UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

How to Grind JAVA Programs to Extract Full-bodied Infinite-state Models ?

Mémoire présenté en vue de l'obtention du Diplome d'Études Approfondies en Sciences (filière Informatique et Mathématiques)

Année Académique 2002-2003

GEERAERTS Gilles

This document has been typeset under $\LaTeX\,2_\varepsilon$, using the `book` document class.

Contents

1 Introduction **1**
1.1 The verification problem for concurrent software . . . 1
1.2 Concurrency support in the JAVA language . . . 1
1.2.1 Creation of threads . . . 2
1.2.2 Synchronisation between threads . . . 2
1.3 A verification framework proposal . . . 4
1.3.1 Sketch of the verification process . . . 5
1.3.2 Optimisations . . . 5
1.4 Plan of the work . . . 5

2 The Concurrent Boolean Programs **7**
2.1 Syntax . . . 7
2.1.1 Well-Formed Concurrent Boolean Programs . . . 9
2.2 Semantics . . . 10
2.2.1 Definitions . . . 10
2.2.2 Axioms . . . 12
2.2.3 Execution of a Concurrent Boolean Program . . . 15
2.3 Discussion . . . 15

3 From JAVA to CBP and from CBP to the Global Machines **17**
3.1 From Java Programs to CBPs . . . 17
3.2 From CBP to Global/Local Machines . . . 21
3.2.1 The Global/Local Machines (from [DRVB02b]) . . . 21
3.2.2 Reachable Programs . . . 22
3.2.3 Translation . . . 24
3.3 Relation between the CBPs and the GMs . . . 27
3.4 The CBP2GM compiler . . . 30
3.5 Discussion . . . 31

4 Optimising the Model **35**
4.1 The need for optimisation . . . 35
4.2 Stoller's model checking techniques for distributed JAVA programs . . . 35
4.2.1 System model . . . 36
4.2.2 Visible and invisible . . . 36
4.2.3 Conditions on systems . . . 37
4.2.4 A coarser model thanks to invisible states . . . 37
4.2.5 Smaller Local Machines thanks to invisible states . . . 38
4.3 Static analysis techniques . . . 41
4.4 Related Works . . . 43
4.4.1 On the reduction problem . . . 45
4.4.2 On reducing the exploration of a system's state-space . . . 45
4.4.3 On the problem of shape analysis . . . 45

4.4.4 On various compiler optimisation techniques . . . 46
4.5 Discussion . . . 46

5 Conclusion and Future Works **47**

A A case study: The Bakery algorithm **49**
A.1 The Bakery Algorithm in JAVA . . . 49
A.2 The Bakery Algorithm in CBP . . . 53

B Proof of Lemma 3.2 **57**

List of Figures

1.1 A simple sketch of concurrent program using JAVA's concurrency support. 2
1.2 A simple wait - notify example. 3
1.3 Our framework to verify safety properties on multi-threaded JAVA programs. 6

2.1 Grammar for the Concurrent boolean programs 8
2.2 A simple Concurrent Boolean Program. 9
2.3 The definition of $\gamma(S, \ell)$. 13
2.4 The rendezvous class 16

3.1 A simple example of JAVA class 18
3.2 JAVA thread classes manipulating a shared Point object 18
3.3 The inc and dec CBP threads corresponding to the Inc and Dec JAVA threads of Figure 3.2 20
3.4 Graphical representation of the inductive proof of Lemma 3.1. 24
3.5 Graphical depiction of the inductive step to prove Lemma 3.2 30
3.6 The Local Machine for the thread type Inc, generated by CBP2GM. 32
3.7 The Local Machine for the thread type Dec, generated by CBP2GM. 33

4.1 The transition system for the dec thread 39
4.2 The reduced transition system for the dec thread 40
4.3 The reduced Local Machine for the dec thread of Figure 3.2 42
4.4 An example of SSG 44

Acknowledgments

This work has been realised under the supervision of Prof. Jean-François Raskin, and in close collaboration with Laurent Van Begin (almost PhD). I take the opportunity that is offered to me here, to address them both many thanks for their scientific support and invaluable friendship.

This work should be published soon as a technical report of the Université Libre de Bruxelles.

The typestting has been carried out under LaTeX 2$_\varepsilon$. Christophe Fiorio's wonderful Algorithm2$_\varepsilon$ package has been used to typeset the algorithms. The *pretty printing* of the JAVA sources has been made partly thanks to the `highlight` software, by André Simon. The watermark on the cover is the seal of the University (Sint Michael killing a beast representing the obscurantism). I used Bernard Desruisseaux's `\BackgroundEPS` macro to achieve this. The picture has been scanned from an old document of the University[1].

[1] Namely, an academical report for the academical year 1968-1969, published in 1971, that I found under a heap of dust in the library of the Computer Science Department.

Chapter 1

Introduction

1.1 The verification problem for concurrent software

The formal verification of computer software is a field of computer science that has been consuming a large research effort for about twenty years. Verifying properties of software became an increasing point of concerns around the eighties, when computer scientists began to be aware of the power of the formal techniques that had then successfully been applied to check electronic circuits against their specifications.

Of course, the former problem is far more complex than the latter, because the object of the analysis – that is, a human-written piece of code in a high-level programming language – has a much richer behaviour than an electronic circuit ; which, despite its possibly huge size, just manipulates boolean values in a simple fashion. One reason to this complexity is the *data domain* of computer software, that is conceptually infinite (think about the range of values of an `int` variable in C, for instance).

The most classical solution to this problem is *predicate abstraction* [GS97]. The basic idea is to define an *abstraction function* that maps the infinite domain of the variable to a finite domain of predicates, which give reduced but yet useful information about the variables' concrete valuation. The piece of code is then modified to manipulate only boolean variables, which encode the values of the predicates. Each modification of a variable in the concrete domain should be translated into a *predicate modifier* that reflects the change in the abstract domain. For instance, someone being interested in knowing whether a integer variable `x` is positive could just retain a single predicate $p \equiv \texttt{x} \geq 0$, whose value could be encoded in a boolean variable, say `xpos`. We will come back to predicate abstraction later.

Another reason to the difficulty of analysing computer software lies in *concurrency.* A program is said to be *concurrent* whenever its execution-flow is not linear, but distributed among several pieces of code (often called *threads*) that can execute *in parallel.* Moreover, the verification problem can often be stated in terms such that one doesn't know *a priori* how many instances of a given type of thread will be run. In this case, one speaks about *parameterised verification* [GS92].

This last problem is the one we are going to address in this thesis. More precisely, we are going to look into the verification of concurrent JAVA software [DRVB02b], which has the peculiarity to include *built-in support for concurrency*, as described in the next section.

1.2 Concurrency support in the JAVA language

The JAVA language is inherently multi-threaded (that is, concurrent). To be able to implement concurrency in a program, one needs two important features: the ability to declare *independent pieces of code* able to run in parallel (this is provided in JAVA through the `Thread` object type), and a set of *communication primitives* to let the threads synchronise or exchange messages (JAVA includes such keywords as `wait`, `notify`, `notifyAll`, a.s.o.).

```
public class myThread implements Runnable {

    /* Some private data */

    public myThread(...){
        /* Constructor's code */
    }

    public void run() {
         /* The code here will be executed once the thread starts */
    }
}
public class mySystem {

    public static void main(String[] args) {

        Thread T1, T2 ;

        myThread M1 = new myThread(...)  ;
        myThread M2 = new myThread(...)  ;

        T1 = new Thread(M1) ;
        T2 = new Thread(M2) ;

        T1.start() ;
        T2.start() ;
    }
}
```

Figure 1.1: A simple sketch of concurrent program using JAVA's concurrency support.

1.2.1 Creation of threads

A `Thread` object is usually constructed using the `Thread(Runnable target)` constructor, which expects an object implementing the `Runnable` interface as parameter. The `Runnable` interface only requests the object implementing it to define a `run` method. This method is the one to be called when the corresponding thread is *started*, through the `start` method of class `Thread`.

Example 1 Figure 1.1 presents an example of what we have just sketched above. □

Another way of achieving the same result is through inheritance: an object is a thread if it inherits from the `Thread` class (as long as it implements the `start` method).

1.2.2 Synchronisation between threads

JAVA gives access to several synchronisation primitives one can use in the code of a thread. But before speaking about these features, and in order to let the reader fully understand their intended use, we should mention an important characteristic of the multi-threading in JAVA. Although each thread has a private memory area in which it can allocate local variables the other threads *can't* access, all the threads *share a common part of the memory*. For instance, all the statically- and heap-allocated objects can possibly be accessed by any thread holding a valid reference to it (that has been obtained through the parameters of the constructor, for instance).

One now easily understands the crucial need for *synchronisation* primitives, in order to maintain the coherence of the shared objects through *mutually exclusive access* to these objects. To achieve this, JAVA associates a *lock* to each object. Whenever we want a thread to execute some critical section `I` manipulating an object `O`, we can protect the instructions with `synchronize(O){ I }`, for instance. The effect will be to

```
public class O {
    public void m1()
    {
        /* ... */
        wait() ; /* equivalent to this.wait() ; */
    }

    public void m2()
    {
        /* ... */
        notify() ; /* equivalent to this.notify() ; */
    }
}
```

Figure 1.2: A simple `wait` - `notify` example.

take the lock associated with object `O` before executing the instructions `I`, and to *release* it afterwards. Of course, if the lock was already hold by another thread, then the thread attempting to synchronise is blocked and put in a wait-set associated to the object. Once the lock is relinquished, a thread arbitrarily chosen among the threads of the wait-set, and is elected to take the lock.

A thread can also chose to go to a special *sleep* state, in which it is inactive, up to its awakening by another thread. To let a thread sleep, one uses the `wait` method of the `Object` class[1]. Again, this inactive state is associated to a given object (in this case, the object on which the `wait()` was called) *and* a given point of code (the place where the `wait` is called). To awaken one arbitrarily chosen thread among those that went to sleep on a given object, one uses the `notify()` method on this object. To awaken all the threads, one uses the `notifyAll()` method.

Example 2 For instance, look at the piece of code of Figure 1.2. If a thread calls method `m1` on a given instance `i` of class `O` it will eventually go to sleep. One way to awaken it, is that another thread calls the `m2` method on the same instance `i`. If this second thread calls `m2` on another instance, the `notify()` statement will have no effect on both threads. □

Example 3 As a more complete example, we present here an implementation of Lamport's famous Bakery algorithm for mutual exclusion [Lam74]. The idea of the protocol is to regard the threads as customers of a bakery. Each thread has to draw a ticket when entering. The threads get served in order of their ticket numbers [Lyn97]. The sketch of the algorithm is given at Algorithm 1. The JAVA multi-threaded implementation can be found in Appendix A (A.1).

As one can see in Algorithm 1, each thread simply 'scans' all the tickets of its neighbours (by looking into their respective *number*), and chooses for itself the maximum of all the tickets, plus one (that is, the very next number to be served). The *choosing* field is set to one during this selection process. A problem might arise when two threads execute this sequence of instructions concurrently, which might result in two such threads having the same ticket number. This however is solved by also taking into account the identities of the threads, in the test that admits them to the critical section: $number(j) = 0 \vee (number(i), i) < (number(j), j)$.

The JAVA implementation is built around three main classes:

1. The `ticket` class models the tickets, and is simply a container for an integer and a boolean value. They correspond to *number* and *choosing* in the Algorithm 1, respectively. The `tear` function serves as a reset for the value of the ticket.

2. The `myThread` class inherits from the `Thread` class. Each such thread owns a personal ticket `myTicket` and an unique identifier `myId` (an integer). A reference to the bakery `theBakery`, shared by all threads, is also kept in each instance of this class. After having obtained a ticket through the `getTicket` function of the bakery, the thread tries to enter the critical section by calling `theBakery.getTurn()`. It will remain blocked in this function as long as unserved threads with a higher priority remain. The critical

[1] Remember the `Object` class is the super-class of every class in JAVA.

section in itself is 'simulated' by a small pause of random length. On the end of the critical section, the thread gets rid of its ticket and notifies the other threads that it exits, thanks to the `getOut` function.

3. The `bakery` class manages an array `T[]` of tickets (instance of the `ticket` class). The `getTicket` function is called by a thread entering the bakery and requesting a ticket and is a straightforward JAVA translation of lines 1 - 3 of the algorithm. The `getTurn` function implements the **foreach** loop of Algorithm 1 (line 4). The thread calling this function can get blocked at two different points. First, the call to `getChooseValue()` on another thread's ticket can block if the owner of the ticket is busy setting its value (this corresponds to line 5 in the algorithm). Then, the thread can get blocked if has not the highest priority among the threads waiting to be served (*cf.* line 6 in the algorithm). Finally, when the `while` loop is left, the thread is elected to enter the critical section.

□

The previous example shows clearly when the use of `notifyAll` is necessary. In this case, each thread has its own identity, and we want to wakeup the one with the highest priority. However, JAVA offers no mechanism to select the thread to be awoken, and we are not guaranteed we will re-activate the very thread we are interested in. Therefore, we had to wakeup all of them, and let them check by themselves if they are allowed to go on running, or if they have to go back waiting.

In the case where all the threads are symmetrical, and one needs to wake up a single thread, then one can use the `notify()` statement instead of `notifyAll()`.

Algorithm 1: Lamport's Bakery algorithm (from [Lyn97])

Shared Variables ($\forall 1 \leq i \leq n$)**:**
$choosing(i) \in \{0,1\}$: initially 0, writable by i and readable by $j \neq i$
$number(i) \in \mathbb{N}$: initially 0, writable by i and readable by $j \neq i$
Algorithm for thread i**:**

1 $choosing(i) \leftarrow 1$;
2 $number(i) \leftarrow 1 + \max_{j \neq i} number(j)$;
3 $choosing(i) \leftarrow 0$;
4 **foreach** $j \neq i$ **do**
5 | **waitfor** $choosing(j) = 0$;
6 | **waitfor** $number(j) = 0 \vee (number(i), i) < (number(j), j)$;
7 Critical section ;
8 $number(i) \leftarrow 0$

This quick introduction should be sufficient to help the reader understand the sequel of this work. That's why we won't look any further into the details of the implementation of concurrency in JAVA. More information can be found in [Gra97, Fla97], for instance.

1.3 A verification framework proposal

In order to verify properties on multi-threaded JAVA programs, with an *unbounded* number of threads, we propose a framework built around the language of Concurrent Boolean Programs (or CBP for short). The language of CBP is a *specification language* that allows one to describe the behaviour of concurrent programs with *variables ranging over a boolean domain*. It can be regarded as a multi-thread extension of the Boolean Programs introduced by Ball and Rajamani [BR00][2]. This language aims at being syntactically close to JAVA, which should make easier the translation of a JAVA program into a corresponding CBP. We fully describe the syntax and semantics of the language of CBP in Chapter 2.

[2] We wouldn't be complete without mentioning Ball's and Rajamani's own multi-threaded extension of the Boolean Programs, presented in [BCR01b]. It is however important to remark that the synchronisation primitives considered in this paper are less powerful than the *broadcasts* we are about to consider in this work. Indeed, the *Local/Global Finite Sates Machines* – this is the name given to the automata considered there – can be translated into a regular Petri net. To cope with our model, we need to define a broader class of Petri nets: the Multi-Transfer nets (see [DRVB02b]).

1.3.1 Sketch of the verification process

Here are the main steps of the framework

1. Given a multi-threaded JAVA program, we translate it into a CBP. This can be done thanks to predicate abstraction [GS97]. Therefore, we have to be able to find an interesting sets of predicates, small enough to keep the resulting program tractable, be precise enough to allow us to verify the properties we are interested in. We won't address this really interesting matter in this thesis, although we plan to work on it in the future.

2. We translate the resulting CBP into the formalism of Global/Local machines, as introduced by Delzanno, Raskin and Van Begin in [DRVB02b]. Roughly speaking, a Global machine is a collection of Local machines plus a set of global boolean variables. Each Local machine is an automaton which can synchronise with the other Local machines through *rendez-vous*, asynchronous *rendez-vous*, and broadcasts. It has been showed in [DRVB02b] that this model is well-suited to describe the behaviour of multi-threaded JAVA programs.

3. In the same paper, the authors have showed how to translate a Global machine into an Multi-Transfer Net (Multi-Transfer Nets are an extension of regular Petri nets, which allow to *flush* all the token from one place to another). We apply this step to the Global machine we have obtained so far.

4. It is then possible to verify *reachability properties*, provided that the unsafe set of states be represented by an *upward-closed set of markings* [DRVB01]. Efficient symbolic methods have been devised in [DRVB02b] to carry on this step, using the Covering Sharing Tree data structure [DRVB02a, DR00]. Moreover, we have shown in our master's thesis [Gee02] that other symbolic data structures could be used too. All these methods have been implemented in a tool called BABYLON [ADG+02].

1.3.2 Optimisations

If this framework were to be applied 'as is', the generated model would surely become intractable. Therefore, many optimisations need to be applied, at the various steps of the translation. Several of them have already been studied, namely:

1. **At the JAVA program step**: Several static analysis techniques have up to now been developed for JAVA. For instance, one can try to attack the overload induced by *unnecessary synchronisations* [BH99]. Most of these techniques use *shape analysis* [CWZ90, Cor00] and *escape analysis* [CGS+99].

2. **At the Global/Local machine step**: Techniques like those presented by Corbett in [Cor00] could be applied in order to eliminate *invisible transformations*, that is, manipulations of variables that are *purely local to a thread* (because, the manipulated data can't escape the thread's scope).

3. **A the Petri net level**: Several Petri net reduction techniques exist that preserve the properties we are interested in [Ber85]. One could also try to reduce the size of the net using structural invariants, in a fashion similar to [DRVB01].

We investigate these works in Chapter 4.

1.4 Plan of the work

Chapter 2 presents the language and the semantics of the Concurrent Boolean Programs, an extension of Ball & Rajamani's Boolean Programs. Our extension features concurrency primitives such as *broadcasts* and (asynchronous) *rendez-vous*.

Chapter 3 sketches briefly the translation scheme of a JAVA program into a CBP, using the predicate abstraction technique. We then recall the notion of Global/Local machine (previously introduced in [DRVB02b]), and show how to translate a given CBP into a set of Global Machines, suitable for the verification of safety properties. The trace equivalence between the models is proved, and we introduce `CBP2GM`, a tool that automatises the translation of a CBP into a GM.

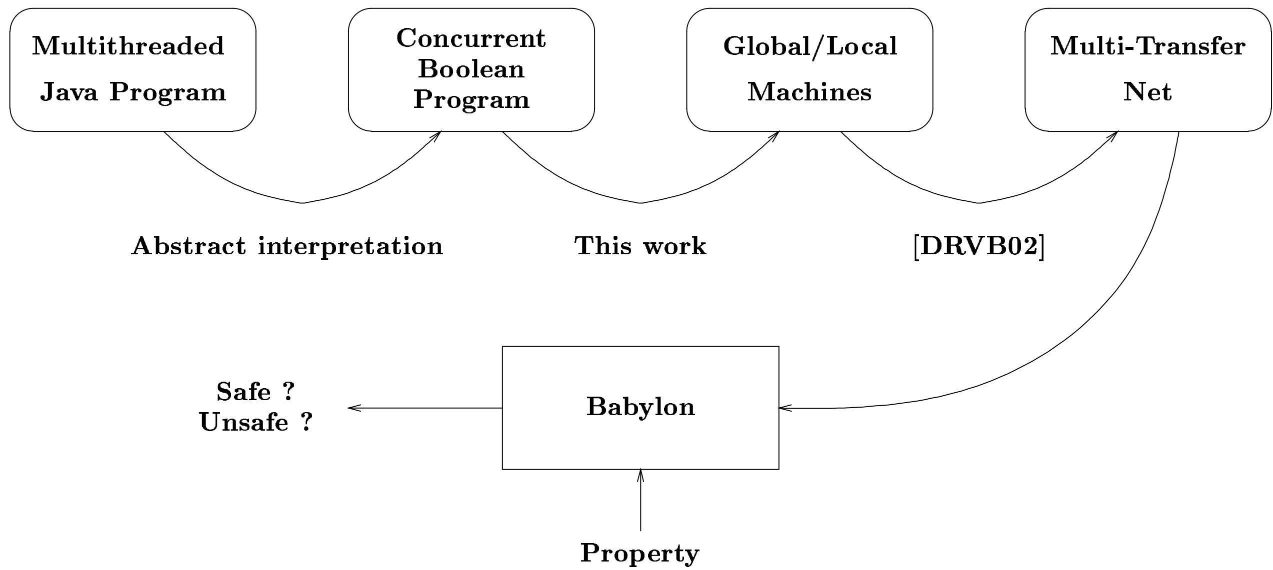


Figure 1.3: Our framework to verify safety properties on multi-threaded JAVA programs.

Chapter 4 explains what are the optimisations one could try to apply at the various step of the framework, in order to reduce as much as possible the size of the resulting model. This section can be seen as a kind of mini-survey of the literature in this domain.

Chapter 5 draws some conclusions and tries to suggest some new paths of research.

Chapter 2

The Concurrent Boolean Programs

In this chapter, we introduce the *language of CBPs*[1] which are the cornerstone of our verification framework, as we plan to translate JAVA programs into CBPs and CBPs into GMs. In 2.1, we present the *syntax* of this language as well as a *first definition* of what is a CBP. In 2.2, we present the semantics of the language, thanks to a second, more formal definition of CBP (the two definitions are showed however to be equivalent). The semantics is stated through a set of *axioms*, in a fashion similar to the discussion of chapter 8 of [AO97].

2.1 Syntax

We begin by introducing the syntax of CBP. When designing this language, we have tried to keep as close as possible to JAVA programing language, but without losing the ability to translate other multi-threaded languages (like Ada or C with POSIX threads) into CBP. This should make the translation quite straightforward in a first approach.

As said in the introduction, CBPs can be regarded as a natural concurrency-aimed extension of the Boolean Programs, defined by T. Ball and S.K. Rajamani (see, for instance, [BR00]). As the name suggests, a CBP is made of several pieces of code we will call *threads*, in a fashion similar to Java. These threads are rather independent processing units, that will be able to run in *parallel*, following an interleaving semantics (as we will show in the next section). More precisely, we have extended Ball and Rajamani's Boolean Programs by means of the following primitives: `sleep`, `wakeup`, `wakeupall`, `start`, `accept` and `rendezvous`. In order to fully capture the synchronisation semantics of Java, which requires the ability to let a thread lock an object, we have also added the concept of *lock*.

It is worth remarking that the *boolean* variables handled by a CBP might be of two types. From the one side, we have the *global variables*, that can be read and written to by any thread. From the other side, we have several sets of local variables, which belong to a single thread, and can be accessed by their sole owner.

In this section we give the necessary syntactic rules to write a well-formed CBP. The first (classical) step toward this achievement, is to state the *grammar* of CBP. One can find it at Figure 2.1. We have taken the convention to write the *keywords* of the language (as `start` or `sleep`) in `typewriter face` ; and all the other terminals (like the variables names: **varName**) in **bold face**.

Example 4 An example of a very simple CBP is given at Figure 2.2. It comprises three threads: `t1`, `t2` and `main`. The program begins with a declaration section, as every CBP. One first declares the global variables (in this case: `g1` and `g2`), the locks (no locks are declared in this example), the messages (a single message `msg` is declared here) and the name of the type of threads. The definition of each type of thread – which begins by the declaration of the local variables – is then given.

We can already briefly sketch the semantics of this example: the `main` thread will *start* one copy of `t1` and one copy of `t2` (assigning `false` to local variable `l1` and `true` to `l2`). The running instance of `t2` will try to synchronise with the instance of `t1` through the `accept` statement (the receiving side of the *rendez-vous*).

[1] In the sequel, we will use the notation CBP to speak about *the language of CBPs*, as well as a Concurrent Boolean Program – a piece of code written in the language of CBP. The context should be sufficient to let the find out the which meaning to should be applied...

[1]	Program	→	Declarations Threads$^+$
[2]	Declarations	→	VarDec LockDec MsgDec ThreadDec
[3]	VarDec	→	`vars :` **varName*** `;`
[4]	LockDec	→	`locks :` **lockId*** `;`
[5]	MsgDec	→	`messages :` **msgId*** `;`
[6]	ThreadDec	→	`threads :` **threadTypeName*** `;`
[7]	Threads	→	**threadTypeName** { VarDec ThreadBody }
[8]	ThreadBody	→	LabeledCommands$^+$
[9]	LabeledCommands	→	**[label]** Command
[10]		→	Command
[11]	Command	→	Assignment `;`
[12]		→	`if` (Condition) { LabeledCommands } `else` { LabeledCommands }
[13]		→	`choice` { GuardedAssign$^+$ }
[14]		→	`while` (Condition) { LabeledCommands }
[15]		→	`skip ;`
[16]		→	`goto` (**label**) `;`
[17]		→	`lock` (**lockId**) `;`
[18]		→	`unlock` (**lockId**) `;`
[19]		→	`sleep` (**msgId**, **lockId**) `;`
[20]		→	`wakeup` (**msgId**) `;`
[21]		→	`wakeupall` (**msgId**) `;`
[22]		→	`start` (**threadTypeName**) `;`
[23]		→	`start` (**threadTypeName**, Parameters) `;`
[24]		→	`rendezvous` (**msgId**) `;`
[25]		→	`rendezvous` (**msgId**, ExpressionList) `;`
[26]		→	`accept` (**msgId**) `;`
[27]		→	`accept` (**msgId**, VariableList) `;`
[28]	Condition	→	`*`
[29]		→	ExpConjunction
[30]	ExpConjuction	→	Expression
[31]		→	Expression `and` ExpConjunction
[32]	GuardedAssign	→	Condition `:` Assignment `;`
[33]	Assignment	→	**varName** AssignMiddle Expression
[34]	AssignMiddle	→	, **varName** AssignMiddle Expression,
[35]		→	`:=`
[36]	Parameters	→	Litteral, Parameters
[37]		→	Litteral
[38]	ExpressionList	→	Expression , ExpressionList
[39]		→	Expression
[40]	VariableList	→	**varName**, VariableList
[41]		→	**varName**
[42]	Expression	→	**varName**
[43]		→	`!` **varName**
[44]		→	Litteral
[45]	Litteral	→	`true`
[46]		→	`false`

Figure 2.1: Grammar for the Concurrent boolean programs

```
vars :  g1, g2 ;
locks :  ;
messages :  msg ;
threads :  t1, t2, main ;

t1 { vars :  l1 ;
   if (g1) {
      skip ;
   } else {
      l1 := g1 ;
   }
   rendezvous(msg, l1) ;
}
```

```
t2 { vars :  l2 ;
   skip ;
   accept (msg, l2) ;
}

main { vars :  ;
   start(t1, false) ;
   start(t2, true) ;
}
```

Figure 2.2: A simple Concurrent Boolean Program.

After some tests and assignments, the instance of `t1` will send the *rendez-vous*, along with the value stored in `l1`. This value will be assigned to `l2` by the instance of `t2`. □

In order to simplify and clarify the subsequent discussion, we need to add, to the grammar, a set of more restrictive well-formedness rules. These rules are stated in the next sub-section. It should be easy for the reader to convince himself that these restrictions do not lower the expressive power of CBP.

2.1.1 Well-Formed Concurrent Boolean Programs

We begin this subsection with a set of useful definitions:

Let $\mathit{Locks}(S)$, $\mathit{Global}(S)$, $\mathit{Messages}(S)$ be respectively the set of locks, global variables, and messages declared in S. Let $\mathit{NameThreads}(S)$ be the set of names of thread types declared in the preamble of S, and let $\mathit{Threads}(S)$ be the set of threads whose code is specified in S. Furthermore, let $\mathit{Labels}(S)$ be the set of labels present in program S (that is, for every ℓ in $\mathit{Labels}(S)$, there exists a line in S beginning by $[\ell]$, and conversely). Finally, let $\mathit{Local}(T)$ be the set of *local variables* declared for the thread type T, and $\mathit{Body}(T)$ be the sequence of instructions ('LabeledCommands' in Figure 2.1) obtained by derivation of 'ThreadBody', for the thread type T.

A *well-formed* concurrent boolean program is a program S, generated by the grammar shown at Figure 2.1 which also satisfies the following conditions:

Labels In S, one cannot find two lines labelled with the same label ;

Goto For each instruction `goto`(ℓ) in S: ℓ must belong to $\mathit{Labels}(S)$;

Declarations If a variable v is used in the body of thread type T, v must be either in $\mathit{Global}(S)$, either in $\mathit{Local}(T)$. Each lock that is used in S (in a `lock`, an `unlock` or a `sleep` instruction) must belong to $\mathit{Locks}(S)$. Each thread in $\mathit{NameThreads}(S)$ must also be in $\mathit{Threads}(S)$ and conversely. Finally, each message used in S (in a `sleep`, a `wakeup(all)`, a `rendezvous` or an `accept`) must be part of $\mathit{Messages}(S)$.

Thread start For each `start`$(T, \pi_1, \ldots, \pi_n)$ instruction in S, two conditions must hold: 1) $T \in \mathit{Threads}(S)$ and 2) n – the number of parameters – and the number of local variables of thread type T must be equal ;

Main thread There must be a `main` thread declared in S: $\{\texttt{main}\} \in \mathit{NameThreads}(S)$;

Rendez-vous For each `rendezvous`$(M, e_1, \ldots, e_n)$ in the body of thread type T, every $e_i : 1 \leq i \leq n$ must be an expression over variables in $\mathit{Local}(T)$;

Accept For each `accept`$(M, u_1, \ldots, u_n)$ appearing in the body of thread type T, all the $u_i : 1 \leq i \leq n$ must be in $\mathit{Local}(T)$.

Assignment For each $u_1, \ldots, u_n{:=}v_1, \ldots, v_n$ in the body of any thread: $\forall 1 \leq i, j \leq n : i \neq j \Rightarrow u_i \neq u_j$.

Locks The `lock` and `unlock` operations must always be paired at the same `if` or `while` level. By this last constraint we mean that the `lock` and the `unlock` must be present in the same or `while` block. For instance, one cannot have something like `lock`(L) ;`if`(E) {`unlock`(L) ;}`else`{`skip` ;}. Moreover if a `goto` appears inside a `lock`/`unlock` block, then it must branch to a label *inside* the block. Symmetrically, no `goto` outside of a `lock`/`unlock` block ca branch to a label inside the block. Of course, the `lock` must happen *before* the corresponding `unlock`.

This last rule has been introduced to avoid situations in which a thread unlocks a lock it doesn't hold. In the particular case we are interested in (the translation of a JAVA program into a CBP), this restriction is not annoying, as JAVA owns a special `synchronized` statement that allows one to build a complete block protected by a single lock. Another solution to cope with this potential problem, would have been to assign a unique identity to each thread, and to check the identities at each `unlock` operation. We have chosen to reject this solution as this would have complicated our notations and definitions, with only a poor improvement in the precision of our reasoning.

Definition 2.1 (Well-Formed) Concurrent Boolean Program – First definition
A **(Well-Formed) Concurrent Boolean Program** is any program written in the programming language defined with the grammar of Figure 2.1, and following the well-formedness rules of 2.1.1. □

2.2 Semantics

We can now address the behavioural aspect of CBP. We achieve it in this section by presenting a set of *axioms* that allows to describe the *operational semantics* of CBP. These axioms are mainly inspired by the discussion of chapter 8 of [AO97].

We open the section with some useful definitions that state precisely what we mean by 'Concurrent Boolean Program'. We also define the concept of *state* of a CBP, which is the basis to a formal operational semantics. After that, we present the axioms. Finally, we give the definition of the *execution of a CBP*.

2.2.1 Definitions

Up to now, we have seen a CBP as a set of *type of threads*, plus some *global variables* and *locks* (recall the declaration section of 2.1). We can now state this more precisely:

Definition 2.2 Concurrent Boolean Program (CBP)
A **Concurrent Boolean Program** is a tuple $\langle \mathcal{G}, \mathcal{L}, \mathcal{T} \rangle$, where:

- $\mathcal{G}$ is a set of *global boolean variables* ;
- $\mathcal{L}$ is a set of *locks* ;
- $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ is a set of *types of thread.* We denote the singleton $\{T_j\}$ as T_j.

□

A *type of thread* contains all the necessary information to *instantiate* a new thread:

Definition 2.3 Type of Thread
A **Type of Thread** is a tuple $\langle \mathcal{V}, S \rangle$, where:

- $\mathcal{V}$ is a set of *local boolean variables* ;
- S is a sequence $I_1 \cdot I_2 \cdot \ldots \cdot I_n$ of instructions (we use $\cdot$ as a *sequence operator*). Each such instruction can be derived from the LabeledCommands rule of the grammar at Figure 2.1.

□

This definition of a CBP is completely equivalent to Definition 2.1. Indeed, given a CBP S_1 constructed thanks to the rules of 2.1, one can construct a CBP $S_2 = \langle \mathcal{G}, \mathcal{L}, \mathcal{T} \rangle$, such that:

1. For each variable in $Global(S_1)$, there exists one and only one variable in $\mathcal{G}$;
2. For each lock in $Locks(S_1)$, there exists one and only one lock in $\mathcal{L}$;
3. For each thread T in $Threads(S_1)$, there exists one and only one type of thread $\langle \mathcal{V}, S \rangle$ in $\mathcal{T}$, such that:
   (a) For each variable in $Local(T)$, there exists one and only one variable in $\mathcal{V}$;
   (b) S is exactly $Body(T)$.

Example 5 For instance, the type of thread for the thread `t2` of Figure 2.2 is

$$\langle \underbrace{\{\texttt{12}\}}_{\text{local variables}} , \underbrace{\texttt{skip}\ ;\cdot\ \texttt{accept(msg,12)}\ ;}_{\text{sequence of instructions}} \rangle$$

It might sound quite cumbersome to have two different definitions for the same concept. However, syntactical matters about CBPs are more easily introduced through the grammar of Figure 2.1, hence from Definition 2.1. On the other hand, the semantical aspect of the CBPs can very easily be introduced in a formal program-rewriting system. Therefore, we need to have a definition such as Definition 2.2 at our disposal.

With this new definition, we can now explain the behaviour of a CBP. In its initial state, a special thread (called `main`) is the only one to be active (such a thread always exists: see the well-formedness rules of 2.1.1). Each thread has the possibility to start several other threads of a given type, thanks to the `start` primitive. It should be clear that the evolution of a CBP being in a given 'state' depends only of the evolution of the different thread instances that are active in that state: we do not need more information in order to be able to predict the future evolution of the CBP.

Therefore, we define the notions of *global state* of a CBP, which depends of the local states of the thread instances, and of the valuations of the global variables and of the locks. The idea of 'local state' of the thread is captured through the *thread instantiation* definition, which gives the valuation of the local variables of the thread, as well as the point of the code that has been reached so far.

Definition 2.4 Global State of a Concurrent Boolean Program.
A **Global State of a CBP** $\langle \mathcal{G}, \mathcal{L}, \mathcal{T} \rangle$ is a tuple $\langle \Gamma, \Lambda, \Theta \rangle$, where:

- $\Gamma : \mathcal{G} \mapsto \{\top, \bot\}$ is a valuation of the global variables. One extends this valuation to Expressions over boolean variables (see rule 42 and following of Figure 2.1) as follows:
  - $\Gamma(\texttt{true}) = \top$
  - $\Gamma(\texttt{false}) = \bot$
  - $\Gamma(!u) = \bot$ if $\Gamma(u) = \top$
  - $\Gamma(!u) = \top$ if $\Gamma(u) = \bot$
- $\Lambda : \mathcal{L} \mapsto \{locked, unlocked\}$ is a valuation of the locks ;
- Θ is a multi-set of thread instantiations of types in $\mathcal{T}$.

□

Θ has been defined as a *multi-set* in order to be able to avoid problems when two threads are in the same local state. One can see this as a *counting abstraction*, in which we just retain the number of threads being in each local state, instead of the exact local state of every single thread. This means that we *abstract* the identities of the threads.

The counting abstraction will become more important in the sequel, because a CBP will be eventually translated into a Petri net, where each thread will be encoded into an 'anonymous' token. Moreover, this allows us to simplify the subsequent proofs.

Definition 2.5 Thread Instantiation.
A **thread instantiation** of $T_i = \langle \mathcal{V}_i, S_i \rangle$ $(1 \leq i \leq k)$ is a tuple $\langle P, \rho \rangle$, such that P denotes the instructions remaining to be executed ($P = \varepsilon$ if no instructions remain) and $\rho : \mathcal{V}_i \mapsto \{\top, \bot\}$ is a valuation of the local variables. ρ is extended to the Expressions over boolean variables similarly to Γ (see Definition 2.4). □

2.2.2 Axioms

The axioms given in this subsection allow one to compute all the legal transitions between two global states of a CBP. Two types of operations executed by threads can be distinguished :

1. **The local operations** These operations are executed by one thread and has no effect on the others. The associated rules are given in an inductive fashion. The base cases are those where we consider a global state with a single thread instantiation, containing a single instruction. These axioms (from 2.1 to 2.9) are of the form $G \rightarrow G'$, which indicates that the transition from G to G' is legal.

   The inductive case is obtained through the sequence axiom (2.10) and the interleaving axiom (2.11). They are of the form $\frac{G_1 \rightarrow G_1'}{G_2 \rightarrow G_2'}$, which means: 'If the transition $G_1 \rightarrow G_1'$ is allowed, then so is $G_2 \rightarrow G_2'$'.

   Among these axioms, one is of particular interest: the axiom 2.6 for the `goto`, because of the definition of function γ we have to introduce there (it will be re-used in the sequel). Roughly speaking, this function allows one to know, given a program S and a label ℓ, what are the instructions of S that remains to be executed from the operations labelled by ℓ. γ is necessary to handle correctly the cases where, e.g., we make a jump to a label that is into several nested `while` or `if` blocks. In these cases, it should be obvious that the instructions to execute after the one at label ℓ might not be these that immediately follow ℓ in S.

2. **The synchronisation operations** These operations are executed by a thread but modify the local state of other ones. Therefore, we present the corresponding rules by considering global states with more than one thread instantiation.

Along this section, we assume a mapping from the set of tokens **lockId**, **varName** and **threadTypeName** to their corresponding instances in the sets $\mathcal{L}$, $\mathcal{G} \cup_i \mathcal{V}_i$ and $\mathcal{T}$, respectively. We use M and L to denote a message and a lock respectively.

The axioms for `rendezvous`(**msgId**), `accept`(**msgId**) and `start`(**threadTypeName**) have been omitted. They can be deduced from the axioms 2.16 (for `accept` and `rendezvous`) and 2.15 (for `start`), with an empty set of parameters.

Assignment Rule

$$\begin{array}{c} \langle \Gamma, \Lambda, \langle v_1, \ldots, v_n := u_1, \ldots, u_n;\ , \rho \rangle \rangle \\ \rightarrow \\ \langle \Gamma[v_{i_1} = c_{i_1} \ldots v_{i_k} = c_{i_k}], \Lambda, \langle \varepsilon, \rho[v_{i_{k+1}} = c_{i_{k+1}} \ldots v_{i_n} = c_{i_n}] \rangle \rangle \end{array} \quad (2.1)$$

where the set of local variables of the thread is $\mathcal{V}$ and where $c_j = (\rho \cup \Gamma)(u_j)$ $(1 \leq j \leq n)$. We also assume a mapping of the variable index: $\{1, \ldots, n\} \rightarrow \{i_1, \ldots, i_n\}$, in order to *sort the variable by type.* Indeed, our mapping is such that: $\forall j :\ 1 \leq j \leq k :\ v_{i_j} \in \mathcal{G}$ and $\forall j : k < j \leq n : v_{i_j} \in \mathcal{V}$.

If Rule

$$\langle \Gamma, \Lambda, \langle \texttt{if } (B) \texttt{ then } \{S_1\} \texttt{ else } \{S_2\}, \rho \rangle \rangle \rightarrow \left\{ \begin{array}{ll} \langle \Gamma, \Lambda, \langle S_1, \rho \rangle \rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \models B \\ \langle \Gamma, \Lambda, \langle S_2, \rho \rangle \rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \not\models B \end{array} \right. \quad (2.2)$$

Choice rule For every j such that $1 \leq j \leq m$ and $\Gamma \cup \rho \models B_j$:

$$\begin{array}{c} \langle \Gamma, \Lambda, \langle \texttt{choice}\{B_1 : v_{1,1}, \ldots, v_{1,n_1} := u_{1,1}, \ldots, u_{1,n_1};\ \ldots\ B_m : v_{m,1}, \ldots, v_{m,n_m} := u_{m,1}, \ldots, u_{m,n_m};\}, \rho \rangle \rangle \\ \rightarrow \\ \langle \Gamma[v_{i_1} = c_{i_1}, \ldots, v_{i_k} = c_{i_k}], \Lambda, \langle \varepsilon, \rho[v_{i_{k+1}} = c_{ik+1}, \ldots, v_{i_{m_j}} = c_{i_{m_j}}] \rangle \rangle \end{array} \quad (2.3)$$

1. If S does not contain label ℓ, then $\gamma(S, \ell) = \varepsilon$.
2. If $S \equiv S_1 \cdot [\ell']$ $\texttt{while}(B)\{S_2\} \cdot S_3$, where S_1 does not contain ℓ, but S_2 does ; then $\gamma(S, \ell) = \gamma(S_2, \ell) \cdot \texttt{while}(B)\{S_2\} \cdot S_3$;
3. If $S \equiv S_1 \cdot [\ell']$ $\texttt{if}(B)$ $\{S_2\}\texttt{else}\{S_3\} \cdot S_4$, where S_1 does not contain ℓ, then:
   (a) Either S_2 contains ℓ, then: $\gamma(S, \ell) = \gamma(S_2, \ell) \cdot S_4$;
   (b) Either S_3 contains ℓ, then: $\gamma(S, \ell) = \gamma(S_3, \ell) \cdot S_4$.
4. If $S \equiv S_1 \cdot [\ell] I \cdot S_2$, where S_1 does not contain ℓ ; then: $\gamma(S_1, \ell) = [\ell] I \cdot S_2$;

Figure 2.3: The definition of $\gamma(S, \ell)$.

where $c_j = (\rho \cup \Gamma)(u_j)$ $(1 \leq j \leq n)$, assuming a re-ordering of the variables v_j and of the expressions u_j, such that the k first variables are global, and the remainder are local (as we did for the assignment).

While Rule

$$\langle \Gamma, \Lambda, \langle \texttt{while } (B)\{S\}, \rho \rangle \rangle \rightarrow \begin{cases} \langle \Gamma, \Lambda, \langle S \cdot \texttt{while } (B)\{S\}, \rho \rangle \rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \models B \\ \langle \Gamma, \Lambda, \langle \varepsilon, \rho \rangle \rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \not\models B \end{cases} \tag{2.4}$$

Skip Rule

$$\langle \Gamma, \Lambda, \langle \texttt{skip;}\ , \rho \rangle \rangle \rightarrow \langle \Gamma, \Lambda, \langle \varepsilon, \rho \rangle \rangle \tag{2.5}$$

Goto Rule

$$\langle \Gamma, \Lambda, \langle \texttt{goto } (\ell);\ , \rho \rangle \rangle \rightarrow \langle \Gamma, \Lambda, \langle \gamma(S, \ell), \rho \rangle \rangle \tag{2.6}$$

Where:

- The body of the thread is S ;
- $\gamma(S, \ell)$ is defined as shown in Figure 2.3. This function will be crucial in the sequel (see 3.2). It simply returns, given a program S, and a label ℓ (which, following the well-formedness rules, should be present at most once in S), the sequence of instructions remaining to be executed when the execution reaches ℓ. For instance, if a thread T executes S (S is the program in the thread's body), which contains ℓ, then the local state of this thread will be exactly $\langle \gamma(S, \ell), \rho \rangle$ when the flow of execution of T will reach the instruction labelled by ℓ.

Lock Rule The semantics of the `lock` instruction is simply to *take a lock*. This instruction, along with the corresponding `unlock`, has been introduced in order to model the `synchronized` blocks of JAVA.

$$\langle \Gamma, \Lambda, \langle \texttt{lock } (L);\ , \rho \rangle \rangle \rightarrow \langle \Gamma, \Lambda[L = \textit{locked}], \langle \varepsilon, \rho \rangle \rangle \tag{2.7}$$

iff $\Lambda(L) = \textit{unlocked}$

Unlock Rule

$$\langle \Gamma, \Lambda, \langle \texttt{unlock } (L);\ , \rho \rangle \rangle \rightarrow \langle \Gamma, \Lambda[L = \textit{unlocked}], \langle \varepsilon, \rho \rangle \rangle \tag{2.8}$$

iff $\Lambda(L) = \textit{locked}$

Sleep Rule The `sleep` instruction has been introduced to model the `wait` instruction of JAVA. The semantics of this latter construct is to let the thread release the last taken lock and jump to a kind of *sleep state*, waiting to be awoken (as sketched in the introduction: see 1.2). Therefore, the `sleep` instruction has two parameters: the first one corresponds to the message that will awake the thread ; and the second one, to the lock to be relinquished.

To model the *sleep state* of the thread, we have to introduce a special instruction *wait* (labelled by $\mathbf{wait}_\ell$. To symbolise that a thread moves to the sleep state when it calls `sleep`, we substitute it by *wait* in the program of the thread. Remark that, after being woken up, the thread immediately tries to re-acquire the lock it had just relinquished (see axioms 2.12 to 2.14).

$$\begin{array}{c}\langle \Gamma, \Lambda, \langle \texttt{sleep}\ (M, L), \rho\rangle\rangle \\ \rightarrow \\ \langle \Gamma, \Lambda[L = \mathit{unlocked}], \langle [\mathbf{wait}_\ell] \mathit{wait}(M, L), \rho\rangle\rangle\end{array} \tag{2.9}$$

iff $\Lambda(L) = \mathit{locked}$

Sequence Rule As explained before, this rule, as well as the next ones are for the induction.

$$\frac{\langle \Gamma, \Lambda, \langle S_1, \rho\rangle\rangle \rightarrow \langle \Gamma', \Lambda', \langle S_2, \rho'\rangle\rangle}{\langle \Gamma, \Lambda, \langle S_1 \cdot S, \rho\rangle\rangle \rightarrow \langle \Gamma', \Lambda', \langle S_2 \cdot S, \rho'\rangle\rangle} \tag{2.10}$$

Interleaving Rule

$$\frac{\langle \Gamma, \Lambda, \langle S_i, \rho_i\rangle\rangle \rightarrow \langle \Gamma', \Lambda', \langle T_i, \rho_i'\rangle\rangle}{\langle \Gamma, \Lambda, \Sigma\rangle \rightarrow \langle \Gamma', \Lambda', \Sigma'\rangle} \tag{2.11}$$

where Σ' is defined from Σ as follows. If

$$\Sigma = \{\langle S_1, \rho_1\rangle, \langle S_2, \rho_2\rangle, \ldots, \langle S_i, \rho_i\rangle, \ldots, \langle S_k, \rho_k\rangle\}$$

then

$$\Sigma' = \{\langle S_1, \rho_1\rangle, \langle S_2, \rho_2\rangle, \ldots, \langle T_i, \rho_i'\rangle, \ldots, \langle S_k, \rho_k\rangle\}$$

Wakeup Rules The `wakeup` statement has been introduced to model the `notify` statement of JAVA (about the semantics of this JAVA construct, see 1.2). As a `notify` can have no effect on the other threads if none of them is waiting to be awoken, two rules are necessary for the `wakeup` statement. The first one gives the semantics when there is at least one thread waiting to synchronise on a given message. The second is for the case where no such thread exists.

$$\begin{array}{c}\langle \Gamma, \Lambda, \{\langle \texttt{wakeup}\ (M) \cdot S_1, \rho_1\rangle, \langle [\mathbf{wait}_\ell] \mathit{wait}(M, L) \cdot S_2, \rho_2\rangle\} \cup \Sigma\rangle \\ \rightarrow \\ \langle \Gamma, \Lambda, \{\langle S_1, \rho_1\rangle, \langle [\mathbf{lock}_\ell] \texttt{lock}\ (L) \cdot S_2, \rho_2\rangle\} \cup \Sigma\rangle\end{array} \tag{2.12}$$

$$\langle \Gamma, \Lambda, \langle \texttt{wakeup}\ (M) \cdot S_1, \rho_1\rangle \cup \Sigma\rangle \rightarrow \langle \Gamma, \Lambda, \langle S_1, \rho_1\rangle \cup \Sigma\rangle \tag{2.13}$$

iff:

$$\forall j : 1 \leq j \leq k : j \neq i : \forall L : S_j \not\equiv [\mathbf{wait}_\ell] \mathit{wait}(M, L) \cdot S_j'$$

Wakeupall Rule As one could expect, the `wakeupall` statement has been introduced to model the effect of a `notifyall` in JAVA. As for the `wakeup`, a `wakeupall` could be of no effect if no threads are waiting to be awoken.

$$\langle \Gamma, \Lambda, \langle \texttt{wakeupall}\ (M) \cdot S, \rho\rangle \cup \Sigma_1 \cup \Sigma_2\rangle \rightarrow \langle \Gamma, \Lambda, \langle S, \rho\rangle \cup \Sigma_1' \cup \Sigma_2'\rangle \tag{2.14}$$

where

- for all $\sigma \in \Sigma_1$, we have $\sigma \equiv \langle [\mathbf{wait}_\ell] \mathit{wait}(M, L) \cdot S, \rho\rangle$;

- for all $\sigma \in \Sigma_2$, we have $\sigma \not\equiv \langle[\mathbf{wait}_\ell]wait(M,L)\cdot S,\rho\rangle$;
- $\Sigma_1' = \{\langle[\mathbf{lock}_\ell]\texttt{lock}\ (L)\cdot S,\rho\rangle |$ there exists $\langle[\mathbf{wait}_\ell]wait(M,L)\cdot S,\rho\rangle$ in $\Sigma_1\}$;
- $\Sigma_2' = \Sigma_2$.

Start Rule This statement is useful to model the call to the `run` method of a `Thread` object. Its parameters are: the name of the type of thread whom we want a new instance to be started, and a set of values, which should be assigned to the local variables of the newly created instance.

$$\langle\Gamma,\Lambda,\langle\texttt{start}\ (T_i,\pi_1,\ldots,\pi_{|\mathcal{V}_i|});\cdot S',\rho\rangle\cup\Sigma\rangle \to \langle\Gamma,\Lambda,\{\langle S',\rho\rangle,\langle S_i,\rho'\rangle\}\cup\Sigma\rangle \tag{2.15}$$

where $T_i = \langle\mathcal{V}_i, S_i\rangle \in \mathcal{T}$ and ρ' is such that $\forall i : 1 \le i \le |\mathcal{V}_i| : \rho'(v_i) = \pi_i$.

Rendez-vous **Rule** This is the classical blocking synchronisation primitive. A `rendezvous` always have to be paired with a corresponding (w.r.t. the message) `accept`. Our version of the *rendez-vous* also allows to pass values from the 'sending' thread (calling `rendezvous`) to the 'accepting' thread (doing the `accept`). We have chosen to restrict ourselves to *local values*[2], as a global variable can always be consulted by any thread.

There is no *rendez-vous*-like primitive in JAVA, but we have chosen to retain it in order to model other concurrent languages such as ADA [Bar96]. In ADA, one can define *entry points* in the tasks (the ADA equivalent to threads). These entry points can have parameters, through which one can pass values to the task implementing them (an entry point in Ada is thus similar to an `accept` in CBP). The '*rendez-vous*' in ADA is implemented as a function call, with the name of the entry points being the name of the function.

A *rendez-vous* can however easily be implemented in JAVA, as shown in Figure 2.4.

$$\begin{array}{c}\langle\Gamma,\Lambda,\langle\texttt{accept}\ (M,v_1,\ldots,v_n)\cdot S,\rho\rangle,\langle\texttt{rendezvous}\ (M,u_1,\ldots,u_n)\cdot S',\rho'\rangle\cup\Sigma\rangle\\ \to\\ \langle\Gamma,\Lambda,\{\langle S,\rho[v_1=\rho'(u_1),\ldots,v_n=\rho'(u_n)]\rangle,\langle S',\rho'\rangle\}\cup\Sigma\rangle\end{array} \tag{2.16}$$

2.2.3 Execution of a Concurrent Boolean Program

The last step to fully achieve the semantics of CBP is to state what is an *execution* of a CBP :

Definition 2.6 Execution of a Concurrent Boolean Program
Given a CBP B, an **Execution** of B is a sequence $e = G_1, G_2, \ldots, G_n$ of global states such that: $\forall i : 1 \le i \le n-1 : G_i \to G_{i+1}$ (according to the previous axioms). G_1 is said to be the *initial state* of e. G_n is the *target state* of e. □

From this definition, we can infer the notion of *reachable state*:

Definition 2.7 Reachable State
Given a CBP B, a global state G_t of B is said to be **reachable** from global state G_i of B (written $G_i \Rightarrow G_t$) iff there exists an execution e of B such that G_i is the initial state of e, and G_t is the target state of e. □

2.3 Discussion

In this chapter, we have presented the syntax and the semantics of the language of CBPs, which allows to describe concurrent programs. Among these programs, we are most of all interested in the encoding and the verification of multi-threaded JAVA programs. The next chapter shows how to encode a given JAVA program into a CBP, and how to translate a given CBP into a Global/Local Machine [DRVB02b], following our verification framework (see Figure 1.3).

[2] Values stored in a local variable, or literals.

```
public class rendezvous
{
   private Object content ;
   private boolean accepting = false ;

   public  rendezvous()
   {
      content = null ;
   }

   public synchronized Object accept()
   {
      Object temp ;
      accepting = true ;
      while(content == null)
      {
         try
         {
            wait() ;
         }
         catch(InterruptedException e)
         {
            System.out.println("Interrupted !");
         }

      }
      notify() ; /* This wakes up the sending thread if necessary */
      /* Before leaving, we have to reset the rendez-vous */
      accepting = false ;
      temp = content ;
      content = null ;
      return temp ;
   }

   public synchronized void send(Object O)
   {
      content = O ;
      notify() ; /* Wakes up the thread doing an accept if it exists */
      if(accepting == false)
      {
         try
         {
            wait() ;
         }
         catch(InterruptedException e)
         {
            System.out.println("Interrupted !");
         }

      }
   }
}
```

Figure 2.4: The `rendezvous` class that implements a *rendez-vous* in JAVA. The possibility to pass a value from the *sender* to the *receiver* is also implemented. Once the *rendezvous* has been initialised by the constructor, the sender can call method `send(Object O)`. Symmetrically, the receiver uses `Object accept()`. Both methods are will block the thread until the other party has joined the *rendezvous*. The `accepting` variable, initially false, is set to true when the `accept` function is called. It remains unchanged as long as no object has been put into the `content` field of the class. `accepting` serves as a flag to let the `send` function call `notify()` once it has changed the value of `content`.

Chapter 3

From JAVA to CBP and from CBP to the Global Machines

In this chapter, we deal with the two first steps of the verification framework we have presented in the introduction (see Figure 1.3). That is, we present theoretical results allowing one to automatically compute a boolean program from a given JAVA program, thanks to the technique of *predicate abstraction.* We then explain how to translate a CBP into a GM, and prove our translation rules to be correct. Finally, we briefly present `CPB2GM`, a tool that implements this latter translation.

3.1 From Java Programs to CBPs

Under some assumptions, multi-threaded JAVA programs can easily be encoded into CBPs. In the following, we assume that:

1. the references to global objects can be detected using some static analysis techniques,
2. the threads have a finite control part, i.e. we avoid infinite recursion, and
3. nested locks can be detected and removed.[1]

In [DRVB02b], the authors have shown how to translate the synchronisation primitives of JAVA thanks to communication mechanisms like the broadcast, the *rendez-vous* and the asynchronous *rendez-vous.* These constructs can be found among the keywords of the CBP language.

More precisely: the `notifyAll` primitive of JAVA is encoded thanks the `wakeupall` operation, the `notify` primitive is encoded with the `wakeup` operation and `sleep` is used for the `wait` primitive. As expected, `start` is used to instantiate threads.

The *management of locks* in a JAVA program requests to introduce, in the CBP, one lock for each global object in the JAVAprogram. The `lock` and `unlock` operations are used to encode the `synchronized` methods (blocks) of JAVA.

The valuations of the variables manipulated by the threads in the JAVA program are encoded using the *predicate abstraction* technique. This technique, which can be made fully automatic [BHPV00, BMMR01] even when predicates relate to variable references [BMR02], is used to construct finite abstractions of infinite state systems. It was first proposed by Graf and Saidi in [GH97] and recently applied in [BMMR01, BPR02, HJMS02] to prove safety properties. Our approach differs from the previous works in the fact that the resulting (abstract) program still has a (potentially) *infinite state space* as the number of threads may be unbounded.

Predicate abstraction is mainly based on the fact that only the information that allows to decide the tests in the `if` and `while` statement is relevant to define the set of executions of a program. Therefore,

[1] In JAVA a thread can re-acquire many times a lock that it already owns. This is not possible in a CBP, as the precondition of `lock` is that the lock is free. This can easily be detected as we inline the functions calls and disallow infinite recursion.

```
public class Point{
    private int x = 0 ;
    private int y = 0 ;

    public synchronized void incx(){
        x = x + 1 ;
        notifyAll() ;
    }

    public synchronized void decx() {
        while (x == 0)
            wait() ;
        x = x - 1 ;
    }

    public synchronized void incy(){
        y = y + 1 ;
        notifyAll() ; }
    public synchronized void decy() {
        while (y == 0)
            wait() ;
        y = y - 1 ; }
}
```

Figure 3.1: A simple example of JAVA class

```
public class Inc extends Thread {
    private Point p ;
    public Inc(Point p) {
        this.p = p ;
    }
    private void incpoint() {
        p.incx() ;
        p.incy() ;
    }
    public void run() {
        while (true)
            incpoint() ;
    }
}
```

```
public class Dec extends Thread {
    private Point p ;
    public Dec(Point p) {
        this.p = p ;
    }
    private void decpoint() {
        p.decx() ;
        p.decy() ;
    }
    public void run() {
        while (true)
            decpoint() ;
    }
}
```

Figure 3.2: JAVA thread classes manipulating a shared `Point` object

this technique manages to *map* the infinite domain of the JAVA variables into a finite domain of well-chosen predicates, that retain enough information to decide the tests.

A boolean variable b_i is thus defined for each condition e_i in the program, to encode the truth value of the expression describing the condition. JAVA programs are then translated into boolean programs having the same control part. These boolean programs are *abstract programs* that manipulate the b_i variables and whose operations reflect the modification of the truth value of the expressions e_i when applying the corresponding concrete program operations.

In the case of the CBPs, we will use *global variables* to retain the truth values of the predicates ranging on *fields of global objects*. The *Local variables* will be similarly used for the predicates referring to *data's that are local to the threads* (and potentially global data's).

Example 6 As an example, let us look at the JAVA thread classes of Figure 3.2 and let us consider a program in which several such threads execute concurrently. These threads manipulate a shared `point` object described in Figure 3.1. The thread of type `Dec` continuously try to decrement the coordinates of the point (thanks to the `decx` and `decy` functions), while the threads of type `Inc` use `incx` and `incy` to increment them. The coordinates may never drop under zero, so, when a `Dec` thread tries to decrement a null coordinate, it starts waiting (by calling the `wait` primitive). Each thread that increments the coordinates executes the `notifyAll` method to wake up all the potentially waiting threads. These awoken threads then compete for the lock of the `Point` object, which might send them back to sleep if the coordinates are still null.

To translate these JAVA threads into threads of CBP, we first inline the methods. Then, the predicate abstraction technique is applied on the inlined program. In our example, there are two loops guarded by the constraints `x == 0` and `y == 0`. Two boolean variables `x0` and `y0` are thus defined to describe the truth value of the two constraints respectively. More precisely, `y0` = *true* specifies that the expression `x == 0` is true.

The CBP program is constructed by replacing the tests and operations on `x` and `y` by tests and operations on x_0 and y_0 as follows:

- The conditions of the `while` statements in the `decx` and `decy` methods are replaced by $x0$ and $y0$, respectively ;
- The operation `x = x + 1` is replaced by

  choice { `x0 : x0 := false ; !x0 : x0 := false ; !x0 : x0 := true ;` }

- The increments on `y0` are translated in the same way ;
- The operation `x = x - 1` is also translated into:

  choice { `x0 : x0 := false ; !x0 : x0 := false ; !x0 : x0 := true ;` }

- The operation `y = y - 1` is replaced in the same way.

As one can see, some non-determinism has been introduced in the resulting CBP. Non-determinism is a consequence of the loss of information during the translation. As the exact values of the concrete variables are lost, it is not always possible to define a unique configuration resulting in the application of some transformations. However, it is reported in [BMMR01] that the loss is often negligible in practice to verify programs (and systems in a more general way).

To finish our translation we add a lock `lockpoint` and a message `msgpoint` that will be used to translate synchronized methods as well as the `wait/notifyAll` mechanism. Figure 3.3 shows the two CBP threads corresponding to the JAVA threads of Figure 3.2. □

A challenging problem of predicate abstraction is to find a good compromise between the information kept in the abstract model, i.e. the number of predicates used for its construction, and the size of the abstract model. The technique investigated in [BPR02, HJMS02] to deal with this problem consists in starting with a poor model constructed using few predicates (we do not start with the set of predicates corresponding to tests in JAVA programs) and then refining this model by analysing false error traces.

Analysing coarse abstract models often leads to a false negative answer of the model checker. This kind of answer comes from the detection of an abstract execution that does not correspond to an execution of the (concrete) program. Typically, this comes from non-determinism introduced by predicate abstraction and the fact that some choices in executions of the abstract models have no concrete counterparts.

One can then analyse the false error traces, and the bad choices are (automatically) detected leading to some new predicate that avoid these choices: these new predicates remove the detected bad error trace (and potentially other ones) from the abstract model. The refined model is then analysed again.

The refinement process ends in two cases:

1. Either the abstract model is proved to be safe. The safety property of the original program is then *directly deduced* (thanks to Lemma 3.2 presented in the sequel).
2. Or an error trace of the abstract model is detected and allows to construct a corresponding unsafe execution of the original program.

According to the theoretical results (following Rajamani in [BCR01a], undecidability of safety property checking for CBPs can be proved along the lines of [Ram99]), this process is not guaranteed to stop.

Example 7 The translation in CBP of the Bakery Algorithm we had introduced in Chapter 1 is given in Appendix A (A.2). As the identities of the threads (encoded by their tickets' values) are relevant in this case, we need to *specialise* our model in order to reflect this behaviour. Therefore, we have translated the

```
inc {vars :  ;
   while(true) {
      lock(lockpoint) ;
      choice {
         x0 :  x0 := false ;
         !x0 :  x0 := false ;
         !x0 :  x0 := true ;
      }
      wakeupall(msgpoint) ;
      unlock(lockpoint) ;
      lock(lockpoint) ;
      choice {
         y0 :  y0 := false ;
         !y0 :  y0 := false ;
         !y0 :  y0 := true ;
      }
      wakeupall(msgpoint) ;
      unlock(lockpoint) ;
   }
}
```

```
dec {vars :  ;
   while(true) {
      lock(lockpoint) ;
      while(x0) {
         sleep(msgpoint, lockpoint) ;
      }
       choice {
         x0 :  x0 := false ;
         !x0 :  x0 := false ;
         !x0 :  x0 := true ;
      }
      unlock(lockpoint) ;
      lock(lockpoint) ;
      while(y0) {
         sleep(msgpoint, lockpoint) ;
      }
      choice {
         y0 :  y0 := false ;
         !y0 :  y0 := false ;
         !y0 :  y0 := true ;
      }
      unlock(lockpoint) ;
   }
}
```

Figure 3.3: The `inc` and `dec` CBP threads corresponding to the `Inc` and `Dec` JAVA threads of Figure 3.2

`myThread` JAVA thread into three CBP thread. The `Env` thread has been obtained by extracting the skeleton of `myThread` (it is thus fairly coarse). We allow an unbounded amount of such threads to be instantiated (see the **while**{*} loop in the `main` thread). The threads `Ti` and `Tj` are specialised encodings of the `myThread` JAVAthread, owning distinct identities (we assume, without loss of generality, that when `Ti` and `Tj` have the same ticket, `Ti` is elected to enter the critical section). These threads manipulate several boolean variables that encode the orderings of the respective tickets as follows:

Variable name	Intended meaning
`Ieq0`	`Ti`'s ticket equals 0
`Jeq0`	`Tj`'s ticket equals 0
`IeqJ`	`Ti`'s ticket equals `Tj`'s ticket
`IgJ`	`Ti`'s ticket is greater than `Tj`'s ticket
`JgI`	`Tj`'s ticket is greater than `Tj`'s ticket
`IChoose`	`Ti` is choosing a ticket value
`JChoose`	`Tj` is choosing a ticket value

`BakeryLock` and `BakeryMsg` are used to model the synchronisation of the Bakery. `ILock`, `JLock`, `IMsg`, `Jmsg` are used similarly for `Ti`'s and `Tj`'s respective tickets.

The property we have to verify on this model is the mutual exclusion between `Ti` and `Tj`. If we can prove it for these two threads, it remains true for any other thread, by symmetry of the concrete system. □

Now that we have shown how to translate a JAVA program into a CBP[2], let's show (according to our *verification framework* of Figure 1.3) how to translate a CBP into a Global/Local Machine.

[2]This might sound quite optimistic, as we haven't tried to analyse the quality of this first step of translation. As stated in the introduction, many optimisations could and should be applied in order to reduce the size of the CBP. The reader should refer to Chapter 4 for a survey of such optimisation techniques.

3.2 From CBP to Global/Local Machines

Let us now introduce a new step toward the verification of properties on CBP(in Figure 1.3, it is the second step). We will show in this section how to translate a CBP into a *family of Global/Local Machines.* But before doing so, we need to introduce some preliminary concepts. First of all, we need to define the concept of Global Machine (this is done in 3.2.1). Afterwards, we state an important lemma that shows that, for each line of a piece of code S, there is one and only one set of instructions remaining to be executed. This result allows to establish a one-to-one correspondence between the pairs

(program remaining to be executed, valuation of the local variables)

which define the local states of the thread instances ; and the pairs

(program label, valuation of the local variables)

which allows one to encode the local states of the local machines. But to be complete, we need the assumption that each program line has a label, and that there are not two distinct program lines that share the same label. We enforce this thanks to the *Relabel* function whose definition opens 3.2.2.

Then, we eventually explain the translation.

3.2.1 The Global/Local Machines (from [DRVB02b])

Conceptually, a Global Machine (GM) is a set of Local Machines plus a set of global boolean variables. The Local Machines can be seen as communicating automata, which can manipulate the global boolean variables (with communication primitives like the broadcast or the *rendez-vous*).

We start the description of Global Machines from the operations needed to handle these variables.

Definition 3.1 Boolean Formula's

Let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a finite set of *global boolean variables*, and let $\mathcal{B}'$ be their primed version. A *boolean guard* φ_g is either the formula *true* or the conjunction of literals $L_1 \wedge \ldots \wedge L_p$, $p \leq n$, such that L_i is either b or $\neg b$ for some $b \in \mathcal{B}$. A *boolean action* φ_a is a formula $b'_1 = v_1 \wedge \ldots \wedge b'_n = v_n$, where $v_i \in \{true, false, b_i\}$ for $i : 1, \ldots, n$. □

Boolean guards and actions are used to express pre-and post-conditions on the variables in $\mathcal{B}$. We now introduce the notion of Local Machine.

Definition 3.2 Local Machine

A *local machine* is a tuple $\langle Q, \Sigma, \delta \rangle$, where: Q is a finite set of states; Σ is the set of synchronisation labels used to build the set of possible actions $\mathcal{A}$ of a process (defined later); and $\delta \subseteq (Q \times \mathcal{A} \times Q)$ is the local transition relation. In the following, we will use $s \xrightarrow{\alpha} s'$ to indicate that $\langle s, \alpha, s' \rangle \in \delta$. □

The *actions* of a Local Machine are defined as follows (in the following φ represent the conjunction of a boolean guard with an action (Def. 3.1), and $\ell \in \Sigma$):

- *Internal action:* $\ell : \varphi$;
- *Rendez-vous:* $\ell! : \varphi$ (sending), and $\ell?$ (reception);
- *Asynchronous Rendez-vous:* $\ell\uparrow$: φ (sending), and $\ell\downarrow$ (reception);
- *Broadcast.* $\ell!! : \varphi$ (sending), and $\ell??$ (reception).

Having in mind the translation from Java programs, we will also apply the following restrictions:

1. The set of source and target states of a broadcast (asynchronous rendez-vous) reception must be distinct ;

2. Broadcasts receptions associated to the same sending can be partitioned so that each partition is defined over a distinct set of states.

Note, in fact, that we will use asynchronous rendez-vous and broadcast to model the semantics of `wakeup` and `wakeupall`. Our restriction avoids *cyclic rules* like $sloc_1 \xrightarrow{n!!} sloc_2$, $rloc_1 \xrightarrow{n??} rloc_2$, and $rloc_2 \xrightarrow{n??} rloc_1$ that have no meaning if *sloc* and *rloc* are control points in the code of the sender and of the receiver, respectively, and n corresponds to a `notifyAll`.

Having all the necessary definitions at our disposal, we can define the notion of *Global Machine*:

Definition 3.3 Global Machine
A *global machine* is a tuple $\mathcal{G} = \langle \mathcal{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$, where: $\mathcal{B} = \{b_1, \ldots, b_n\}$ is the set of *global boolean variables* for $i : 1, \ldots, m$; $\mathcal{L}_i = \langle Q_i, \Sigma_i, \delta_i \rangle$ is the i-th local machine; and k_i the number of its copies. Furthermore, we have that $Q_i \cap Q_j = \emptyset$ for any $i, j : 1, \ldots, m$ with $i \neq j$. □

3.2.2 Reachable Programs

Let us first define *Relabel*(S) that transforms a program S by adding a new label (that is, a label that was not previously in *Labels*(S)) to each of its line that was not labelled. Furthermore, *Relabel*(S) also adds the instruction $[\mathbf{end}]\varepsilon$ at the end of S.

We also need to define $\widetilde{\gamma}$, an extension of γ (see Figure 2.3), in order to handle the special labels $[\mathbf{wait}_\ell]$ and $[\mathbf{lock}_\ell]$, that we have introduced along the axioms 2.9 and 2.7 of 2.2.2. The definition should be self-explanatory.

Definition 3.4 $\widetilde{\gamma}$: The Extension of γ.

$$\widetilde{\gamma}(S,\ell) = \begin{cases} \gamma(S,\ell) & \forall \ell \in \mathit{Label}(\mathit{Relabel}(S)) \\ [\mathbf{wait}_{\ell'}]\mathit{wait}(\mathbf{msgId}, \mathbf{lockId}) \cdot S' & \text{if } \ell = \mathbf{wait}_{\ell'} \\ & \text{and } \gamma(S,\ell') = [\ell']\mathtt{sleep}(\mathbf{msgID}, \mathbf{lockId})\ ; \cdot S' \\ [\mathbf{lock}_{\ell'}]\mathtt{lock}(\mathbf{lockId})\ ; \cdot S' & \text{if } \ell = \mathbf{lock}_{\ell'} \\ & \text{and } \gamma(S,\ell') = [\ell']\mathtt{sleep}(\mathbf{msgID}, \mathbf{lockId})\ ; \cdot S' \end{cases}$$

□

$\widetilde{\gamma}$ is useful to us because, given a label ℓ and a program S, it should return exactly the part of S remaining to be executed when the label ℓ is reached. This implies that there is *one and only one* such program corresponding to *each* program label. It should be clear that this results is important to define a procedure to translate a CBP into a GM. We first have to prove that $\widetilde{\gamma}(S,\ell)$ is the only program reachable in S at label ℓ:

Lemma 3.1 *Let $B = \langle \mathcal{G}, \mathcal{L}, \{T_1, \ldots, T_i = \langle \mathcal{V}, S_{T_i} \rangle, \ldots, T_n\} \rangle$ be a well-formed Concurrent Boolean Program. If ℓ is a label of T_i, $S_j = [\ell]I \cdot S'$ is a program and $Init_B$ is the initial state of B, then the following holds.*

$$Init_B \Rightarrow \langle \Gamma, \Lambda, \ldots \langle S_j, \rho_j \rangle \ldots \rangle \text{ if and only if } \widetilde{\gamma}(S_{T_i}, \ell) = S_j$$

Proof If we have reached the local state $\langle S_j, \rho_j \rangle$ from $\langle [\ell']\mathtt{goto}(\ell); S', \rho_j \rangle$, then, it immediately follows from the `goto` axiom (2.6) that $S_j = \widetilde{\gamma}(S_{T_i}, \ell)$. In the other case, we prove the lemma by induction on the number k of `if` and `while` blocks containing the label ℓ:
case $k = 0$ If ℓ is not contained into a `if` or a `while` block, then $S_{T_i} = S_1 \cdot [\ell]I \cdot S_2$ (this holds because ℓ is a label of S_{T_i}). Two cases hold. Either the sequence of instructions to reach ℓ contains at least one `goto` operation or not. In the first case, suppose the last `goto` performed to reach ℓ is labelled by ℓ'. S_1 can be decomposed in $S_{11} \cdot [\ell']\mathtt{goto}(\ell'') \cdot S_{12}$. We obtain $S_{T_i} \Rightarrow [\ell'']I \cdot S_3 \cdot [\ell]I \cdot S_2$. Following the sequence axiom (2.10) we obtain $[\ell'']I \cdot S_3 \cdot [\ell]I \cdot S_2 \rightarrow_{2.10} [\ell]I \cdot S_2$.

In the second case, using the sequence axiom (2.10), we know that we can reach $S_j = [\ell]I.S_2$ from S_{T_i}. In the two cases, S_j is unique and corresponds to $\widetilde{\gamma}(S_{T_i}, \ell)$.
case $k = n$ Suppose that: if ℓ is contained in at most $n-1$ blocks of `while` and `if`, then $\widetilde{\gamma}(S_{T_i}, \ell) = S_j$ (this is implied by the induction hypothesis). Let's prove that it is also the case if ℓ is contained into n blocks of `while` and `if`:

If the label ℓ is reached by firing at least one `goto` operation to an instruction in the same n blocks that ℓ, the result is obtained following the same reasoning used in the basic case (using the `goto` axiom (2.6) and the sequence axiom (2.10)).

In the other case, we need to consider $S = S_b \cdot [\ell]I \cdot S_a$, the sequence of instructions in which ℓ appears (Figure 3.4 should help the reader to get the picture...). S is such that either:

(1) $[\ell']\texttt{while}(E)\{S\}$ is the $n-th$ block containing ℓ, or

(2) $[\ell']\texttt{if}(E_1)\{S_1\}\ldots\texttt{elseif}(E_k)\{S_k\}$ is the $n{-}th$ block containing ℓ (in this case, S is one of the $S_1 \ldots S_k$).

To execute the instruction labelled by ℓ, it is first necessary to execute the instruction labelled by ℓ'. By induction hypothesis, we know that there is only one program that is reachable at label ℓ': it is $\widetilde{\gamma}(S_{T_i}, \ell')$. For the sake of clarity we will denote it by $S_{\ell'}$ in the sequel. Let's suppose $S_{\ell'} = [\ell']I \cdot S''$ (such that $S_{T_i} \Rightarrow S' = [\ell']I \cdot S''$).

If we are in case (1), then $S_{\ell'} = [\ell']\texttt{while}(E)\{S\} \cdot S''$. Let's apply the `while` axiom (2.4):

$$S_{\ell'} \rightarrow_{2.4} S \cdot [\ell']\texttt{while}(E)\ \{S\}) \cdot S''$$

and then by the sequence axiom (2.10):

$$S \cdot [\ell']\texttt{while}(E)\ \{S\}) \cdot S'' \rightarrow_{2.10} [\ell]I \cdot S_a \cdot \texttt{while}(E)\ \{S\}) \cdot S'' = S_\ell$$

It follows that, starting from S_{T_i} (which is, in $Init_B$, the program remaining to execute for the thread we are considering) we can construct only one program (S_ℓ) starting by an instruction labelled by ℓ.

Let us prove now that $S_\ell = \widetilde{\gamma}(S, \ell)$. As ℓ and ℓ' are in the same $n-1$ first blocks of `if` and `while`, we have (by definition of $\widetilde{\gamma}$):

$$\begin{aligned}
\widetilde{\gamma}(S_{T_i}, \ell) &= \widetilde{\gamma}([\ell']\texttt{while}(E)\ \{S\}), \ell) \cdot S'' \\
&= \widetilde{\gamma}(S, \ell) \cdot [\ell']\texttt{while}(E)\ \{S\}) \cdot S'' \\
&= [\ell]I \cdot S_a \cdot [\ell']\texttt{while}(E)\ \{S\}) \cdot S'' \\
&= S_\ell
\end{aligned}$$

If we are in case (2), then $S_{\ell'} = [\ell']\texttt{if}(E_1)\ \{S_1\}\texttt{else}\{S_2\} \cdot S''$ with $S_i = S$ ($i \in \{1, 2\}$). Thanks to the `if` axiom (2.2) we obtain

$$S_{\ell'} \rightarrow_{2.2} S_i \cdot S''$$

and by the sequence axiom (2.10):

$$S_i \cdot S' \rightarrow_{2.10} [\ell]I \cdot S_a \cdot S'' = S_\ell$$

It follows that starting from S_{T_i} we can reach only one program (S_ℓ) starting by an instruction labelled by ℓ.

Let us prove that this program S_ℓ is the program $\widetilde{\gamma}(S, \ell)$. As ℓ and ℓ' are in the same $n-1$ first blocks of `if` and `while`, we have:

$$\begin{aligned}
\widetilde{\gamma}(S_{T_i}, \ell) &= \widetilde{\gamma}([\ell']\texttt{if}(E_1)\ \{S_1\}\texttt{else}\{S_2\}, \ell) \cdot S'' \\
&= \widetilde{\gamma}(S_i, \ell) \cdot S'' \\
&= [\ell]I \cdot S_a \cdot S'' \\
&= S_\ell
\end{aligned}$$

■

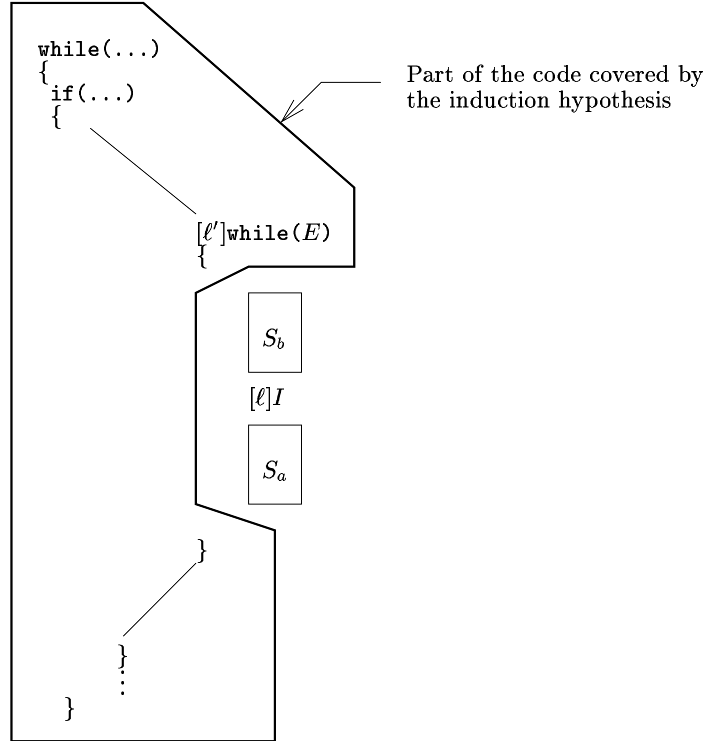


Figure 3.4: Graphical representation of the inductive proof of Lemma 3.1.

3.2.3 Translation

We are now assured that exactly one tuple (program label, valuation of the local variable) corresponds to each local state of a thread (and vice-versa), and this will help us define precisely the translation between the CBP and GM.

This translation is not really difficult to imagine, though there are some tricky issues. One of them is a consequence of the *dynamic creation* of threads. Such a behaviour is possible in a CBP, but not in with a GM, because a GMis a collection of a *fixed* number of local machine. Therefore, we have devised a special initial state for the Local Machines, which we denote by $\langle \theta_\top, \mathbf{begin} \rangle$. This state does not correspond to any local state of a thread, and thus, any local machine in this state represents a 'not-yet-created' thread. The `start` primitive will be used to make one of these local machines *move* to the state corresponding to the initial state of the thread.

This move will represent the call to the constructor of the corresponding thread, which means that we simultaneously need to set the initial values of the local variables. We can achieve this thanks to a *rendez-vous*, whose message encodes the type of thread as well as the values to assign to the local variables.

Another problem arises from the dynamic creation of threads: in the general case, we can't bound the number of instances of a given thread that will be created. Hence, we don't know how many local machine we should put in the corresponding global machine. We will cope with this problem later. Let us first explain how to translate each thread type of a CBP into a single Local Machine:

Translation of a CBP into a set of Local Machines Let $\langle \mathcal{G}, \mathcal{L}, \{\langle \mathcal{V}_1, S_1 \rangle, \ldots, \langle \mathcal{V}_n, S_n \rangle\} \rangle$ be a CBP. We translate it into a *set of Local Machines* $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ such that there is a bijection between this set and the set of types of threads. In the sequel, we will assume that the local machine $\mathcal{L}_i$ always corresponds to the type of thread T_i.

$\mathcal{L}_i = \langle Q_i, \Sigma_i, \delta_i, I_i \rangle$ is constructed from thread type $T_i = \langle \mathcal{V}_i, S_i \rangle$ as follows[3]:

[3]Although the syntax of the Local Machines requests them, we have systematically omitted the self-assignments ($g' = g$: frame axiom) in the labels of the transitions in order to clarify the subsequent discussion

1. $Q_i = \{\langle\theta, \ell\rangle\}$ such that:
   - θ is a valuation of the variables in $\mathcal{V}_i$;
   - $\ell \in$ *Label*(*Relabel*(S_i)) $\cup \{\mathbf{lock}_{\ell'}, \mathbf{wait}_{\ell'} | \exists \ell' \in$ *Label*(*Relabel*(S_i)) : γ(*Relabel*(S_i), ℓ') $\equiv [\ell']$`sleep`...$\}$ $\cup \{\mathbf{begin}, \mathbf{end}\}$

2. δ_i and Σ_i can be constructed as follows: $\forall \ell \in$ *Label*(*Relabel*(S_i)) $\cup \{\mathbf{begin}, \mathbf{end}\}$: we distinguish several cases depending on the value of $\gamma(\ell,$ *Relabel*(S_i)). If it is equal to:

   (a) $[\ell]u_1, \ldots, u_n{:=}v_1, \ldots v_n \cdot [\ell']I' \cdot S'$, then let us divide the set of couples $\langle u_i, v_i\rangle$ according to the type of u_i and v_i. Let $U_1 = \{\langle u_i, v_i\rangle | u_i, v_i \in \mathcal{G}\}$, $U_2 = \{\langle u_i, v_i\rangle | u_i \in \mathcal{G}$ and $v_i \in \mathcal{V}_i\}$, $U_3 = \{\langle u_i, v_i\rangle | u_i \in \mathcal{V}_i$ and $v_i \in \mathcal{G}\}$ and $U_4 = \{\langle u_i, v_i\rangle | u_i \in \mathcal{V}_i$ and $v_i \in \mathcal{V}_i\}$.
   Then, the following relation holds for every tuple $a = (a_1 \ldots a_{|U_1|+|U_3|})$ such that, for every tuple $\langle u_i, v_i\rangle \in U_1 \cup U_3$, there is an a_j of the form v_i or $\neg v_i$:

   $$\langle\theta, \ell\rangle \xrightarrow{\tau\ :\ a_1 \wedge \ldots \wedge a_{|U_1|+|U_3|} \wedge g_1' = b_1 \wedge \cdots \wedge g'_{|\mathcal{G}|+|\mathcal{L}|} = b_{|\mathcal{G}|+|\mathcal{L}|}} \langle\theta', \ell'\rangle$$

   where $\theta' = \theta[l_1 = c_1, \ldots, l_{|U_3|+|U4|} = c_{|U_3|+|U4|}]$
   and:
   - for each $\langle u_i, v_i\rangle \in U_1$, there exists $1 \leq k \leq |\mathcal{G}| + |\mathcal{L}|$ such that: if $v_i \in a$, then $g_k = u_i$ and $b_k = true$, else if $\neg v_i \in a$, then $g_k' = u_i$ and $b_k = false$.
   - for each $\langle u_i, v_i\rangle \in U_2$, there exists $1 \leq j \leq |U_2| + |U_3|$ with $g_j = u_i$ and $b_j = \theta(v_i)$.
   - for each $\langle u_i, v_i\rangle \in U_3$, there exists $1 \leq k \leq |U_3| + |U4|$ such that: if $v_i \in a$, then $l_k = u_i$ and $c_k = true$. Otherwise, if $\neg v_i \in a$, then $l_k' = u_i$ and $c_k = false$.
   - for each $\langle u_i, v_i\rangle \in U_4$, there exists $1 \leq j \leq |U_3| + |U_4|$ with $l_j = u_i$ and $c_j = \theta(v_i)$.
   - for each $v \in \mathcal{G} \cup \mathcal{L}$ such that there is no $\langle u_i, v_i\rangle \in U_1 \cup U_2$ with $u_i = v$, there is $1 \leq j \leq |\mathcal{G}| + |\mathcal{L}|$ with $g_j = v$ and $b_j = v$.

   (b) $[\ell]$`if`(E) $\{[\ell_1]I_1 \cdot S_1\}$`else`$\{[\ell_2]I_2 \cdot S_2\} \cdot S'$, then

   if $E \equiv *$, we have $\langle\theta, \ell\rangle \xrightarrow{\tau\ :\ \top} \langle\theta, \ell_1\rangle$ and $\langle\theta, \ell\rangle \xrightarrow{\tau\ :\ \top} \langle\theta, \ell_2\rangle$;
   otherwise, $E \equiv c_1$ `and` c_2 `and`...`and` c_n, where each $\forall 1 \leq i \leq n : c_i \equiv v_i \vee c_i \equiv !v_i \vee c_i \in \{$`true`, `false`$\}$, with $v_i \in \mathcal{V}_i \cup \mathcal{G}$. Let's divide E into two sub-expressions. The former, $E^{\mathcal{G}}$, is made of the conjunction of every c_i appearing in E, and ranging over *global variables.* More formally:

   $$E_{\mathcal{G}} \equiv \bigwedge_i c_i \text{ such that } c_i \text{ appears in } E \text{ and } v_i \in \mathcal{G}$$

   The latter expression, $E^{\mathcal{V}}$, ranges over local variables and literals:

   $$E_{\mathcal{G}} \equiv \bigwedge_i c_i \text{ such that } c_i \text{ appears in } E \text{ and} \begin{cases} c_i \in \{\texttt{true}, \texttt{false}\} \text{ or} \\ c_i \equiv v_i \text{ and } v_i \in \mathcal{V}_i \text{ or} \\ c_i \equiv !v_i \text{ and } v_i \in \mathcal{V}_i \end{cases}$$

   Now we have that:

   $$\begin{cases} \langle\theta, \ell\rangle \xrightarrow{\tau\ :\ E^{\mathcal{G}}} \langle\theta, \ell_1\rangle & \text{if } \theta \models E^{\mathcal{V}} \\ \langle\theta, \ell\rangle \xrightarrow{\tau\ :\ E^{\mathcal{G}}} \langle\theta, \ell_2\rangle & \text{otherwise} \end{cases}$$

   (c) $[\ell]$`choice`$\{E_1{:}u_{1,1}, \ldots, u_{1,m_1}{:=}v_{1,1}, \ldots, v_{1,m_1}; \ldots E_n{:}u_{n,1}, \ldots, u_{n,m_n}{:=}v_{n,1}, \ldots, v_{n,m_n};\} \cdot [\ell']I' \cdot S'$, then, we apply the following steps for every guarded assignment $E_i{:}u_{j,1}, \ldots, u_{j,m_j}{:=}v_{j,1}, \ldots, v_{j,m_j}$; $(1 \leq j \leq n)$: First of all, we decompose E_j into two sub-expressions $E_j^{\mathcal{G}}$ and $E_j^{\mathcal{V}}$, as we did in the `if` case. Then, we consider the assignment without its guard and compute all the relations

$\langle \theta, \ell \rangle \xrightarrow{a:g} \langle \theta', \ell' \rangle$ (where a is the action and g is the guard) that hold for it (see rule 2a). Now, to take the guard into account, we only retain these relations such that θ satisfies $E_j^{\mathcal{V}}$, and we complete their guards with $E_j^{\mathcal{G}}$.

More formally, for every relation $\langle \theta, \ell \rangle \xrightarrow{a:g} \langle \theta', \ell' \rangle$ holding for the assignment without the guard, then the following holds too:

$$\langle \theta, \ell \rangle \xrightarrow{a:g \wedge E_j^{\mathcal{G}}} \langle \theta', \ell' \rangle \text{ iff } \theta \models E_j^{\mathcal{V}}$$

(d) $[\ell]$`while`$(E)\{[\ell_1]I_1 \cdot S_1\} \cdot [\ell_2]I_2 \cdot S_2$, then

if $E \equiv *$, we have $\langle \theta, \ell \rangle \xrightarrow{\tau\,:\,\top} \langle \theta, \ell_1 \rangle$ and $\langle \theta, \ell \rangle \xrightarrow{\tau\,:\,\top} \langle \theta, \ell_2 \rangle$;
otherwise, $E \equiv c_1$ `and` c_2 `and`...`and` c_n, where each $\forall 1 \leq i \leq n : c_i \equiv v_i \vee c_i \equiv\, !v_i$, with $v_i \in \mathcal{V}_i \cup \mathcal{G}$. Again (see the `if` case), we divide E into two sub-expressions $E^{\mathcal{G}}$ and $E^{\mathcal{V}}$. We then have that:

$$\begin{cases} \langle \theta, \ell \rangle \xrightarrow{\tau\,:\,E^{\mathcal{G}}} \langle \theta, \ell_1 \rangle & \text{if } \theta \models E^{\mathcal{V}} \\ \langle \theta, \ell \rangle \xrightarrow{\tau\,:\,E^{\mathcal{G}}} \langle \theta, \ell_2 \rangle & \text{otherwise} \end{cases}$$

(e) $[\ell]$`skip` $; \cdot [\ell']I \cdot S''$, then

$$\langle \theta, \ell \rangle \xrightarrow{\tau\,:\,\top} \langle \theta, \ell' \rangle$$

(f) $[\ell]$`goto`(ℓ') $; \cdot S$, then

$$\langle \theta, \ell \rangle \xrightarrow{\tau\,:\,\top} \langle \theta, \ell' \rangle$$

(g) $[\ell]$`lock`$(\mathbf{lockId})$ $; \cdot [\ell']I' \cdot S''$, then

$$\langle \theta, \ell \rangle \xrightarrow{\tau\,:\,\neg\mathbf{lockId} \wedge \mathbf{lockId}'=\top} \langle \theta, \ell' \rangle$$

(h) $[\ell]$`unlock`$(\mathbf{lockId})$ $; \cdot [\ell']I' \cdot S''$, then

$$\langle \theta, \ell \rangle \xrightarrow{\tau\,:\,\mathbf{lockId}'=\bot} \langle \theta, \ell' \rangle$$

(i) $[\ell]$`sleep`$(\mathbf{msgId}, \mathbf{lockId})$ $; \cdot [\ell']I' \cdot S''$, then

$$\begin{cases} \langle \theta, \ell \rangle \xrightarrow{\tau:\mathbf{lockId}'=\bot} \langle \theta, \mathbf{wait}_\ell \rangle \\ \langle \theta, \mathbf{wait}_\ell \rangle \xrightarrow{??\mathbf{msgId}\,:\,\top} \langle \theta, \mathbf{lock}_\ell \rangle \\ \langle \theta, \mathbf{wait}_\ell \rangle \xrightarrow{\downarrow\mathbf{msgId}\,:\,\top} \langle \theta, \mathbf{lock}_\ell \rangle \\ \langle \theta, \mathbf{lock}_\ell \rangle \xrightarrow{\tau\,:\,\mathbf{lockId}=\bot \wedge \mathbf{lockId}'=\top} \langle \theta, \ell' \rangle \end{cases}$$

(j) $[\ell]$`wakeup`$(\mathbf{msgId})$ $; \cdot [\ell']I' \cdot S''$, then

$$\langle \theta, \ell \rangle \xrightarrow{\uparrow\mathbf{msgId}\,:\,\top} \langle \theta, \ell' \rangle$$

(k) $[\ell]$`wakeupall`$(\mathbf{msgId})$ $; \cdot [\ell']I' \cdot S''$, then

$$\langle \theta, \ell \rangle \xrightarrow{!!\mathbf{msgId}\,:\,\top} \langle \theta, \ell' \rangle$$

(l) $[\ell]$`start`$(\mathbf{threadTypeName}, \pi_1, \ldots, \pi_n)$ $; \cdot [\ell']I' \cdot S''$, then

$$\langle \theta, \ell \rangle \xrightarrow{!\mathbf{threadTypeName}\pi_1 \ldots \pi_n\,:\,\top} \langle \theta, \ell' \rangle$$

(m) $[\ell]$`rendezvous`$(\textbf{msgId}, u_1, \ldots, u_n) \cdot [\ell']I \cdot S'$, then we have

$$\langle \theta, \ell \rangle \xrightarrow{!\textbf{msgId}\theta(u_1)\ldots\theta(u_n)\ :\ \top} \langle \theta, \ell' \rangle$$

(n) $[\ell]$`accept`$(\textbf{msgId}, v_1, \ldots, v_n)$; $\cdot [\ell']I' \cdot S''$, then

$$\forall \rho = (\rho_1 \ldots \rho_n) \in \{\top, \bot\}^n \ :\ \langle \theta, \ell \rangle \xrightarrow{?\textbf{msgId}\rho\ :\ \top} \langle \theta[v_1 = \rho_1, \ldots, v_2 = \rho_2], \ell' \rangle$$

Finally, assuming that $S_i = [\ell]I \cdot S'$ and $\theta_\top(v_i) = \top, \forall v_i \in \mathcal{V}_i$ we have

$$\langle \theta_\top, \textbf{begin} \rangle \xrightarrow{?\textbf{threadTypeName}a_1\ldots a_n\ :\ \top} \langle \theta_\top[v_1 = a_1, \ldots, v_n = a_n], \ell \rangle$$

for all the possible values of $a_1, \ldots, a_n$ with $a_i \in \{\top, \bot\}$.

3. Σ_i is constructed together with δ_i. More precisely:

   (a) Each time we encounter a **msgId** as a parameter of a `wakeup()` ;, `wakeupall()` ; or `unlock()` ; during the construction of δ_i, we had **msgId** to Σ_i ;

   (b) For each `start`$(\textbf{threadTypeName}, \pi_1, \ldots, \pi_n)$;, we add $\textbf{threadTypeName}\pi_1 \ldots \pi_n$ to Σ_i ;

   (c) For each $\langle \theta, \ell \rangle \xrightarrow{!\textbf{msgId}\theta(u_1)\ldots\theta(u_n)\ :\ \top} \langle \theta, \ell' \rangle$ transition added to δ_i, we add to Σ_i the message $\textbf{msgId}\theta(u_1) \ldots \theta(u_n)$;

   (d) For each `accept`$(\textbf{msgId}, v_1, \ldots, v_n)$;, we add $\textbf{msgId}v_1 \ldots v_n$ to Σ_i ;

   (e) Finally, we also add the set of messages $\{\textbf{threadTypeName}a_1 \ldots a_n | (a_1, \ldots, a_n) \in \{\top, \bot\}^n\}$ to Σ_i.

4. The initial state I_i is $\langle \theta_\top, \textbf{begin} \rangle$.

We can now easily obtain a set of Local Machines from a given CBP. But this is not enough to construct a full GM. Indeed, we need to specify how many instances of each Local Machine exist in the GM. As each Local Machine corresponds to one thread instantiation, and as we can't easily predict the number of threads that will be instantiated, we define the result of the translation of the CBP as a *family of Global Machines.* It will be the set of all Global Machines constituted of any number of the previously obtained Local Machines.

Definition 3.5 Resulting Family of Global Machines.
Given a Concurrent Boolean Program $B = \langle \mathcal{G}, \mathcal{L}, \mathcal{T} \rangle$ and its translation into a set of local machines $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, the **Family of Global Machine** one can construct is:

$$\mathcal{F}(B) = \{\langle \mathcal{G} \cup \mathcal{L}, \{\langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_n, k_n \rangle\} \rangle \text{ such that: } \forall 1 \leq i \leq n \ :\ k_i \geq 1\}$$

□

This implies that we will need to verify the desired properties on this set of GM. We can achieve this thanks to the parametric verification techniques as defined, for instance, in [DRVB02b].
Example 8 The translation of the CBP of Figure 3.3 into a set of Local Machines can be found in 3.4 (especially Figures 3.6 and 3.7). □

3.3 Relation between the CBPs and the GMs

It is now time to show that our translation from the CBP to the GM is trace-conservative. Therefore, we prove the *trace inclusion* between the CBPs and the GMs, in both direction. Roughly speaking: given a CBP and its translation into a family of GM : (i) for each execution of the CBP, there exists a GM whose run has an equivalent trace (in the sense of the mapping functions), and (ii), for every run of a GM of the family, there exists an execution of the CBP with an equivalent trace. This means that we can safely verify

on the GM the properties we are interested in; and immediately deduce the corresponding property on the CBP.

Before we can state the proof of this statement, we need to define some *mapping functions* to establish a correspondence between the global and local states of both models. Our set of mapping function is as follows: the φ function maps the thread instantiations on tuples describing to a local state of a local machines ; the Φ function maps a CBP's global state on a tuple describing a global state of a Global Machine ; and the φ' and Φ' functions can be seen as the inverses of the former and the latter respectively.

Definition 3.6 The φ Function
Given a thread instantiation $s = \langle S, \rho\rangle$, $\varphi(s)$ is defined as the tuple $\langle \rho, \ell\rangle$ if $Relabel(S) = [\ell]I \cdot S'$. □

Definition 3.7 The Φ Function
Given a global state $S = \langle \Gamma, \Lambda, \{I_1, \ldots, I_n\}\rangle$ of a Concurrent Boolean Program, $\Phi(S)$ is defined as the tuple $\langle \Gamma \cup f(\Lambda), \{\varphi(I_1), \ldots, \varphi(I_2)\}\rangle$.

The f function is defined as follows: for every lock l:

$$f(\Lambda(l)) = \begin{cases} \top & \text{if } \Lambda(l) = \textit{locked} \\ \bot & \text{if } \Lambda(l) = \textit{unlocked} \end{cases}$$

We extend in the obvious way Φ to runs of Concurrent Boolean Programs. □

Let us now define the corresponding functions to map a global state of a GM to a global state of a CBP.

Definition 3.8 The φ' Function
Given a local state $s' = \langle \theta, \ell\rangle$ of the Local Machine and a sequence of instruction I, $\varphi'(s', I)$ is defined as the tuple:

$$\begin{cases} \emptyset & \text{if } \ell \equiv \textbf{end} \\ \langle \tilde{\gamma}(I, \ell), \theta\rangle & \text{otherwise} \end{cases}$$

□

φ' is, indeed, a function. To compute its value, one needs to know the sequence of instructions I. As I is uniquely fixed, and provided the well-formedness rules have been respected in I (that is, there is at most one occurrence of ℓ), one is sure of the unity of $\tilde{\gamma}(I, \ell)$.

Of course, we intend to use this function to establish a correspondance between the runs of a Local Machine and the executions of the CBP thread it has been translated from (and vice-versa). Thus, if $T_i = \langle \mathcal{V}_i, S_i\rangle$ is a CBP thread type, and $\mathcal{L}$ is the Local Machine we have obtained by translating T_i, each time we will apply φ' on a local state s' of $\mathcal{L}$, the second parameter (the sequence of instructions) will be S_i (the body of T_i). In the sequel, we will assume that we always know from which CBP thread a Local Machine has been extracted. We will benefit by this assumption to systematically omit the second parameter of φ' and write $\varphi'(s')$ instead of $\varphi'(s', S_i)$.

Definition 3.9 The Φ' Function
Given a global state $S' = \langle G, \{s_1, \ldots, s_n\}\rangle$ of a GM, $\Phi(S')$ is defined as the tuple $\langle \Gamma, \Lambda, \{\varphi'(s_1), \ldots, \varphi'(s_n)\}\rangle$ such that $\Gamma \cup f(\Lambda) = G$.

We extend in the obvious way Φ to runs of Concurrent Boolean Programs. □

As all the $\varphi'(s_i)$ are unique, so is $\Phi(S')$ (remember that a global state of a CBP contains a *multi-set* of threads instances. Therefore, neither the identities, nor the ordering of the threads are relevant).

There is a last problem – inherent in the way we have defined the translation – we have to cope with. We should indeed take into account the 'not-yet-created' threads, that appear in the GM as Local Machines in the special $\langle \theta_\top, \textbf{begin}\rangle$ state (As we have already explained, it is necessary to have this reserving of local machines in the GM if we want to be able to handle the subsequent spawns of threads in the CBP). The *Live* function, we now define, allows one to 'forget about' the non-active local machines in a global state of a GM, and focus only on the one that have a real corresponding thread in the global state of the CBP.

Definition 3.10 The *Live* Function

Given a global state $S' = \langle G, \{s_1, \ldots, s_n\}\rangle$ of a global machine, $Live(S') = \langle G, \{live(s_1), \ldots, live(s_n)\}\rangle$ where

$$Live(\langle S, \rho\rangle) = \begin{cases} \langle S, \rho\rangle & \text{if } S \not\equiv [begin]I \cdot S'' \\ \emptyset & \text{otherwise.} \end{cases}$$

Once again, we extend in the obvious way *Live* to runs of Concurrent Boolean Programs. □

Our last auxiliary function is NT, a counting function, that will turn to be useful in the sequel.

Definition 3.11 Number of Threads in a Global State

Given a global state G of a Boolean Concurrent Program, and a type of thread T_i: $NT(G, T_i)$ is defined as the number of thread instantiations of type T_i in G. □

We can now eventually state the lemmas. Note the restriction on the number of thread instantiations. Indeed, one cannot map an execution of a CBP with n threads of a given type T, onto a run of a GM that has less than n local machines corresponding to T.

Lemma 3.2 (Trace Inclusion: From CBP to GM) *Given $B = \langle \mathcal{G}, \mathcal{L}, \{T_1, \ldots, T_n\}\rangle$, a Concurrent Boolean Program, and $G_B = \langle G, (\mathcal{L}_1, k_1), \ldots, (\mathcal{L}_n, k_n)\rangle$, one of the global machines that can be obtained from B following the translation rules of 3.2.3, given an execution e of B, there exists a run f of G_B such that:*

$$\Phi(e) = Live(f)$$

provided that the number of threads in e never exceeds the number of instances of the corresponding local machine. More precisely: for every global state G in e, we have: $\forall 1 \leq i \leq n \ : \ NT(G, T_i) \leq k_i$.

Proof The full version of the proof is to be found in Appendix B. The proof is by induction on the length of e. The base case is quite straightforward. To establish the induction case, we first make the induction hypothesis (I.H.) that the lemma holds for every run of length at most i. We then look into a run $e'' = e' \cdot S \cdot S'$, of length $i + 1$.

The induction is step is built as follows (one can look at Figure 3.5 for a graphical depiction of the induction step): (1) We look at one single thread's local state s in S and consider all the possible relationships between S and S', depending on the way s is going to 'evolve'. This involves looking at all the possibilities for the next remaining instruction in s. (2) On the other hand we know, thanks to the I.H., that there exists a local machine in the last global configuration of $\Phi(e' \cdot S)$, whose local state l corresponds to s. (3) We can use the information about the transition from S to S', and apply our translation rules, to find all the possible successor of l[4]. (4) It then suffice to compare this set of states to the set of successor states of s, and see that they are equivalent modulo Φ. ■

In the other direction, the inclusion of the runs of a GM into the executions of the corresponding CBP can be proved similarly.

Lemma 3.3 (Trace Inclusion: From GM to CBP) *Given $B = \langle \mathcal{G}, \mathcal{L}, \{T_1, \ldots, T_n\}\rangle$, a CBP, and $G_B = \langle G, (\mathcal{L}_1, k_1), \ldots, (\mathcal{L}_n, k_n)\rangle$, one of the GM that can be obtained from B following the translation rules of 3.2.3, given an execution f of G_B, there exists a run e of B such that:*

$$\Phi'(f) = e$$

Proof To prove this lemma, one can apply the same reasoning that was previously used for Lemma 3.2. ■

Thanks to the previous lemmas, we can now state the following equivalence theorem.

[4] Of course, we also have to take into account the *semantics of the GM* in this step.

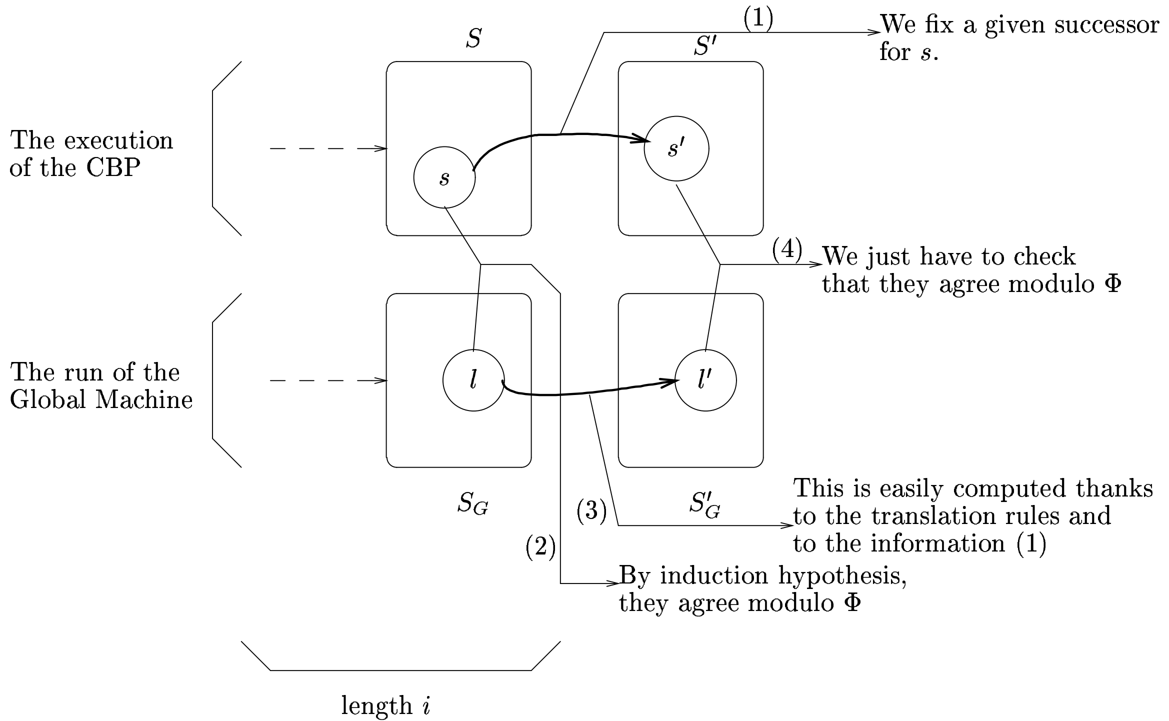


Figure 3.5: A graphical depiction of the inductive step to prove Lemma 3.2. The boxes with round corners represent global states/configurations (of the CBP and of the GM respectively). The thick arrows represent the transition relation. The numbers on the straight arrow refer to the different stages of the inductive step.

Theorem 3.4 (Trace Equivalence) *Given $B = \langle \mathcal{G}, \mathcal{L}, \{T_1, \ldots, T_n\}\rangle$, a CBP, and $\mathcal{F}(B)$, its corresponding family of GM :*
e is an execution of B **iff** *there exists a GM $G \in \mathcal{F}(B)$, and a run f of G such that:*

$$\Phi'(f) = e$$

Proof Stems from Lemma 3.2 and Lemma 3.3. ■

3.4 The `CBP2GM` compiler

All the concepts presented in 3.2 have been implemented in a tool called `CBP2GM`. This compiler takes a `.cbp` file as input, which should respect the grammar given in the previous chapter at Figure 2.1. Its output is a file describing the corresponding Global Machine, in a format that is suitable as input for the BABYLON toolkit [ADG+02] (see http://www.ulb.ac.be/di/ssd/lvbegin/CST/index.html for more practical information about the BABYLON project).

Roughly speaking, `CBP2GM` works in two phases:

1. It first constructs the control-flow graph of the CBP it has received as input. This is quite natural, but yet really valuable to us, because the CFG is often the basis of the optimisations one could want to apply to reduce the size of the model (see Chapter 4).

2. It then computes a *forward fix-point* to generate the local states of each Local Machine (remember that we construct one Local Machine per type of thread). Indeed, the rough way to compute all the local states of a Local Machine, is simply to build the Cartesian product of the set of every possible valuation of the local variables, and the set of control-flow points. But, given a initial value of the local variables – and we can easily find these values if we look at the `start` primitives – the set of local

states that the Local Machine is able to reach is much smaller! We thus begin by constructing the set of Local Machine states that corresponds to the initial states of the threads (whose control point is the entry point of the thread). Then, we apply the forward fix-point in order to compute all the possible valuations that are reachable at each CFG point.

Example 9 As an example, look at Figure 3.7 and Figure 3.6. They present the Local Machines for the two types of threads presented at Figure 3.2. The states (represented by ellipses) contain their respective names. Each name beginning by _CBP2GM_ has been generated by CBP2GM (One can see this as an implementation of the *Relabel*() function). The transitions' labels are of the form: 'synchronisation label : guard -> action' (when there's no action to execute, we simply omit it). These Local Machines have been automatically generated by CBP2GM. □

The CBP2GM compiler is *free software*[5], and is released under the General Public Licence. It is publicly available as a part of the BABYLON project, on the project's homepage[6].

3.5 Discussion

In this chapter, we have presented sufficient theoretical results to automatise the process of translating a JAVA program into a CBP, and a CBP into a GM, which was the aim of our verification framework (Figure 1.3). We have also shown that our translation rules from the language of CBPs to the GM is correct (see Theorem 3.4. Finally, we have presented a tool, CBP2GM that implements the translation of the CBPs into Global Machines.

Up to now, we haven't looked into the quality of the generated models. We already know that they are adequate for the verification of safety properties. But this verification could be intractable if the models were too big to be analysed within a reasonable amount of time and memory. In the next chapter, we present several ideas that could be useful to reduce the size of the models.

[5] See the web site of the Free Software Foundation (http://www.gnu.org) for more information about free software.
[6] http://www.ulb.ac.be/di/ssd/lvbegin/CST/index.html

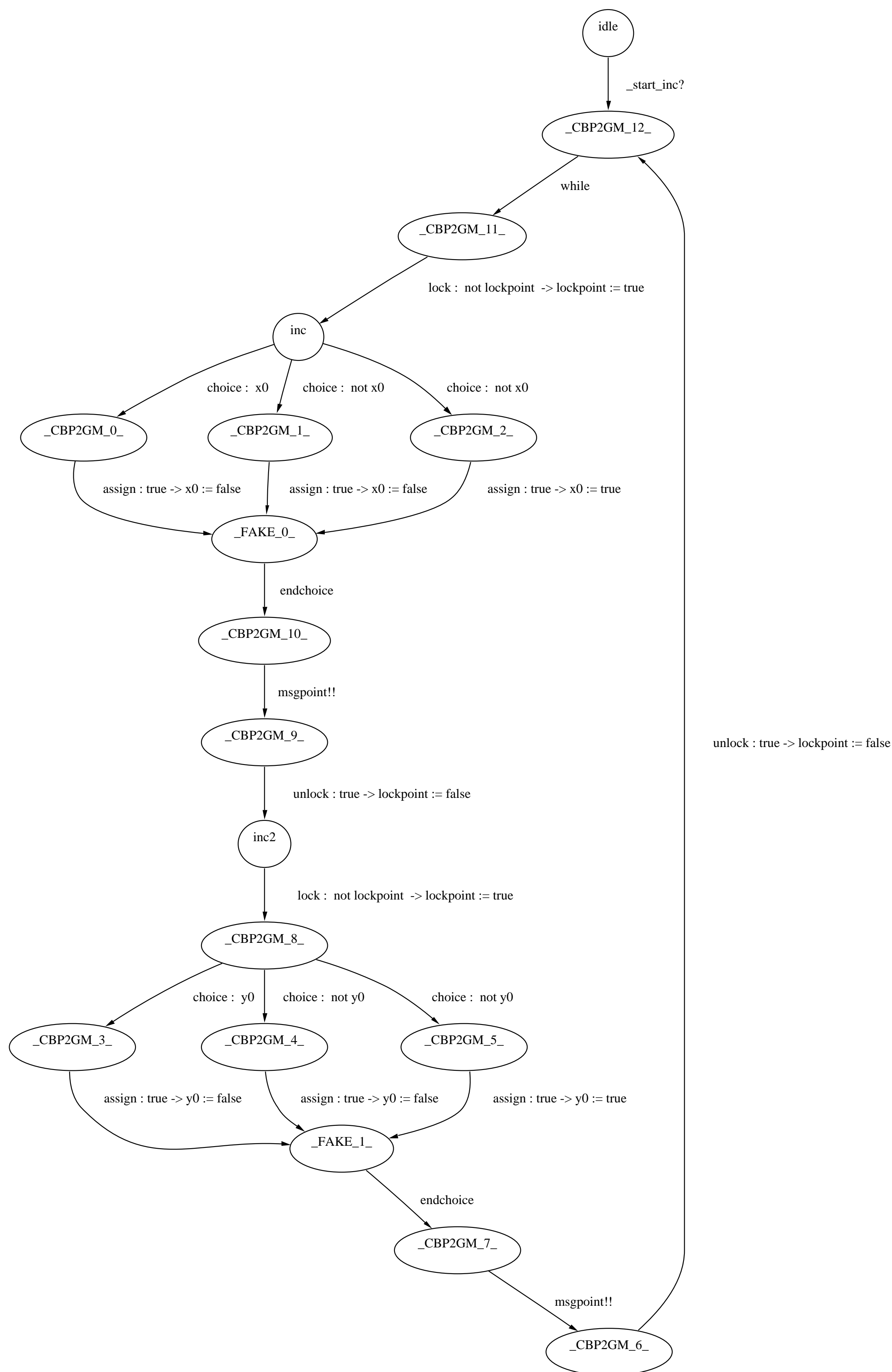


Figure 3.6: The Local Machine for the thread type `Inc`, generated by `CBP2GM`.

Figure 3.7: The Local Machine for the thread type Dec, generated by CBP2GM.

Chapter 4

Optimising the Model

In this chapter, we survey several static analysis techniques that can be useful in order to *reduce the size* of the models we extract from the CBP, as explained in the previous chapter. Our survey covers mainly two works: one by Scott Stoller [Sto00], and one by Corbett [Cor00].

4.1 The need for optimisation

As said in the previous chapter, the models constructed by `CBP2GM` are almost certainly useless, as their sizes could be intractable. As an example, look at Figure 3.6: it is easy to find out several transitions that could be suppressed, by concatenating them with their successors. For instance, the many dummy transitions that have been introduced by the compiler, in order to 'close' a `choice` or `while` statement.

But more than that, one could intuitively think about grouping other sets of transitions. For instance, two transitions that manipulate *local boolean variables*, which encode predicates related to local variables, could easily be merged. Indeed, if another thread ever interleaved one or several of its transitions between these two, this would have absolutely no influence on them. Because the two transitions we are concerned with (let's call them t_1 and t_2) manipulate *local data only*, they can be seen as *independent* of the other ones. Executing:

$$s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \xrightarrow{t'} s_4$$

instead of:

$$s_1 \xrightarrow{t_1} s_2 \xrightarrow{t'} s_5 \xrightarrow{t_2} s_4$$

is completely equivalent, if we are interested in the reachability of s_4. We can thus apply the same reasoning to any other transition that could happen between t_1 and t_2, and consequently decide to merge them:

$$s_1 \xrightarrow{t_2 \circ t_1} s_3 \xrightarrow{t'} s_4$$

The question that shows up is thus: 'how can we generalise this idea?'. This chapter addresses this problem by giving a kind of brief survey of the literature about reduction (and more particularly in the JAVA case). The main paper we are going to follow is by Scott Stoller [Sto00]. We summarise this work here, and try to relate it to similar works, namely by Corbett [Cor00] and Lamport [LC98]. We also show how to use these techniques in the case of the CBP.

4.2 Stoller's model checking techniques for distributed JAVA programs

In this section, we summarise Stoller's work as presented in [Sto00]. We also show how to adapt these techniques to the case we are dealing with.

4.2.1 System model

Definition 4.1 Concurrent system
A **(concurrent) system** is a tuple $\langle \Theta, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{ld}, \mathcal{O}_{com}, s_{init}, \mathcal{T} \rangle$, where:

- Θ is a finite set of threads. A thread is finite set of control points ;
- $\mathcal{O}_{unsh}$ is the set of *unshared objects* ;
- $\mathcal{O}_{syn}$ is the set of *synchronisation objects* ;
- $\mathcal{O}_{ld}$ is the set of objects respecting a given *locking discipline* ;
- $\mathcal{O}_{com}$ is the set of *communication objects* ;
- s_{init} is the initial state ;
- $\mathcal{T}$ is a finite set of transitions of the form $\langle S, G, C, F \rangle$, where S and F are control points of some thread, G is a boolean guard and C is a command (an expression built from operations on objects and mathematical functions).

□

This division of the set of objects appeals some comments. The *unshared objects*, are all the objects that are accessible by one single thread during each possible execution of the system (we define this latter notion more precisely bellow).

The *synchronisation objects* are used to encode the synchronisation states of the JAVA objects. It thus contains information such as the state of the object's lock, as well as its wait-set (In JAVA, this information is part of the object). A synchronisation object has three fields: *owner*, which refers to the thread owning the object (*free* if no such thread exists) ; *depth*, the number of unmatched *acquire* operation on the object's lock ; and *waiters*, the set of waiting threads. It should be easy to see how operations like the acquisition of a lock, the release of a lock as well as `notify` and `notifyAll`-like operations modify the fields of the synchronisation objects (the reader is referred to [Sto00]).

The locking discipline we write about in Definition 4.1 is intuitively explained as follows: once the object has been initialised, it must be and remain, either read-only or lock protected throughout its whole lifetime. By 'lock protected' we mean that there exists a given lock such that, each time the object is accessed by a thread, this thread owns the lock.

Finally, the *communication objects* are all the objects that don't fall in the three previous categories. They are the objects any thread can access at any time in order to, for instance, exchange values with other threads. It is important to notice that it is always *safe* to classify an object as *communicating*: if indeed the object is never accessed by more than one thread, we can consider it falls in the degenerate case of communicating objects where no communications are performed through it. The converse is not true, as a communication object is nor a synchronisation object, nor an unshared object, and could clearly violate the locking discipline.

Let's now say a few words about the behavioural aspect of the systems that are considered in [Sto00]. A *state* is a pair $\langle L, V \rangle$, such that L is a collection of control points (one from each thread), and V is a valuation of the objects. Given a state s and a thread θ, we denote by $s(\theta)$ the control point of θ in s. The definitions of *pending* and *enabled* transitions are the usual ones. A *sequence* is a function which domain is $\mathbb{N}$, or a finite prefix of $\mathbb{N}$. Finally, an *execution* of a system $\mathcal{M}$ is a sequence σ of transitions such that there exist states s_0, s_1, s_2,... and $s_0 \xrightarrow{\sigma(0)} s_1 \xrightarrow{\sigma(1)} s_2 \ldots$

4.2.2 Visible and invisible

This subsection presents the notion of *visible* and *invisible* operations, following section 3.5 of [Sto00]. Once the operations have been classified following this bipartite scheme, one can greatly reduce the exploration of the state-space, by disallowing context switches before transitions that don't contain visible operations. Here is how the classification works:

1. All the operations on *communication objects* are visible. The operations on *synchronisation objects* that may block are visible. These are the operations like acquiring a lock, or performing a wait.

2. A *transition* t is visible if its guard or command contains a visible operation, or if t is a part of a non-deterministic choice. Stoller devises an over-approximation of this latter condition by classifying as visible all the transitions that make a `notify()` (as well as every call to a method of `java.util.Random`).

3. A *control point* S is visible if all transitions starting in S are visible.

4. A *state* s is visible if all control points in s are visible.

4.2.3 Conditions on systems

Before we can explain how to exploit the classification of the previous paragraph, we need to introduce some conditions on the systems we are going to consider (this is directly taken from [Sto00], section 3.6):

Separation : For every thread θ, for every control point $S \in \theta$, all transitions that start in S are visible, or all of them are invisible.

InitVis The first transition of every thread is visible.

BoundedInvis There must exist a bound b on the length of continuous sequences of invisible transitions by a single thread.

DetermInvis For every reachable state s, every thread θ has at most one enabled invisible transition in s.

NonBlockingInvis For every thread θ, for every invisible control point $S \in \theta$, for every reachable state s containing S, the set of transitions of θ, that are enabled in s, is non-empty.

As pointed out by Stoller, all these conditions but **BoundedInvis** are satisfied by any reasonable model of JAVA programs. **BoundedInvis** might not be satisfied, but this is easy to detect.

4.2.4 A coarser model thanks to invisible states

In section 4 of [Sto00], Stoller recalls Godefroid's State-less Selective Search (SSS) Algorithm [God96, God97]. Remark that these kind of exploration algorithms cannot directly be applied to the parametric case, as they assume a *finite state-space.* In section 5, Stoller explores *Two Approaches to Lock-Based Reduction.* The former is a modified version of SSS. The latter, we are about to explain, works by *aggregating* a *visible* transition and the subsequent sequence of *invisible* transitions of the same thread.

Given a sequence of transitions σ, the composition works as follows. Lets $cmd_c(\sigma)$ be the sequential composition of the commands of the transitions in σ. Moreover, let $guard_c(\sigma)$ be the weakest precondition ensuring that, when each transition t in σ is executed, t's guard holds:

$$guard_c(\sigma) = guard(\sigma(0)) \wedge \bigwedge_{0<i<|\sigma|} wp\Big(guard(\sigma(i)), cmd_c(\sigma(0 \ldots i-1))\Big)$$

where $guard(\langle S, G, C, F\rangle) = G$. Following [Gri81] $wp(R, S)$ is defined as a predicate representing the set of all states such that the execution of S from one of these states is guaranteed to finish in a state satisfying R.

Given a system $\mathcal{M}$ which satisfies some conditions, we can now define its reduced version by the composition of transitions:

Definition 4.2 Reduced System

For a system $\mathcal{M}$ that satisfies the locking discipline of section 4.2.1, as well as **Separation**, **BoundedInvis** and **DetermInvis**, we define $\mathcal{C}(\mathcal{M})$, that is the similar to $\mathcal{M}$, except for its set of transitions, that is constructed as follows.

Let b be the bound holding in **BoundedInvis** for $\mathcal{M}$. For each visible transition $t = \langle S, G, C, F\rangle \in \mathcal{T}$, for each sequence σ of invisible transitions such that:

- $|\sigma| \leq b$;
- $guard_c(\langle t \rangle \cdot \sigma) \neq false$ (where $\cdot$ denotes the concatenation of sequences of transitions) ;
- the destination control point of t is the origin of $\sigma(0)$;
- for all $i < |\sigma| - 1$, the destination of $\sigma(i)$ is the origin of $\sigma(i+1)$;
- the destination of the last transition of σ is visible

$\mathcal{C}(\mathcal{M})$ has the transition $\left\langle S, guard_c(\langle t \rangle \cdot \sigma), cmd_c(\langle t \rangle \cdot \sigma), extr(\sigma(|\sigma|)) \right\rangle$, where $extr(\langle S, G, C, F \rangle)$ is F. □

Theorem 4.1 *Let $\mathcal{M}$ be a system satisfying the locking discipline,* **Separation**, **InitVis**, **BoundedInvis** *and* **DetermInvis**. *$\mathcal{M}$ and $\mathcal{C}(\mathcal{M})$ have the same reachable visible states.*

Proof See theorem 4 in [Sto00]. ■

Example 10 As an example, let us come back to the example of Figure 3.2 and let's see what is visible in the code of the `dec` thread. This thread can be seen – in a very informal, but yet comprehensive way – as a transition system, shown in Figure 4.1. As one can see, the `wait()` operation has been split into two transitions: the former labelled by *begin_wait*(), the latter by *end_wait*(). An extra state – represented here by an empty circle – has also been added. This is a simplified representation of what happens when a thread performs a `wait`. *begin_wait*() represents the release of the lock, as well as the move of the thread to the *sleep state*. *end_wait*() represents the fact that the threads is awoken and re-acquires the lock. We don't present these operations in further detail as they are not relevant to what we want to show: the possibility to get a much smaller model from the one presented here. One should refer to Stoller's paper to know how to correctly label these transitions, in order to let them manipulate the various fields of the synchronisation object ℓ_1.

Following Stoller's classification, we see ℓ_1 as a synchronisation object. The `Point` object `p` (which should be declared in the `main` function and passed to both thread's constructors in the JAVA program) is an object observing the *locking discipline*. ℓ_1 is the object that allows to represent `p`'s synchronisation sate. There are nor communication objects, nor unshared objects.

On Figure 4.1, we have represented the visible transitions with their respective guards and commands in red. These visible transitions are potentially blocking: the *end_wait*() could block because the thread tries to re-acquire the lock, which could be owned by another thread, and the *acquire*() could block for the same reason.

We can now consider all the sequences of transitions that begin by one of these visible transitions, and end in a visible state; merge these transitions following the rule we have given above; and obtain the reduced transition system presented at Figure 4.2. Remark that some simplifications could be performed on the commands of the transitions present in this model. For instance, a transition bearing $\ell_1.acquire() \cdot y = y - 1 \cdot \ell_1.release()$ can be simplified in $y = y - 1$, because the lock is taken and released atomically in the original command. We write in green the parts of the commands that can be suppressed. □

Example 11 We can also largely lower the size of the model we had obtain for the Bakery Algorithm. The `IChoose` and `JChoose` clearly follow a locking discipline. All the other variables are communication variables. □

4.2.5 Smaller Local Machines thanks to invisible states

After this short introduction of Stoller's approach to reducing a finite-state model of a JAVA program, we can go back to our main concern, and try to apply these techniques in the case of Local Machines. In this section, we present the adaptation of Stoller's techniques to CBP and GM in a very informal way. Further works are still needed in order to clarify and improve the results.

First of all, we have to remark that Stoller's work has been designed for *finite-state* system. He was indeed interested in applying some kind of *selective-search algorithm* [God96], which doesn't finish when applied to systems with infinite state-space. We argue that the same reasoning can be applied in the parametric case,

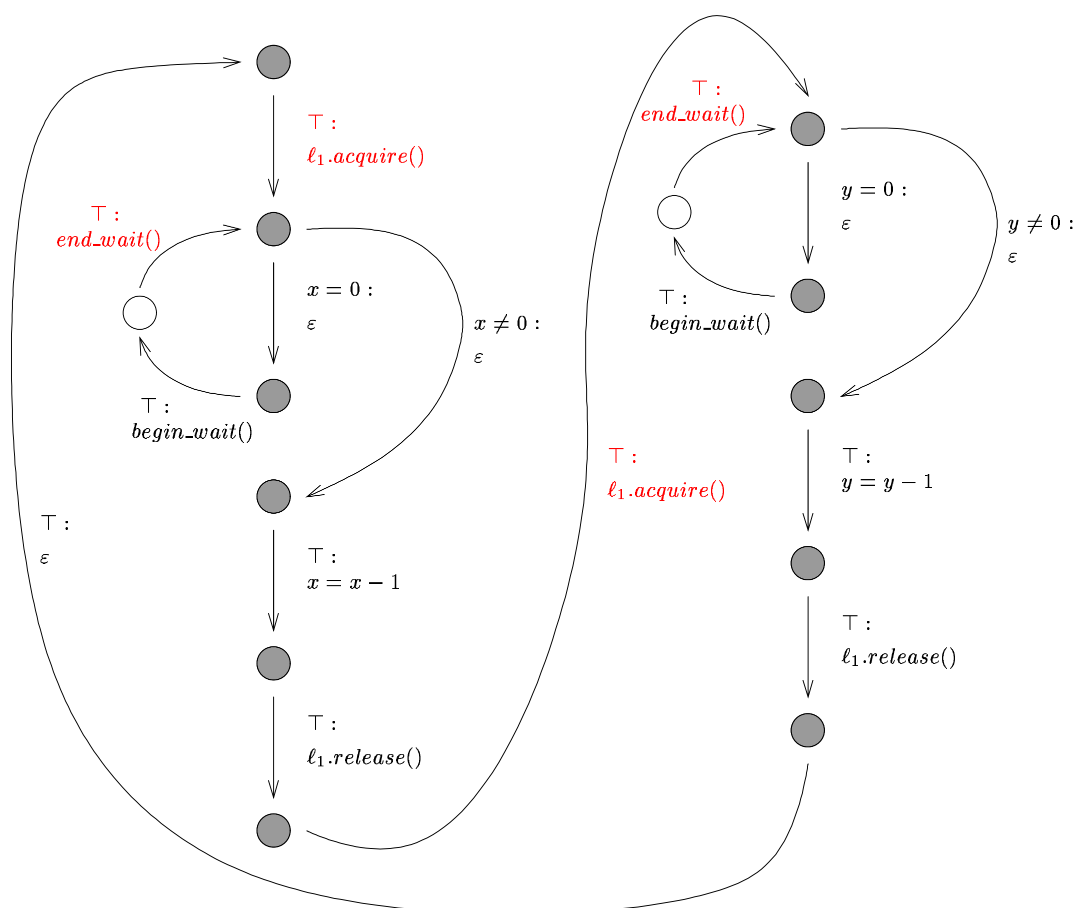


Figure 4.1: The transition system for the `dec` thread of Figure 3.2. Each transition is drawn by an arrow, labelled by 'guard:command' (where ⊤ denotes *true* for the guard, and ε denotes the empty command).

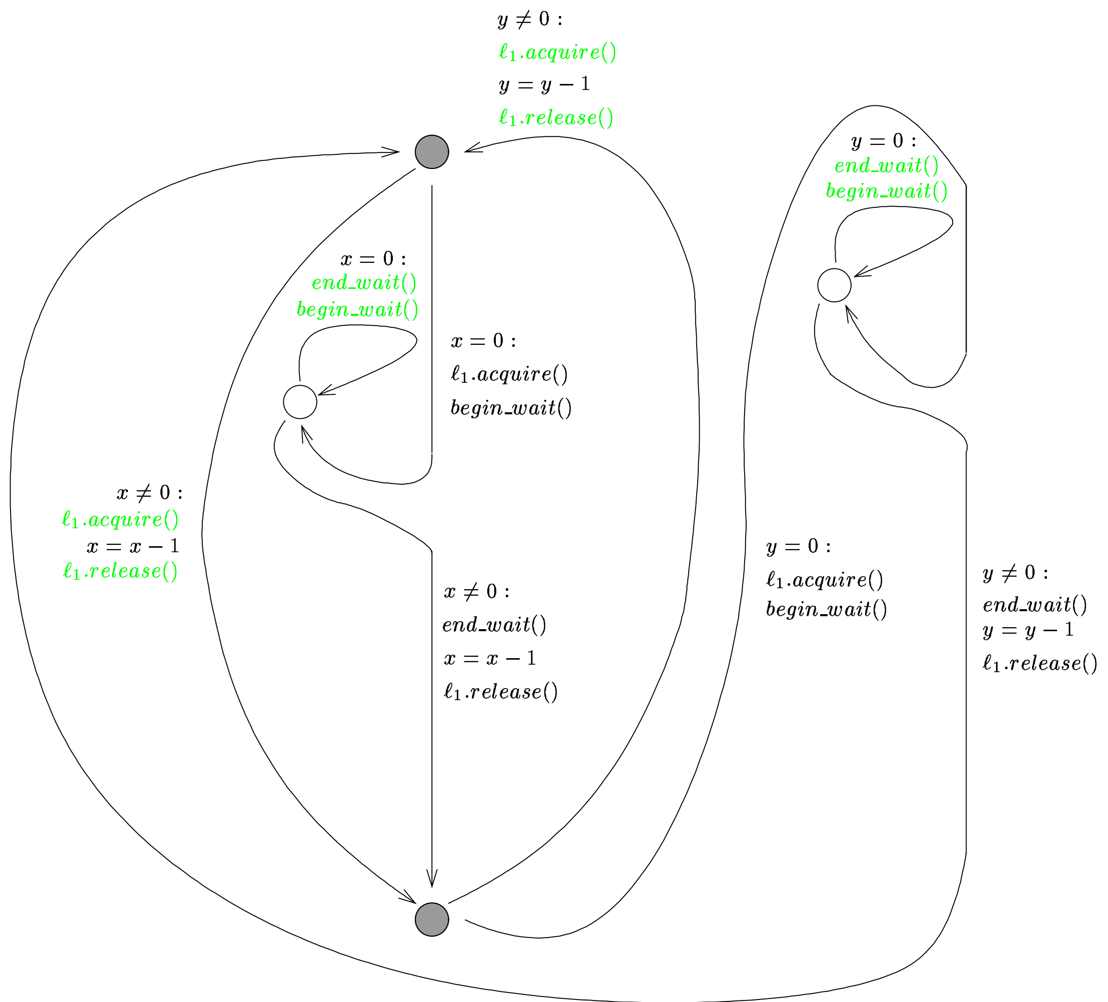


Figure 4.2: The reduced transition system for the `dec` thread of Figure 3.2, obtained from Figure 4.1

mainly because the transformations applied to reduce the model concern the way we model a *type of thread*, no matter how many times it will be instantiated.

Beside the problem of the SSS algorithm, the limitation to finite-state systems appears in [Sto00] when the author explains how to encode the `notify()` primitive of JAVA with respect to the fields of the corresponding synchronisation object. It is explained that $|\Theta| + 1$ transitions should be created: one transition to throw an `IllegalMonitorStateException`, one transition for the case where there are no waiting threads, and $|\Theta| - 1$ transitions to allow each potentially waiting thread to wake up. As only one of these transitions will be fired, one single waiting thread, if any, will be woken up. Remark that in the case of the Local Machines, we don't explicitly handle the set of waiting threads because the so-called *sleep state* of the threads is eventually encoded as a Local Machine state, where the thread waits to catch a specific message which will wake it up. The limitation is thus inherent in Stoller's model.

Given a CBP Let's now see how we can reduce the size of the Local Machine we extract from it. Here is how we adapt the four way classification of objects:

Unshared Objects As the notion of *object* doesn't exist in CBP, we might consider *unshared variables*. These are all the local variables of the CBP threads.

Synchronisation Objects As said before, we don't explicitly handle the wait sets in CBP. The synchronisation objects are thus the *locks* and *messages*.

Objects observing a Locking Discipline In this category, we put all the global variables such that there exists a lock ℓ that each thread owns when it accesses the variable. This is really easy to detect in a CBP, at least much easier than in a JAVA program.

Communication Objects All the global variables that are not in the previous category fall in this one. We call them *communication variables*

The classification of *visible* and *invisible* operations and transitions is quite natural:

1. All the operations on *communication variables* are visible.
2. All the potentially blocking operations on *synchronisation objects* are visible. These are mainly the `lock`, *wait* (recover from a `sleep`), `rendezvous` and `accept`.
3. The `wakeup` operation is visible.

Everything else is *invisible*. Note that a non-deterministic assignment (`choice`) remains invisible as we assume that it is always used to encode a deterministic assignment in the concrete JAVA program.

The concatenation of transitions is carried out as explained in [Sto00]. Note however that some concatenation might not be possible, due to the limitations of the Local Machines (for instance, one cannot have transition with both $msg_1 \uparrow\uparrow$ and $msg_2 \uparrow\uparrow$). We might thus sometimes be forced to classify invisible transitions as visible[1].

Example 12 We can apply the policy explained here to the CBP of the `dec` thread of Figure 3.2 and obtain the Local Machine of Figure 4.3. With only seven states and seventeen transitions, it is much smaller than the Machine we had previously obtained (The Local Machine of Figure 3.7 has twenty-four states and thirty-two transitions). Note that this reduced Machine fits well the reduced transition system of Figure 4.2. □

4.3 Static analysis techniques

While exposing the classification of the previous section, we haven't explained how to determine when an object respects a given locking discipline. As one could expect, this is not an easy task. A solution to this problem can be found in [Cor98] and [Cor00] (the latter being a generalisation of the former).

In [Cor00], Corbett presents five reductions based – similarly to Stoller – on the notion of *visible and invisible transitions*. A transition is invisible when it manipulates local variables only. By *local*, we mean

[1] Remember that this is always safe.

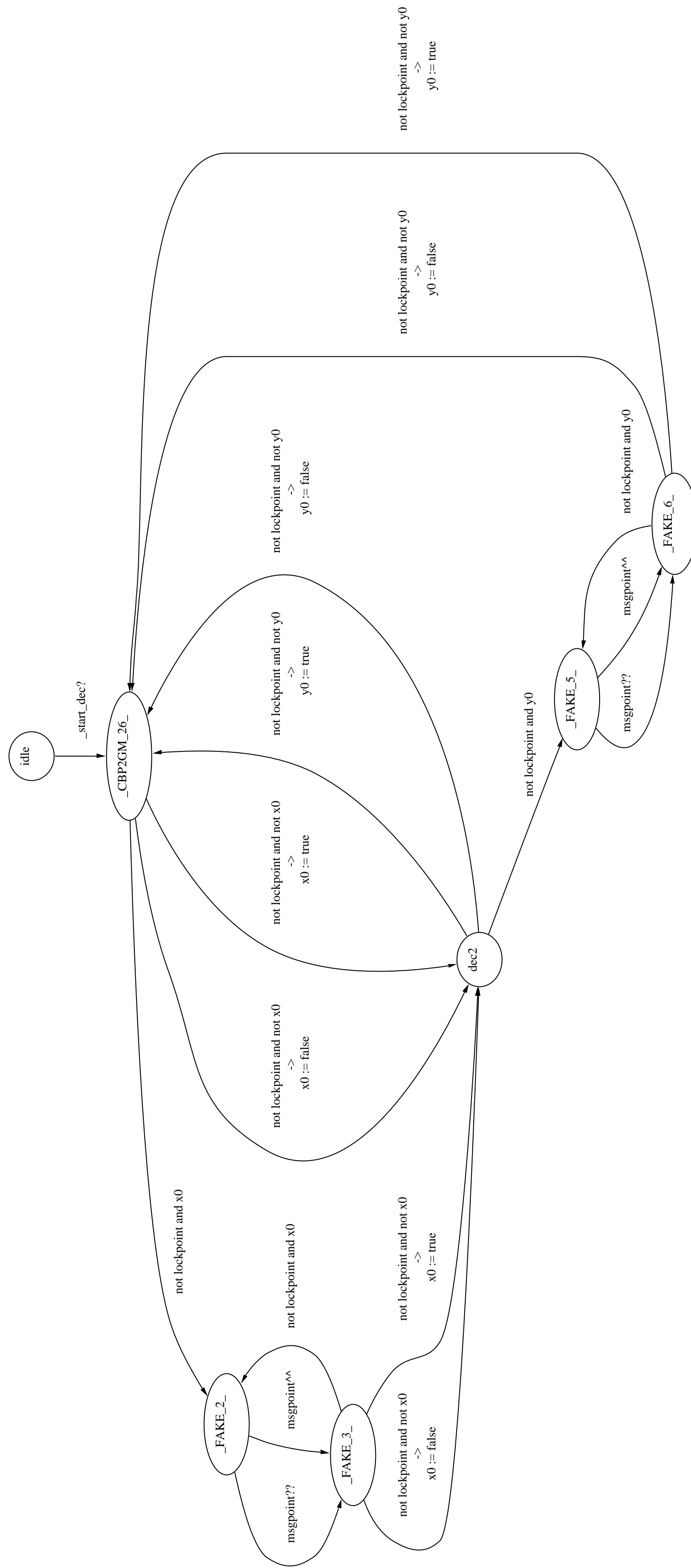


Figure 4.3: The reduced Local Machine for the `dec` thread of Figure 3.2

here 'that can be accessed by one single thread'. Working at the byte-code level, Corbett identifies several instructions that could possibly be visible: `get`, `put`, `load`, `store` (instructions that manipulate memory contents), `monitor_enter`, `monitor_exit` (instructions to take and release locks), `wait`, `notify`, `notifyAll` and `start`. In some contexts, however, these instructions can be classified as invisible:

Owned variables reduction A static variable that is accessed only by some code reachable from `main()` or by code reachable from the `run()` method of a thread that is allocated only once, is *owned* by the thread. A heap-allocated variable is *owned* by the thread that allocates it if it is only reachable from stack variables or static variables owned by the same thread. One can make invisible every transition that performs a `get`, `put`, `load`, `store`, `monitor_enter` or `monitor_exit` on these variables. It is important to remark here that a *change of ownership* can be acted when, for instance, an object $\mathcal{O}$ is passed by thread T_1 to thread T_2, and T_1 deletes its reference to $\mathcal{O}$.

Protected variables reduction This class of variables is similar to the class $\mathcal{O}_{ld}$ of [Sto00].

Relock reduction This reduction simply amounts to suppress nested locks.

Notify reduction One can compose an transition doing a `notify` with the next transition, provided this transition is invisible, or is a `monitor_exit`.

Unlock reduction One can compose a `monitor_exit` with the next transition, when the latter is invisible, or is itself a `monitor_exit`.

In order to apply the two first reductions, Corbett describes some techniques based on *shape analysis*. The problem of *shape analysis* is to construct an approximation of the run-time heap structure. The classical approach, described in [CWZ90], consists in building, for each program statement, a graph that represents the heap at every execution of the statement. Each node of the graph represents one or several heap cells, and the (directed) arcs stand for the pointer relationships among the heap. Such a graph is called a *storage structure graph* (SSG).

In a SSG, there are two types of nodes: *variables nodes* (one for each statically allocated variable and for each stack allocated reference variable), and *heap nodes*. There are one or two heap nodes in the graph for each allocator (an allocator is a call to the `new` operator). If we construct the SSG for a statement s that is in the same loop than a given allocator A, then the SSG of s contains two heap nodes for A. One of them is the *current node*, that represents the *current* instance of the class created by A. The other one is a *summary node* that sums up all the previously created instance of this class. If s is not in the same loop than A, then there is no summary node for A in the SSG of s.

Example 13 For instance, look at Figure 4.4. Figure 4.4(a) presents a code excerpt that creates elements of a list and head-insert them. The rest of Figure 4.4 presents the evolution of the computation of the SSG (for more details about how to compute and SSG, please refer to [CWZ90]). The summarisation operation, presented at Figure 4.4(e) is performed only when we reach the end of the `while` loop. It consists simply in redirecting all the edges from/to current nodes to the corresponding summary nodes. □

In addition to the previously mentioned informations, Corbett explains how to compute *one-to-one relationships*, which are necessary to compute the protected variables reduction. For this reduction, one needs to be sure that *each time* a given variable/object is accessed by a thread, this thread *owns the same lock*. However, an SSG-edge pointing to the summary node for an allocator A to the summary node for an allocator B means that objects allocated by A *could* point to objects allocated by B. An edge in a SSG is thus marked *one-to-one* when it goes form a node for allocator A to a node for allocator B, and we are sure that each object allocated by A points one and only one object allocated by B.

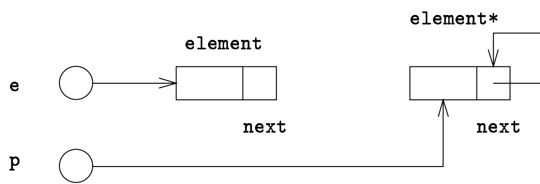
The information collected through the SSG is clearly useful in order to decide, for instance, which variables of a JAVA program follow the locking discipline of [Sto00], or which are candidate for the protected variable reduction of [Cor00].

4.4 Related Works

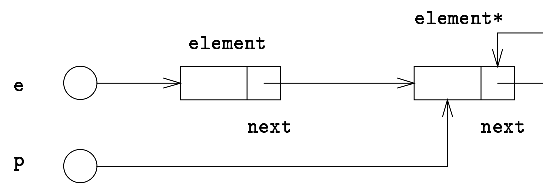
In this section, we present several works related to the problems we have just addressed. Many of them could be of interest for the purpose of reducing a system's model.

```
1:  i = 0 ;
2:  p = null ;
3:  while (...)  {
4:     e = new element(i) ;
5:     e.next = p ;
6:     p = e ;
7:     ++i ; }
```

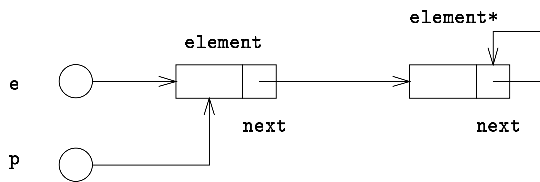
(a) A code excerpt



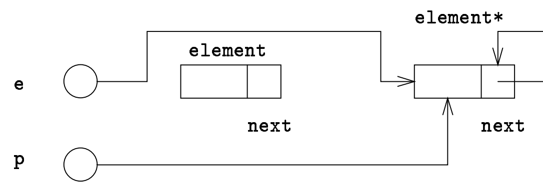
(b) After statement 4



(c) After statement 5



(d) After statement 6, before summary



(e) After summary

Figure 4.4: The evolution of the computation of the SSG for the piece of code (a). The circles represent variables nodes; the rectangles represent heap nodes. The current nodes are labelled by `element`, and the summary nodes by `element*`.

4.4.1 On the reduction problem

Lipton's theorem The problem of *reduction* has been dealt with for many years. The seminal paper is by Lipton [Lip75]. He defines the problem as follows. Given a system P, and one of its statements S, let's define P/S as the *reduction* of P by S. It is simply P in which S has been made atomic. Lipton was interested in studying the relationship between P and P/S (more particularly: 'does P/S stop iff P stops?'). The concepts of *right movers* and *left movers* are thus introduced. If we come back to our introductory example (see 4.1), we can illustrate this concept thanks to transition t_2 which is a *right-mover* : it can be postponed in the sequence of transitions until t' – independent of t_2 – is executed. This allow to isolate t_1 and t_2 in order to compose them. A *left-mover* is symmetrical: it is a transition that can be taken in order to put apart a given sub-sequence of transitions. The same kind of reasoning is applied in the proofs of [Sto00].

Further works by Lamport and Schneider generalise these ideas (see [LS89] and [LC98]). The reductions of [Sto00] are partly inspired of the Eraser algorithm [SBN+97] of Savage *et al.* to detect race conditions.

4.4.2 On reducing the exploration of a system's state-space

Invariant-based techniques Techniques to improve the exploration of the state-space of a Global Machine have already been described by Delzanno *et al.* in [DRVB01]. In this paper, the authors explain how to take advantage of the existence of *structural place invariants* of Petri nets[2]. These are linear equations parameterised by an initial marking and ranging over the markings of the places. As they describe an *over-approximation* of the set of reachable states, one can use to lower the set of states to one needs to keep in memory during the exploration: every state that does not satisfy the invariant can be forgotten.

It might be interesting to remark that, when a given object $\mathcal{O}$ respects the locking discipline, one can expect to find this information in the invariants of the Petri net representing the system. Indeed, if the mutual exclusion is enforced: 'there can be no more than one process in the critical section'. This quoted sentence *is* an invariant of the system. Thus, when we apply the previously presented reduction (see 4.2), we somehow make use the invariants to reduce the system. Remark however that our reduction is even stronger. It does not only forbid other threads to enter the critical section, but it also prevents any interleaving of other threads whether these are in the critical section or not (provided that these threads are independent. Otherwise, some transitions will be made visible).

4.4.3 On the problem of shape analysis

Removing unnecessary synchronisation The idea of analysing the heap structure in order to try and detect which objects are *owned by a single* thread had already been presented by Bogda and Hölzle in [BH99]. Their aim was to remove *unnecessary synchronisations* that arise often in thread-safe libraries provided by JAVA. These libraries usually give access to classes which methods are all `synchronized`. In many cases, a thread calls one of these methods on an object it *owns*, or for which it already owns the lock. In these two cases the (re-)acquiring of the lock induces an undesirable overhead that the authors where trying to get rid of.

TVLA The problem of *shape analysis* has recently been addressed by the team of Mooly Sagiv in [SRW99, LAS00]. They describe a tool, called TVLA (for 'Three-Valued Logic Analyzer'), which allows to 'automatically construct static-analysis algorithms from an operational semantics, where the operational semantics is specified using logical formulae'.

A typical input to TVLA is a set of predicates and a description of a program's control flow graph. The edges of the CFG are annotated by *predicate-modifiers* – logical formulae that describe how the statement corresponding to the edge modifies the predicates. The following table presents an example of predicates one could use to describe the heap of a program, and try and solve the problem of shape analysis (from [LAS00]) The first part of the table presents the *core* predicates – those that describe the basic structure of the memory. The second part lists the *instrumentation* predicates, that are defined thanks to the core predicates. These instrumentation predicates add more expressiveness to the result TVLA computes, and are necessary to prove certain properties (see bellow for the applications of TVLA). Here is the table:

[2] Remember that a Global Machine is first translated into a Petri net as described in [DRVB02b].

Predicate	Intended meaning	Defining formula
$x(v)$	Is v pointed to by variable x?	–
	...	
$n(v_1, v_2)$	Does the n-field of v_1 point to v_2?	–
$r[n, x](v)$	Is v reachable from program variable x using field n?	$\exists v_1 : (x(v_1) \wedge n^*(v_1, v))$
	...	
$c[n](v)$	Does v reside on a directed cycle via dereferences along n-fields?	$n^+(v, v)$
$is[n](v)$	Is v pointed to by more than one n-field?	$\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$

The output of TVLA is a conservative description of the values of the predicates at each program point. So, with the previously mentioned predicates, one would get a quite precise description of the pointer relationships that hold at each statement. The results of TVLA are expressed in Kleene's *three-valued logic*. This extension to the classical two-valued logical admits, in addition to 1 and 0 (for true and false, respectively) a special value 1/2 for cases that can be 0 or 1. This allows to compactly represent unbounded structures. For instance, if we want to finitely and conservatively represent any simply linked list, we could use a *summary node* u to represent any number of nodes, and set the predicate $next(u, u)$ to 1/2. This would mean that *some* nodes represented by u have their *next* field pointing to *some* other nodes.

In [LAS00], the authors report several successful applications of TVLA in various domains, such as the analysis of singly or doubly linked lists, sorting programs (bubble sort, for instance, could be verified on singly linked lists [LARSW00]), mobile ambients and multi-threaded Java programs [Yah01].

We believe TVLA could be of some interest in order to improve the abstraction of JAVA programs, because: (1) it provides a *parametric* framework for static analysis, (2) when used to perform static analysis, it yields to more precise results than the methods of Chase *et al.* for instance and (3) it is highly modular, in that sense that one can choose the predicates one is interested in. The results of TVLA could be used to extract more precise abstract models (of programs using intricate data structures, such as lists a.s.o.), or to improve the static analysis of a program in order to reduce its model even more.

4.4.4 On various compiler optimisation techniques

Constant propagation Many techniques have been developed by the compiler community, in order mainly to improve the code produced by the compiler. For instance, *constant propagation* [WZ91], that tries to find out which symbols of the program remain constant. This information could also be useful to us when we abstract a JAVA program into CBP, because it is always interesting to reduce the number of global and local variables of a system.

Escape analysis Another compiler optimisation is known as *escape analysis* [CGS+99, Bla99]. The problem of escape analysis amounts to determine whether an object can *escape* a method or a thread. More precisely, if an object $\mathcal{O}$ has been created by a method $m()$, is it possible that this object is ever accessed (after the completion of $m()$) by some code outside of $m()$? Or, if $\mathcal{O}$ has been created by a thread T, can other threads ever access it? In [CGS+99], the authors explain how to compute a *connection graph* that allows to answer these questions in many cases (the problem, is, of course, undecidable in general). Such an information could be used in our case in order to suppress the synchronisations on the objects that are *not thread-escaping*. As a matter of fact, these objects are *local* to the thread.

4.5 Discussion

In this chapter, we have quickly surveyed several works that seem useful to reduce the size of a model extracted from a JAVA program. Some of these works were developed for this purpose (it is the case of [Cor00] and [Sto00]). The others come mainly from the compiler community, and aim at optimising the performances of the compilers by acquiring a better knowledge of the structure of the heap at run time. They can non the less be useful to our purpose.

Chapter 5

Conclusion and Future Works

In this work, we have presented the Concurrent Boolean Programs (CBP). A CBP is a multi-threaded program that manipulates variables ranging over a boolean domain. It allows one to use communication constructs such as the *broadcast*, the *rendez-vous* and the *asynchronous rendez-vous*. Function calls however are not supported, and this disallows infinite recursion. In Chapter 2, we have given the grammar and the operational semantics of the CBP. In Chapter 3, we have shown how to translate a CBP into a *family of Global Machines*, which are well-suited for verification. We have also sketched the process of *predicate abstraction*, to translate a JAVA program into a CBP. In Chapter 4 we have quickly surveyed several ideas that could be useful in order to *reduce* the size of the models resulting from this process of translation.

Our final goal is the fully automatic verification of JAVA programs. The introduction of the CBP was an important step toward it, but several problems remain to be addressed. We more particularly think about the *abstraction* of JAVA programs. Concerning this last point, we now present some questions we believe are worth looking into:

1. *How to cope with the identities of the threads?* Up to now, we have only considered systems in which the identities of the threads could forgotten during the abstraction (all the threads were seen as symmetrical). In many cases, however this is not possible anymore. In the case of the Bakery Algorithm (Algorithm 1 and Appendix A), we have seen that it is necessary to take the thread's identities into account On the one hand, a model of such a system that would completely wipe out the values of the tickets would be *too coarse* to analyse to prove the property of mutual exclusion. On the other hand, a model that would retain all the tickets' values would be unbounded. We already know that we can analyse this algorithm, and prove the desired property, if we retain the identities (the tickets) of *a bounded number of threads* only, and abstract all the others into a set of symmetrical and anonymous threads (this idea is similar to the invisible invariants that Pnueli and his team have studied in [APR+01, PRZ01, KPSZ02]). Can we automatise this idea? As far as we know, this still remains to be explored.

2. *How to cope with possibly unbounded data structures?* In the examples we have considered so far, the system used really simple data structures, like integers or arrays of integers, and so forth. In real-life systems however, this is not sufficient anymore. One can easily imagine a multi-threaded web server in which a main thread stores the incoming requests in a (possibly unbounded) list before passing them to one thread picked from a (possibly unbounded) set of threads. The question that arises is 'How can we abstract the content of the list?'. In general, this question seems hardly answered without a good knowledge of the heap structure at run-time (at least, the structure of the list). Sagiv's works could help us to find a set of predicates that describes this structure, and we could use this set to obtain a precise abstraction of the program. As far as we know, this remains to be investigated.

3. *Can we go further in the reduction of the size of the model?* More particularly, is it worth combining the various static analysis we have presented in Chapter 4. If yes, how?

4. *How can we cope with unbounded recursion?*

To answer these questions, and many others, we need to implement an abstractor of JAVA program, and experiment with it. As far as we know, the only such implementation is to be found in JAVA Path Finder [BHPV00]. However, this software is not publicly available and poorly documented in the literature. Note that Bandera [CDH+00, HD01] uses *domain abstraction*, which by the way is not automatic (the designer has to specify the connection between the concrete and the abstract domain). This implementation effort is going to be a part of our future research.

Appendix A

A case study: The Bakery algorithm

A.1 The Bakery Algorithm in JAVA

```
public class bakalgo
{
   public static void main (String argv[]) {

      if (argv.length == 0) {
         System.out.println("Please specify the number of threads !") ;
         return ;
      }

      Integer i = new Integer(argv[0]) ;
      int numthreads = i.intValue() ;

      System.out.println("Creating "+numthreads+ " threads") ;

      ticket t[] = new ticket[numthreads];
      Thread T[] = new Thread[numthreads];

      for (int j=0; j<numthreads; ++j) {
         t[j] = new ticket(0) ;
      }

      bakery B = new bakery(numthreads, t) ;

      for (int j=0; j<numthreads; ++j) {
         T[j] = new myThread(B, t[j], j) ;
         T[j].start() ;
      }
   }
}
```

```
public class ticket {
   private int value = 0 ;
   boolean choosing= false ;

   public ticket(int i) {
      value = i ;
   }

   /* Says whether the thread owning this ticket is choosing a number */
   public boolean isChoosing() {
      return choosing ;
   }

   /* Returns the value of the ticket but blocks if the thread
      is busy choosing a value */
   public synchronized int getChooseValue() {
     while (choosing) {
        try {
           wait() ;
        } catch (InterruptedException e) {
           System.out.println("Ticket interrupted !") ;
        }
      }
      return value ;
   }

   /* Set the "choosing"-status of this ticket.  Notifies the pending
      threads if any.  */
   public synchronized void setChoosing(boolean c) {
      choosing = c ;
      if (c == false)
      notifyAll() ;
   }

   /* Sets the value of the ticket */
   public  void setValue(int i) {
      value = i ;
   }

   /* Return the value of the ticket */
   public  int getValue() {
      return value ;
   }

   /* Once out of the bakery, we tear the ticket and go back home !  */
   public  void tear() {
      value = 0 ;
   }
}
```

```
public class myThread extends Thread
{
   private bakery theBakery ;
   private ticket myTicket ;
   private int myId ;
   private java.util.Random r ;

   public myThread(bakery b, ticket t, int i) {
      theBakery = b ;
      myTicket = t ;
      myId = i ;
      r = new java.util.Random() ;
   }

   public void run() {
      int ti ;
      /* Some intial treatment */
      System.out.println("Thread "+myId+ ":  pretending to work...") ;
      try {
         sleep(r.nextInt(1000)+1) ;
      }
      catch(InterruptedException e) {
         System.out.println("Thread "+ myId +" interrupted") ;
      }
      System.out.println("Thread "+myId+ ":  I'm done !") ;

      ti = theBakery.getTicket(myTicket) ;

      System.out.println("Thread "+myId+ ":  got ticket "+ ti) ;

      /* Now let's try to buy this bread !  */

      theBakery.getTurn(myId, ti) ;

      /* -- This is the critical section --*/
      System.out.println("Thread "+myId+ ":  going in with ticket "+ ti) ;
      try {
         /* Some treatment */
         sleep(r.nextInt(1000)+1) ;
      }
      catch(InterruptedException e) {
         System.out.println("Thread "+ myId +" interrupted") ;
      }
      myTicket.tear() ;
      System.out.println("Thread "+myId+ ":  going out") ;
      theBakery.getOut() ;
      /* -- End of the critical section -- */
   }
}
```

```
public class bakery
{
   private ticket T[] ;
   private int slots ;

   public bakery(int s, ticket t[]) { slots = s ;   T = t ; }

   public int getTicket(ticket t) {
      int max = 0, v ;
      t.setChoosing(true) ;
      /* What we want is the "next number":
         the maximum of all the previous tickets + 1 */
      for (int j=0; j<slots; ++j) {
         v = T[j].getValue() ;
         if (v > max)  max = v ;
      }
      /* Here, some other thread could execute the same function and get
         the same ticket number (while we spend some time choosing a bread)*/
      try {
         Thread.sleep(1) ;
      }
      catch(InterruptedException e) {
         System.out.println("interrupted") ;
      }
      t.setValue(max+1) ;
      t.setChoosing(false) ;
      return max+1 ;
   }

   public void getTurn(int id, int ticket) {
      int turn ;
      for (int j = slots-1; j>=0; --j ) {
         if (j != id) {
            turn = T[j].getChooseValue() ;
            /* turn == zero means thread j has already gone out of the bakery */
            if (turn != 0) {
               /* If j is more prioritary than us (because it has a smaller number on
                  its ticket, or because it has the same number but a lower id)...  */
               if ((turn < ticket) ||((turn == ticket) && (j<id))) {
                  synchronized(this) {
                     try {/*  ..  we wait until it's out of the bakery.*/
                        while (T[j].getValue() != 0) {
                           wait() ;
                        }
                     } catch(InterruptedException e) {
                           System.out.println("Bakery interrupted") ;
                     }
                  }
               }
            }
         }
      }
   }

   public synchronized void getOut() {  notifyAll() ; }
}
```

A.2 The Bakery Algorithm in CBP

```
vars: Ieq0 Jeq0 IeqJ IgJ JgI IChoose JChoose ;
locks: BakeryLlock ILock JLlock ;
messages: BakeryMsg IMsg JMsg ;
threads: Ti Tj Env main ;

main { vars: ;
   Ieq0, Jeq0, IeqJ, IgJ, JgI:= true, true, true, false, false ;
   IChoose := false ;
   JChoose := false ;
   start(Ti) ;
   start(Tj) ;
   while(*) {
      start(Env) ;
   }
}

Env { vars : ;

   while(true)
   {
      if(*) {
         skip ;
      }
      while(*) {
         skip ;
      }

      skip ;

      while(*) {
         if(*) {
            if(*) {
               if(*) {
                  lock(BakeryLock) ;
                  while(*) {
                     sleep(BakeryLock, BakeryMsg) ;
                  }
                  unlock(BakeryLock) ;

               }
            }
         }
      }
      [criticalEnv] skip ;
      lock(BakeryLock) ;
      wakeupall(BakeryMsg) ;
      unlock(BakeryLock) ;
   }
}
```

```
Ti { vars :  ;
   while(true)
   {
      lock(ILock) ;
      IChoose := true ;
      unlock(ILock) ;
      choice { // if the other thread has a zero ticket, we have a higher ticket
         // otherwise, we might have a larger or the same ticket
         !Jeq0 :  IeqJ, Ieq0, JgI, IgJ := true, false, false, false ; // same ticket
         !Jeq0 :IgJ, Ieq0, JgI, IeqJ := true, false, false, false ; // larger ticket
         Jeq0 :  IgJ, Ieq0, JgI, IeqJ := true, false, false, false ; // larger ticket

      }

      lock(ILock) ;
      IChoose := false ;
      wakeupall(IMsg) ;
      unlock(ILock) ;

      lock(JLock) ;
      while (JChoose) {
         sleep(JLock, JMsg) ;
      }
      unlock(JLock) ;

      if(!Jeq0) { // if j == 0, then Tj is not in the doorway
         if (IgJ)
         {
            lock(BakeryLock) ;
            while( !  Jeq0) {
               sleep(BakeryLock, BakeryMsg) ;
            }
            unlock(BakeryLock) ;
         }
         else {
            if(IeqJ) { //I has a lower id than J -> it goes first
               skip ;
            }
         }
      }

      [criticalI]skip ;

      choice {
         Jeq0 :  Ieq0, IeqJ, IgJ, JgI := true, true, false, false ;
         !Jeq0 :  Ieq0, IeqJ, IgJ, JgI := true, false, false, true ;
      }

      lock(BakeryLock) ;
         wakeupall(BakeryMsg) ;
      unlock(BakeryLock) ;
   }
}
```

```
Tj { vars :  ;
   while(true)
   {
      lock(JLock) ;
      JChoose := true ;
      unlock(JLock) ;
      choice { // if the other thread has a zero ticket, we have a higher ticket
         // otherwise, we might have a larger or the same ticket
         !Ieq0 :  IeqJ, Jeq0, IgJ, JgI := true, false, false, false ; // same ticket
         !Ieq0 :JgI, Jeq0, IgJ, IeqJ := true, false, false, false ; // larger ticket
         Ieq0 :  JgI, Jeq0, IgJ, IeqJ := true, false, false, false ; // larger ticket

      }

      lock(JLock) ;
      JChoose := false ;
      wakeupall(JMsg) ;
      unlock(JLock) ;

      lock(ILock) ; // We look for the other's ticket -> it must not be choosing
      while (IChoose) {
         sleep(ILock, IMsg) ;
      }
      unlock(ILock) ;

      if(!Ieq0) { // if i == 0, then Ti is not in the doorway
         if (JgI)
         {
            lock(BakeryLock) ;
            while( !Ieq0) {
               sleep(BakeryLock, BakeryMsg) ;
            }
            unlock(BakeryLock) ;
         }
         else {
            if(IeqJ) { //J has a larger id than I -> it has to wait
               lock(BakeryLock) ;
               while( !Jeq0) {
                  sleep(BakeryLock, BakeryMsg) ;
               }
               unlock(BakeryLock) ;
            }
         }
      }

      [criticalJ]skip ;

      choice {
         Ieq0 :  Jeq0, IeqJ, JgI, IgJ := true, true, false, false ;
         !Ieq0 :  Jeq0, IeqJ, JgI, IgJ := true, false, false, true ;
      }

      lock(BakeryLock) ;
         wakeupall(BakeryMsg) ;
      unlock(BakeryLock) ;
   }
}
```

Appendix B

Proof of Lemma 3.2

We are now about to exhibit a proof by induction on the length of e.

Basis case: If the length of e is 1, then let s be the initial and only state in e. It is easy to see that there is a state s' of G_B such that $\Phi(e) = Live(s')$.

Induction step: Let us suppose the lemma holds for every run of length $\leq i$. Let $e = e' \cdot S \cdot S'$ be a run of length $i+1$ with $S = \langle \Gamma, \Lambda, \{s_1, \ldots, s_n\} \rangle$ and $S' = \langle \Gamma', \Lambda', \{s'_1, \ldots, s'_{n'}\} \rangle$. Furthermore, let $\Phi(S) = S_G$ with $S_G = \langle \Gamma \cup f(\Lambda), l_1, \ldots, l_m \rangle$. We are now going to show that there is a successor S'_G of S_G such that $\Phi(S') = S'_G$.

Depending on S (the i-th state of the CBP in e), we have to consider different cases:

(a) There exists $1 \leq i \leq n$ such that $s_i = \langle [\ell] u_1, \ldots, u_k := v_1, \ldots, v_k; [\ell'] I' \cdot S'', \rho \rangle$. For the sake of clarity we will consider only the cases where the set of source variables and the set of destination variables never contain both local and global variables. Remark that the literals `true` and `false` are assimilated to local variables (in these two cases, ρ is the identity relation).

global→global In this first case, all the variables are global. It follows that (i) $s'_i = \langle [\ell'] I' \cdot S'', \rho \rangle$; (ii) $n = n'$, $\Gamma' = \Gamma[u_1 = \Gamma(v_1), \ldots, u_k = \Gamma(v_k)]$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle \rho, \ell \rangle$. Let us compute its successor. Thanks to the translation rule 2a, we know that there exists a transition labelled by $a_1 \wedge \ldots \wedge a_k \wedge u'_1 = a_1 \wedge \ldots \wedge u'_k = a_k$, and going from l_k to $l'_k = \langle \rho, \ell' \rangle$ (where $\forall 1 \leq j \leq k \,:\, a_j = \Gamma(v_j)$). It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.

local→global In the second case, the source variables are local, and the target variables are global. It follows that (i) $s'_i = \langle [\ell'] I' \cdot S'', \rho \rangle$; (ii) $n = n'$, $\Gamma' = \Gamma[u_1 = \rho(v_1), \ldots u_k = \rho(v_k)]$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle \rho, \ell \rangle$. Let us compute its successor. According to rule 2a of section 3.2.3, it is exactly $l'_k = \langle \rho, \ell' \rangle$. Thanks to the semantics of Global/Local Machines, the valuation of the global variables becomes $\Gamma[u_1 = \rho(v_1), \ldots u_k = \rho(v_k)] \cup f(\Lambda)$.

It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$, and in which the valuation of the global variables has changed as stated before.

global→local In this third case, the source variables are global and the target variables are local. It follows that (i) $s'_i = \langle [\ell'] I' \cdot S'', \rho[u_1 = \Gamma(v_1), \ldots, u_k = \Gamma(v_k)] \rangle$; (ii) $n = n'$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle \rho, \ell \rangle$. Let us compute its successor. Thanks to the translation rule 2a, we know that there exists a transition labelled by $a_1 \wedge \ldots \wedge a_k$, and going from l_k to $l'_k = \langle \rho[u_1 = a_1, \ldots, u_k = a_k], \ell' \rangle$ (where $\forall 1 \leq j \leq k \,:\, a_j = \Gamma(v_j)$). It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.

local→local In this last case, the source variables and target variables are local. It follows that (i) $s'_i = \langle[\ell']I' \cdot S'', \rho[u_1 = \rho(v_1), \ldots u_k = \rho(v_k)]\rangle$; (ii) $n = n'$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$ and (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2a of section 3.2.3, it is exactly $l'_k = \langle\rho[u_1 = \rho(v_1), \ldots u_k = \rho(v_k)], \ell'\rangle$.

It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$, and in which the valuation of the global variables has changed.

(b) There exists $1 \leq i \leq n$ such that (i) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (ii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$; (iii) $s_i = \langle[\ell]\texttt{if}\ (B)\ \texttt{then}\ \{[\ell_1]I_1 \cdot S_1\}\ \texttt{else}\ \{[\ell_2]I_2 \cdot S_2\} \cdot S, \rho\rangle$ and

- either $\Gamma \cup \rho \models B$ and $s'_i = \langle[\ell_1]I_1 \cdot S_1 \cdot S, \rho\rangle$. We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2b of section 3.2.3, it is exactly $l'_k = \langle\rho, \ell_1\rangle$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.
- Otherwise $\Gamma \cup \rho \not\models B$ and $s'_i = \langle[\ell_2]I_2 \cdot S_2 \cdot S, \rho\rangle$. We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2b of section 3.2.3, it is exactly $l'_k = \langle\rho, \ell_2\rangle$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.

(c) There exists $1 \leq i \leq n$ such that (i) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (ii) for all $1 \leq j \leq n\ (j \neq i)$ we have $s'_j = s_j$; (iii) $s_i = \langle[\ell]\texttt{choice}\{B_1{:}u_{1,1}, \ldots, u_{1,m_1}{:=}v_{1,1}, \ldots, v_{1,m_1}; \ldots B_k{:}u_{k,1}, \ldots, u_{k,m_k}{:=}v_{k,1}, \ldots, v_{k,m_k};\} \cdot [\ell']I' \cdot S, \rho\rangle$ (iv) there exists $1 \leq m \leq k$ such that $\Gamma \cup \rho \models B_m$. This last condition tells us that the guard is satisfied for the mth assignation which just have to deal with. The case of the assignation has already been discussed previously.

(d) There exists $1 \leq i \leq n$ such that (i) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (ii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$; (iii) $s_i = \langle[\ell]\texttt{while}\ (B)\ \texttt{do}\ \{[\ell_1]I_1 \cdot S_1\} \cdot [\ell_2]I_2 \cdot S_2, \rho\rangle$ and

- either $\Gamma \cup \rho \models B$ and $s'_i = \langle[\ell_1]I_1 \cdot S_1 \cdot S, \rho\rangle$. We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2d of section 3.2.3, it is exactly $l'_k = \langle\rho, \ell_1\rangle$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.
- Otherwise $\Gamma \cup \rho \not\models B$ and $s'_i = \langle[\ell_2]I_2 \cdot S_2 \cdot S, \rho\rangle$. We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2d of section 3.2.3, it is exactly $l'_k = \langle\rho, \ell_2\rangle$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.

(e) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle[\ell]\texttt{skip;}[\ell']I \cdot S, \rho\rangle$ and $s'_i = \langle[\ell']I \cdot S, \rho\rangle$ and; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2e of section 3.2.3, it is exactly $l'_k = \langle\rho, \ell'\rangle$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.

(f) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle[\ell]\texttt{goto}(\ell'); \cdot S, \rho\rangle$ and $s'_i = \langle[\ell']I \cdot S, \rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2f of section 3.2.3, it is exactly $l'_k = \langle\rho, \ell'\rangle$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$.

(g) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle[\ell]\texttt{lock}(\textbf{LockId}); [\ell']I \cdot S, \rho\rangle$ and $s'_i = \langle[\ell']I \cdot S, \rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda[\textbf{LockId} = \textit{locked}]$; (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle\rho, \ell\rangle$. Let us compute its successor. According to rule 2g of section 3.2.3, it is exactly $l'_k = \langle\rho, \ell'\rangle$. Moreover, **LockId** is put

to $\top$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$ and **LockId** is put to $\top$.

(h) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle [\ell]\texttt{unlock}(\mathbf{LockId}); [\ell']I \cdot S, \rho\rangle$ and $s'_i = \langle [\ell']I \cdot S, \rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda[\mathbf{LockId} = \mathit{unlocked}]$; (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle \rho, \ell\rangle$. Let us compute its successor. According to rule 2h of section 3.2.3, it is exactly $l'_k = \langle \rho, \ell'\rangle$. Moreover, **LockId** is put to $\perp$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$ and **LockId** is put to $\perp$.

(i) There exists $1 \leq i \leq n$ such that: (i) $s_i = \langle [\ell]\texttt{sleep}(\mathbf{msgId}, \mathbf{lockId})\ ; \cdot\ [\ell']I' \cdot S'', \rho\rangle$ and $s'_i = \langle [\mathbf{wait}_\ell]\mathit{wait}(\mathbf{msgId}, \mathbf{lockId}) \cdot [\ell']I' \cdot S'', \rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda[\mathbf{lockId} = \mathit{unlocked}]$; (iii) for all $1 \leq j \leq n(j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle \rho, \ell\rangle$. According to rule 2i (section 3.2.3), its successor is exactly $\langle \rho, \mathbf{wait}_\ell\rangle$. Clearly, $\Phi(S')$ is equivalent to $\Phi(S)$ in which we have replaced l_k by l'_k.

(j) In the case of the `wakeup`, there are two cases to take into account, whether there are or not sleeping threads awaiting to be woken up.

First case In the former case, there exist $1 \leq i < j \leq n$ such that (i):

$$\begin{cases} s_i = \langle [\mathbf{wait}_\ell]\mathit{wait}(\mathbf{msgId}, \mathbf{lockId}) \cdot [\ell']I' \cdot S'', \rho_i\rangle \\ s'_i = \langle [\mathbf{lock}_\ell]\texttt{lock}(\mathbf{lockId})\ ; \cdot\ [\ell']I' \cdot S'', \rho_i\rangle \\ s_j = \langle [\ell_j]\texttt{wakeup}(\mathbf{msgId})\ ; \cdot\ [\ell'_j]I'_j \cdot S''_j, \rho_j\rangle \\ s'_j = \langle [\ell'_j]I'_j \cdot S''_j, \rho_j\rangle \end{cases}$$

and (ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$, and (iii) for all $1 \leq m \leq n : (m \notin \{i, j\}$, we have $s'_m = s_m$.

We know that there exist two local machines $\mathcal{L}_{k_1}$ and $\mathcal{L}_{k_2}$ whose local states in S_G are $l_{k_1} = \langle \rho_i, \mathbf{wait}_\ell\rangle$ and $l_{k_2} = \langle \rho_j, \ell_j\rangle$ respectively. According to rule 2i (section 3.2.3), we know that the successors of these states are: $l'_{k_1} = \langle \rho_i, \mathbf{lock}_\ell\rangle$, for the former one and $l'_{k_2} = \langle \rho_j, \ell'_j\rangle$, for the latter one. It is now easy to see that $\Phi(S')$ is equal to $\Phi(S)$ in which we have replaces l_{k_1} and l_{k_2} by l'_{k_1} and l'_{k_2} respectively.

Second case In the latter case, there exists $1 \leq i \leq n$ such that (i) $s_i = \langle [\ell]\texttt{wakeup}(\mathbf{msgId})\ ; \cdot\ [\ell']I' \cdot S'', \rho\rangle$ and for every $j \neq i$, the program remaining to be executed in s_j does not begin by instruction *wait*; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$; (iii) for all $1 \leq j \leq n\ :\ (j \neq i)$ we have $s'_j = s_j$.

We know that there exists one local machine $\mathcal{L}_k$ whose local state in S_G is $l_k = \langle \rho, \ell\rangle$. According to rule 2j, and following the semantics of the Global Machines, the successor of l_k is $l'_k = \langle \theta, \ell'\rangle$. One can easily see that $\Phi(S')$ is the same as $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_l to l'_k.

(k) In the case of the `wakeupall`, we reorder and divide the set $\{s_1, \ldots s_n\}$ in three subsets $\Sigma_1 = \{s_1, \ldots, s_k\}$, $\Sigma_2 = \{s_{k+1}, \ldots, s_{n-1}\}$ and $\Sigma_3 = \{s_n\}$, such that: (i), for every $1 \leq i \leq k$: s_i is $\langle [\mathbf{wait}_{\ell_i}]\mathit{wait}(\mathbf{msgId}) \cdot [I'_i].S''_i, \rho_i\rangle$; (ii), for every $k < j \leq n$: the program S_j remaining to be executed in s_j does *not* begin by $\mathit{wait}()$ and (iii) s_n is the state of the local machine doing the `wakeupall`. More precisely, $s_n = \langle [\ell_n]\texttt{wakeupall}(\mathbf{msgID})\ ; \cdot\ [\ell'_n]I'_n \cdot S''_n, \rho_n\rangle$.

Then (according to axiom 2.14), we know that S' is such that: $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$, for every $k < j \leq n - 1 : s'_j = s_j$, for every $1 \leq i \leq k\ :\ s_i = \langle [\mathbf{lock}_{\ell_i}]\mathit{wait}(\mathbf{msgId}) \cdot [I'_i].S''_i, \rho_i\rangle$, and finally: $s'_n = \langle [\ell'_n]I'_n \cdot S''_n, \rho_n\rangle$.

We know that there exist k local machines in S_G begin respectively in the states $l_i = \langle \rho_i, \mathbf{wait}_{\ell_i}\rangle$, for all $1 \leq i \leq k$. There also exists one local machine in S_G which is in the state $l_n = \langle \rho_n, \ell_n\rangle$. According to rule 2k, and thanks to the semantics of the Global/Local machines, all these local machines can synchronise on **msgId**. The successor of the machine that is in l_n will be $l'_n = \langle \rho_n, \ell'_n\rangle$, and the successors of all the other ones will be $l'_i = \langle \rho_i, \ell'_i\rangle$ respectively. One can now easily see that, if we replace l_n by l'_n and all the l_i by their corresponding l'_i in S_G, one obtains a global state which is exactly $\Phi(G')$.

(l) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle [\ell]\texttt{start}(\textbf{threadTypeName}\pi_1, \ldots, \pi_m, [])\ ;\ell' I \cdot S, \rho\rangle$ and $s'_i = \langle [\ell'] I \cdot S, \rho\rangle$ and, (ii) $n' = n+1$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda$, and (iii) for all $1 \leq j \leq n (j \neq i)$ we have $s'_j = s_j$. Moreover, if **threadTypeName** is the identifier of thread type $T_j = \langle \mathcal{V}_j, S_j\rangle$, then the state of the $n+1$th thread (the newly created one) is $s_{n+1} = \langle [\ell]_{n+1} I_{n+1} \cdot S'_{n+1}, \rho_{n+1}\rangle$, where:

- $S_j \equiv [\ell]_{n+1} I_{n+1} \cdot S'_{n+1}$;
- ρ_{n+1} is such that: $\forall 1 \leq j \leq m : \rho_{n+1}(v_j) = \pi_j$.

We know that there exists a local machine $\mathcal{L}_k$ in S_G that is in the local state $\langle \rho, \ell\rangle$. By hypothesis of the lemma, we also know that there exists in S_G at least one local machine $\mathcal{L}^*$, whose type corresponds to thread type T_j, and whose state is the *idle state*: $\langle \theta_\top, \textbf{begin}\rangle$.

Thanks to the construction rules explained in section 3.2.3, and thanks to the semantics of Global and Local Machines, $\mathcal{L}_k$ and $\mathcal{L}^*$ can synchronise on $\textbf{threadTypeName}\pi_1 \ldots \pi_m$, and their resulting states will be $\langle \rho, \ell'\rangle$ and $\langle \rho_{n+1}, \ell_{n+1}\rangle$, respectively.

It should now be clear that the resulting global state S'_G is exactly $\Phi(S')$.

(m & n) There exists $1 \leq i < j \leq n$ such that:

$$(i) \begin{cases} s_i = \langle [\ell_i]\texttt{rendezvous}(\textbf{msgId}, u_1, \ldots, u_n)\ ; \cdot [\ell'_i] I'_i \cdot S''_i, \rho_i\rangle \\ s'_i = \langle [\ell'_i] I'_i \cdot S''_i, \rho_i\rangle \\ s_j = \langle [\ell_j]\texttt{accept}(\textbf{msgId}, v_1, \ldots, v_n)\ ; \cdot [\ell'_j] I'_j \cdot S''_j, \rho_j\rangle \\ s'_j = \langle [\ell'_j] I'_j \cdot S''_j, \rho_j[v_1 = u_1, \ldots v_n = u_n]\rangle \end{cases}$$

(ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda$, and (iii) for all $1 \leq m \leq n : m \neq i \wedge m \neq j$, we have $s'_m = s_m$.

We know that there exist two local machines $\mathcal{L}_{k_1}$ and $\mathcal{L}_{k_2}$ whose local states in S_G are $l_{k_1} = \langle \rho_i, \ell_i\rangle$ and $l_{k_2} = \langle \rho_j, \ell_j\rangle$ respectively. According to rule 2i (section 3.2.3), and thanks to the semantics of Global and Local machines, these two machines can synchronise on the message $\textbf{msgId}\rho(u_1) \ldots \rho(u_n)$. Their respective successors are thus: $l'_{k_1} = \langle \rho_i, \ell'_i\rangle$ and $l'_{k_2} = \langle \rho_j[v_1 = u_1, \ldots v_n = u_n], \ell'_j\rangle$. According to this, the global state S'_G, successor of S_G, is exactly $\Phi(S')$.

Finally, by induction hypothesis we know that there is a run $\sigma \cdot S_G$ of G_B such that $\Phi(e' \cdot S) = \mathit{Live}(\sigma)$.

As we have just shown that S'_G is always equal to $\Phi(S')$, we can conclude that there is a run $\sigma' = \sigma \cdot S_G \cdot S'_G$ such that $\Phi(e) = \mathit{Live}(\sigma')$.

■

Bibliography

[ADG$^+$02] Pierluigi Ammirati, Giorgio Delzanno, Pierre Ganty, Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. BABYLON: An integrated tool for the specification and verification of parametrized systems. In *Proceedings of SAVE, 2nd workshop on Specification, Analysis and Validation for Emerging technologies, Copenhagen, Denmark*, 2002.

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Oledrog. *Verification of Sequential and Concurrent Programs, second edition.* Springer-Verlag, 1997.

[APR$^+$01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized Verification With Automatically Computed Inductive Assertions. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 221–234. Springer, 2001.

[Bar96] J. Barnes. *Programming in ADA 95.* Addison-Wesley, 1996.

[BCR01a] T. Ball, S. Chaki, and S.K. Rajamani. Parameterized Verification of Multithreaded Software Libraries. In *Proceedings of the Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 158–173. Springer, 2001.

[BCR01b] Thomas Ball, Sagar Chaki, and Sriram K. Rajamani. Parameterized verification of multithreaded software libraries. *Lecture Notes in Computer Science*, 2031:158–173, 2001.

[Ber85] G. Berthelot. Checking properties of nets using transformations. In *Proceedings of Advances in Petri Nets (APN'85)*, LNCS, pages 19–40. Springer-Verlag, 1985.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. *ACM SIGPLAN Notices*, 34(10):35–46, 1999.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder second generation of a java model checker. In *Proceedings of the Post-CAV Workshop on Advances in Verification 2000 (WAVE 2000)*, 2000.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.

[BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of The 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 203–213. ACM Press, 2001.

[BMR02] T. Ball, T. Millstein, and S.K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, 2002.

[BPR02] T. Ball, A. Podelski, and S.K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 158–172. Springer, 2002.

[BR00] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, 2000.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[Cor98] James C. Corbett. Constructing compact models of concurrent java programs. In *International Symposium on Software Testing and Analysis*, pages 1–10, 1998.

[Cor00] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, 2000.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the S1GPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[DR00] G. Delzanno and J.-F. Raskin. Symbolic Representation of Upward Closed Sets. In *Proceeding of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 426–440. Springer, 2000.

[DRVB01] G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 298–310. Springer, 2001.

[DRVB02a] G. Delzanno, J.-F. Raskin, and L. Van Begin. Covering Sharing Trees: Efficient Data Structurs for the Automated Verification of Parametrized Systems. *accepted for publication in Software Tools for Technology Transfer Manuscript*, 2002.

[DRVB02b] G. Delzanno, J-F. Raskin, and L. Van Begin. Towards the Automated Verification of Multithreaded Java Programs. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

[Fla97] David Flanagan. *Java in a Nutshell, Second Edition.* O'Reilly & Associates, inc., 1997.

[Gee02] Gilles Geeraerts. A comparison of various backward analsysers for parameterized concurrent systems. Master's thesis, Université Libre de Bruxelles, 2002.

[GH97] S. Graf and H.Saidi. Construction of abstract state graphs with pvs. In *Proceeding of the 9th Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[Gra97] M. Grand. *Java Language Reference.* o'reilly, 1997.

[Gri81] David Gries. *The Science of Programming.* Springer Verlag, 1981.

[GS92] S. M. German and A. P. Sistla. Reasoning about Systems with Many Processes. *Journal of ACM*, 39(3):675–735, 1992.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[HD01] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent Java software. *Lecture Notes in Computer Science*, 2154:39–??, 2001.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceeding of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 58–70. ACM Press, 2002.

[KPSZ02] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network Invariants in Action. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002)*, volume 2421 of *LNCS*, pages 101–115. Springer, 2002.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Comunications of the ACM*, 17(8):453–455, 1974.

[LARSW00] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, pages 26–38, 2000.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, pages 280–301, 2000.

[LC98] Leslie Lamport and Ernie Cohen. Reductions in tla. In *Proceedings of CONCUR'98 Concurrency Theory*, number 1466 in Lecture Notes in Computer Science, pages 317–331, 1998.

[Lip75] R. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[LS89] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report TR89-1005, 1989.

[Lyn97] Nancy Lynch. *Distributed algorithms*. Morgan Kaufmann, Inc, 1997.

[PRZ01] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *LNCS*, pages 82–97. Springer, 2001.

[Ram99] G. Ramalingam. Context-sensitive synchronisation-sensitive analysis is undecidable. Technical Report RC21-493, IBM T.J. Watson Research, 1999.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[Sto00] Scott D. Stoller. Model-checking multi-threaded distributed java programs. In *SPIN*, pages 224–244, 2000.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transaction on Programming Languages and Systems*, 13(2):181–210, april 1991.

[Yah01] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, 2001.