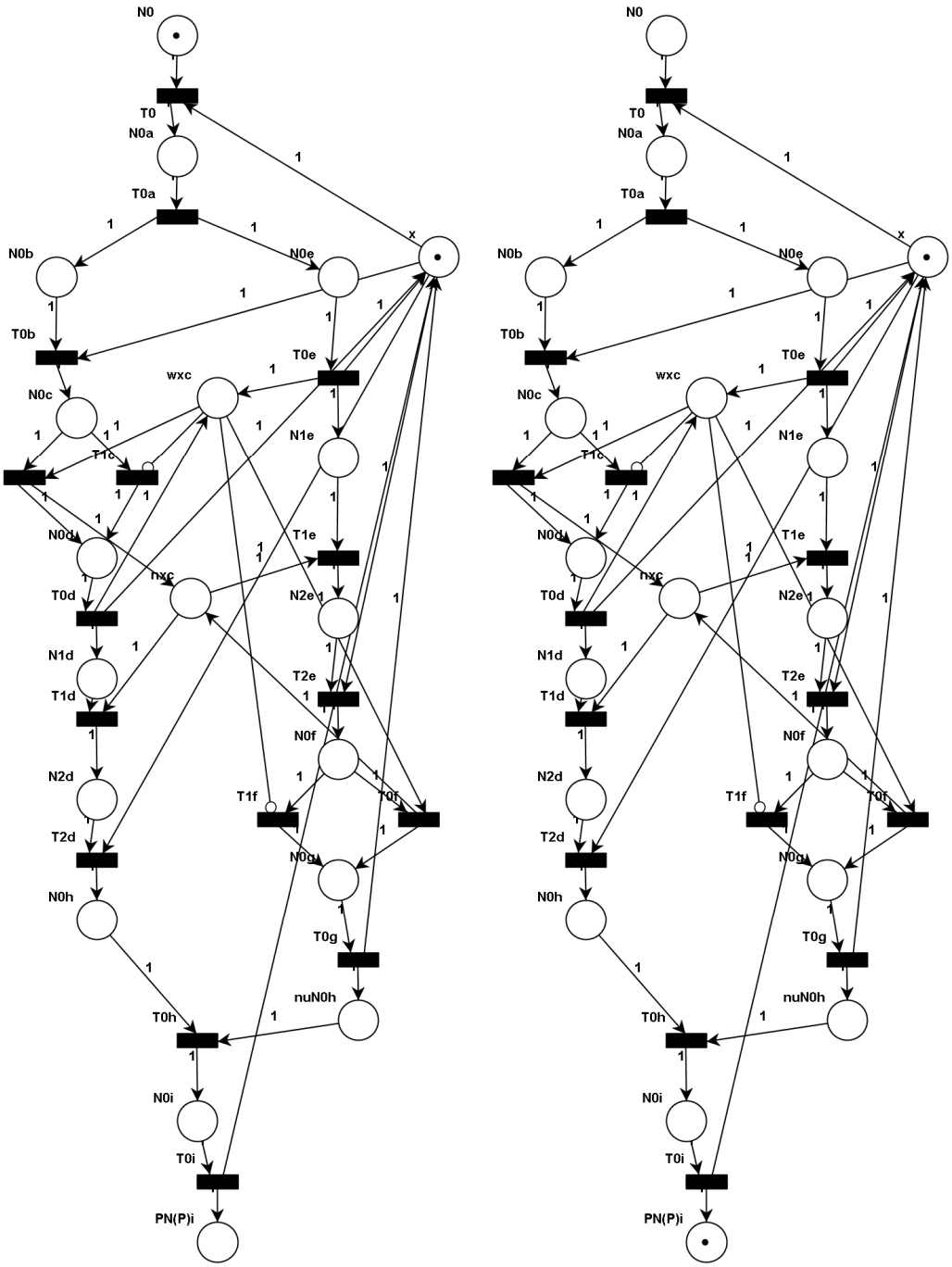



Figure 6.1: Petri Net graph and its execution.

The graph has two initial tokens, one in the x place and the other in the

first place of the graph. After the execution, one token is in `x` place and the other is in the final node. Thus, this representation of the program is correct because analysing the Petri Net graph I obtain that it is deadlock-free.

### 6.1.2 An Example with Deadlock

The following code is the one I will analyze using the technique developed in previous chapters.

$$(\nu\, sync(x)\{\ notify(x).wait(x).\ \}.\ )sync(x)\{\ \ wait(x).notify(x).\ \}.$$

In this case, I have two parallel processes because the $\nu$ is out of the first `synchronized`. So, the compiler gives the following `Java` code:

```
1   public static void main(String[] args) {
2      Object x = new Object();
3      Thread t1 = new Thread(){
4         public void run(){
5            synchronized(x){
6               x.notify();
7               try {
8                  x.wait();
9               } catch (InterruptedException e) {
10              };
11           }
12        }
13     };
14     t1.start();
15     synchronized(x){
16        try {
17           x.wait();
18        } catch (InterruptedException e) {
19        };
20        x.notify();}}
```

If I run the code above, a deadlock does not occur. This is caused by the fact that `Java` executes firstly the thread $t_2$ and then the second `synchronized` with its body. This is the same result shown in the graph in Figure 3.2 in Chapter 2. Thus, with a small change in the compiler I obtain the following code in which a deadlock occurs. The difference is that thread $t_1$ is the first to be executed.

```
1   public static void main(String[] args) {
2      Object x = new Object();
3      Thread t1 = new Thread(){
4         public void run(){
5            synchronized(x){
6               x.notify();
7               try {
8                  x.wait();
9               } catch (InterruptedException e) {
10              };
11           }
12        }
13     };
14     Thread t2 = new Thread(){
15        public void run(){
16           synchronized(x){
17              try {
18                 x.wait();
19              } catch (InterruptedException e) {
20              };
21              x.notify();}
22        }
23     };
24     t1.start();
```

```
25      t2.start();}
```

Figures 6.2 are the Petri Net graph and its final configuration for this example. The difference is that there is not the synchronize Petri Net model at the beginning of the graph and this is the cause of the deadlock. If I execute the $t_2$ part of the Petri Net graph before than the $t_1$ one, I will obtain a deadlock-free configuration and this is in line with the results gained before.

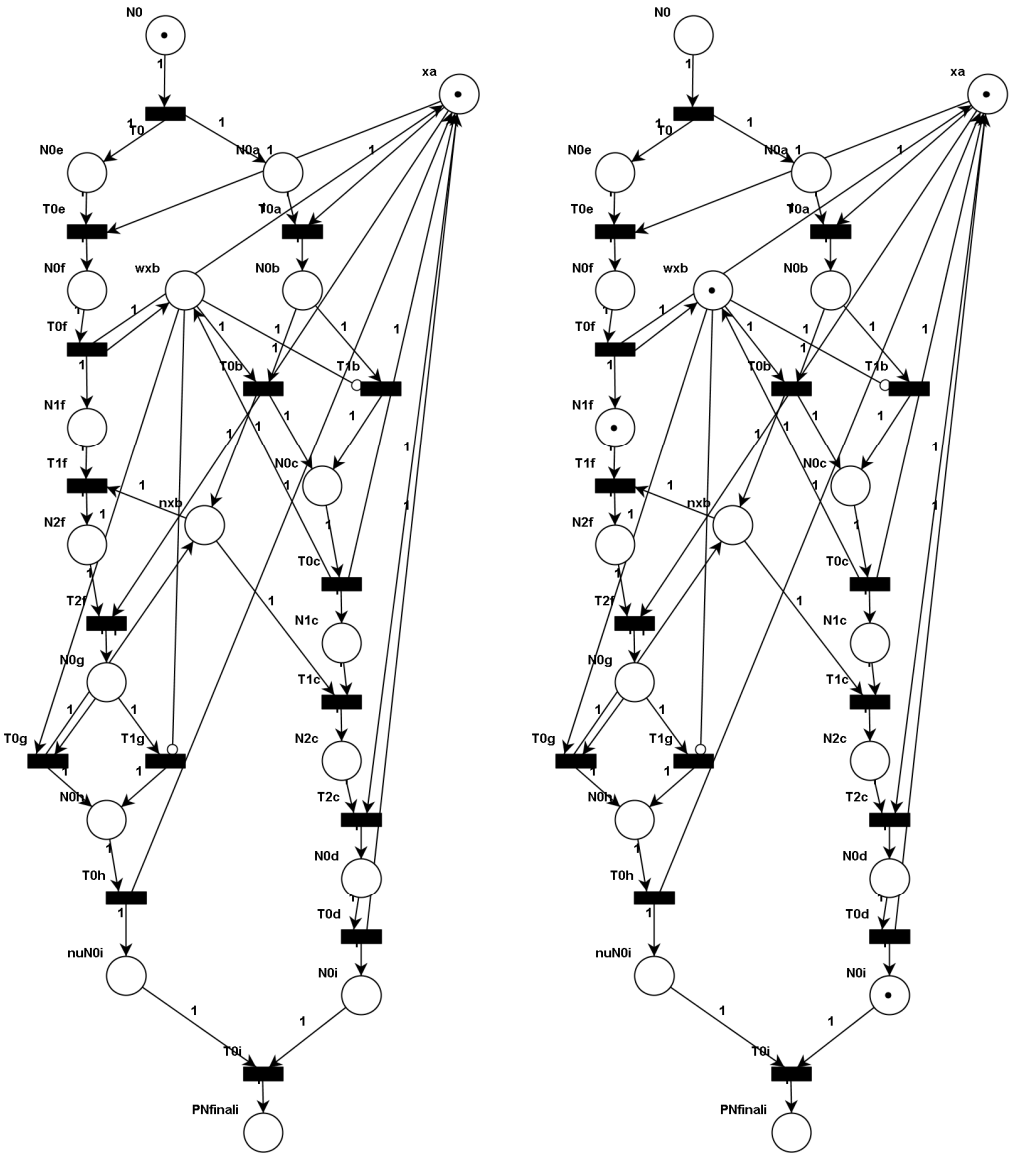



Figure 6.2: Petri Net graph and its final configuration.

### 6.1.3 A Recursive Program

In this section I will show an interesting example with recursion. The program analysed is the following:

$$F(x) = sync(x)\{\ notify(x).wait(x).\ \}.\ F(x)$$
$$G(x) = wait(x).notify(x).G(x)$$
$$sync(x)\{\ (\ \nu F(x))\ G(x)\ \}.$$

Below the corresponing `Java` code is shown. I focus my analysis on the Petri Net graph of this program, reported in Figure 6.3.

```
1   public static void F(Object x){
2      synchronized(x){
3         x.notify();
4         try {
5            x.wait();
6         } catch (InterruptedException e) {
7         };
8         F(x);
9      }
10  }
11
12  public static void G(Object x){
13     try {
14        x.wait();
15     } catch (InterruptedException e) {
16     };
17     x.notify();
18     G(x);
19  }
20
21  public static void main(String[] args) {
22     Object x = new Object();
23     synchronized(x){
24        Thread t1 = new Thread(){
```

```
25              public void run(){
26                  F(x);
27              }
28          };
29          t1.start();
30          G(x);
31      }
32  }
```

In Figure 6.3, on the right, the red transitions are the enabled ones. Performing the Petri Net graph of this example, I obtain always that configuration and it means that a deadlock does not occur. Thus, analysing Petri Net graph of the program it's possible to detect if a deadlock occurs. In this case this is very useful, in fact running the corresponding `Java` code the program does not end because of the recursion.

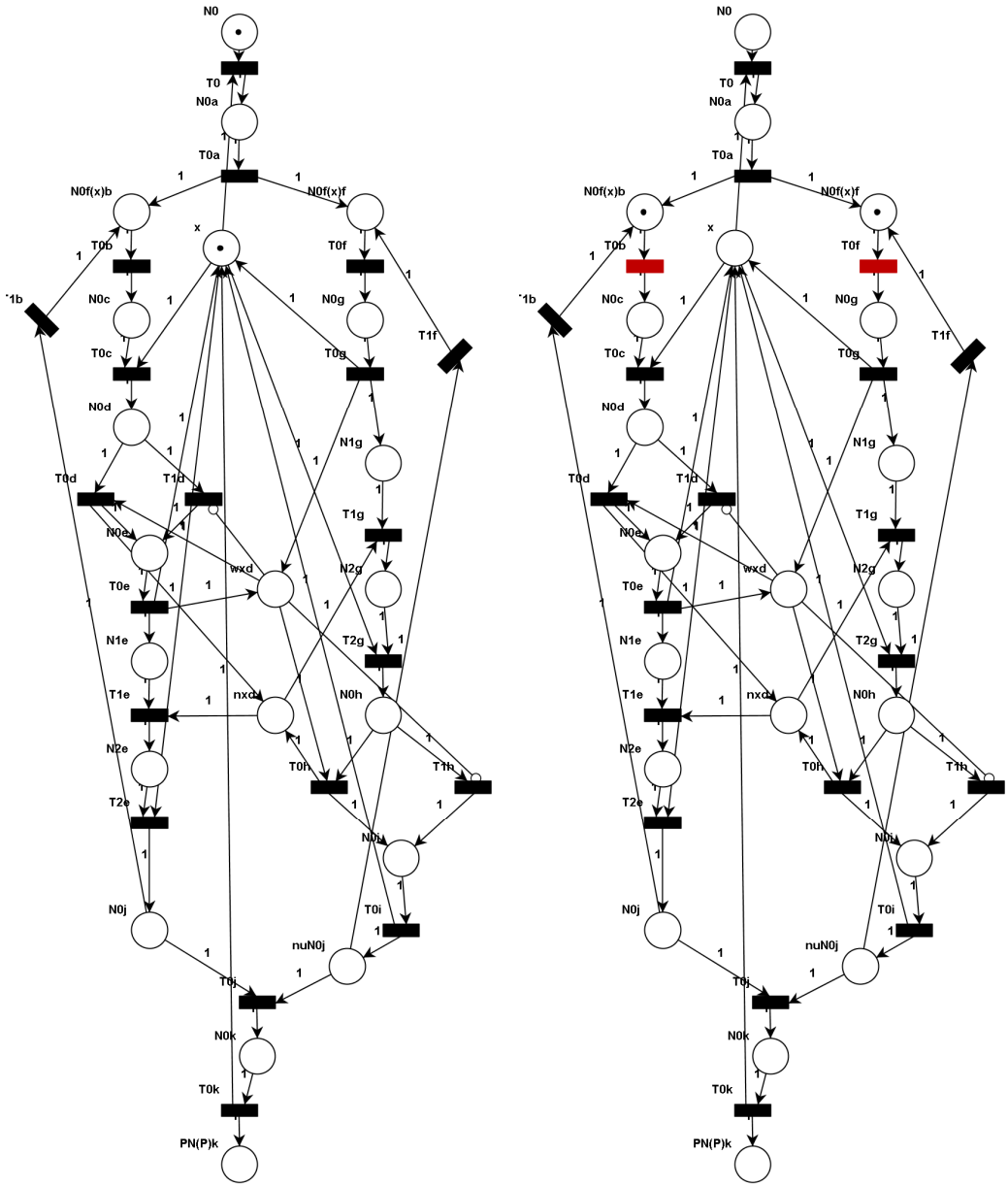



Figure 6.3: Petri Net graph and its configuration.

# Chapter 7

# Conclusions and Future Developments

In my thesis I have defined a technique for detecting `wait - notify` deadlocks in `Java` programs with one object. These methods modify the states of threads as regards locks: the thread executing `x.wait()` is suspended and the corresponding lock on $x$ is released; the thread executing `x.notify()` wakes up one thread suspended on $x$, which in vain will attempt again to grab $x$. Programming patterns with `wait - notify` methods may be faulty. In fact, it may happen that the `notify` is performed before a matching `wait`. Therefore the corresponding waiting thread may risk to blocked forever.

In this thesis, I have analyzed the correctness of `wait - notify` patterns (e.g. deadlock freedom) by using an analysis model that is a basic concurrent language with a formal semantic.
Every process of the language has been modeled into a Petri Net graph.
I designed a compiler that takes in input a program and returns the corresponding Petri Net graph for each process.
By means of an off-the-shelf solver for Petri Net (PIPE [15]), I have analyzed the reachability tree in order to check if a deadlock occurs or not.

This work is a basic step of a more complex and complete project, that aims at defining a general technique for detecting deadlocks in `Java`. In

particular, the technique defined in my thesis will be generalized to programs with two or more objects and combined with previous works [12, 6].

# List of Figures

# Bibliography


[1] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. "Types for safe locking: Static race detection for Java." In: 28 (Mar. 2006), pp. 207–255.

[2] Pierre America et al. "Operational Semantics of a Parallel Object-oriented Language." In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: ACM, 1986, pp. 194–208. DOI: 10.1145/512644.512662. URL: http://doi.acm.org/10.1145/512644.512662.

[3] *ANTLR: ANother Tool for Language Recognition*. http://www.antlr.org/.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks." In: *SIGPLAN Not.* 37.11 (2002), pp. 211–230. ISSN: 0362-1340. DOI: 10.1145/583854.582440. URL: http://doi.acm.org/10.1145/583854.582440.

[5] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. "Type Inference for Atomicity." In: (Jan. 2005), pp. 47–58.

[6] Abel Garcia and Cosimo Laneve. "Deadlock detection of Java Bytecode." In: *CoRR* abs/1709.04152 (2017). arXiv: 1709.04152. URL: http://arxiv.org/abs/1709.04152.

[7] Abel Garcia and Cosimo Laneve. "JaDA – the Java Deadlock Analyzer." In: *Behavioural Types: from Theory to Tools*. Ed. by Simon Gay and Antonio Ravara. River Publishers, 2017, pp. 169–192. URL: https://hal.inria.fr/hal-01643216.

[8] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. "Deadlock Analysis of Unbounded Process Networks." In: *CONCUR 2014 – Concurrency Theory*. Ed. by Paolo Baldan and Daniele Gorla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 63–77. ISBN: 978-3-662-44584-6.

[9] Elena Giachino and Cosimo Laneve. "Deadlock Detection in Linear Recursive Programs." In: *Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. Ed. by Marco Bernardo et al. Cham: Springer International Publishing, 2014, pp. 26–64. ISBN: 978-3-319-07317-0. DOI: 10.1007/978-3-319-07317-0_2. URL: https://doi.org/10.1007/978-3-319-07317-0_2.

[10] Naoki Kobayashi. "A New Type System for Deadlock-Free Processes." In: *CONCUR 2006 – Concurrency Theory*. Ed. by Christel Baier and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 233–247. ISBN: 978-3-540-37377-3.

[11] Naoki Kobayashi and Cosimo Laneve. "Deadlock analysis of unbounded process networks." In: *Information and Computation* 252 (2017), pp. 48–70. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2016.03.004. URL: http://www.sciencedirect.com/science/article/pii/S0890540116000419.

[12] Cosimo Laneve. *A lightweight deadlock analysis technique of object-oriented programs*. 2018.

[13] R. Milner. *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982. ISBN: 0387102353.

[14] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, II." In: *Information and Computation* 100.1 (1992), pp. 41 –77. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90009-5. URL: http://www.sciencedirect.com/science/article/pii/0890540192900095.

[15] *PIPE: Platform Independent Petri net Editor 2*. http://pipe2.sourceforge.net/.

[16] Kohei Suenaga. "Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References." In: *Programming Languages and Systems*. Ed. by G. Ramalingam. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 155–170. ISBN: 978-3-540-89330-1.

[17] Vasco T. Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. "Type Inference for Deadlock Detection in a Multithreaded Typed Assembly Language." In: *Proceedings of PLACES'09 - Programming Language Approaches to Concurrency and Communication-cEntric Software*. Vol. 17. EPTCS. 2010, pp. 95–109.

[18] Akinori Yonezawa, ed. *ABCL: An Object-oriented Concurrent System*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-24029-7.