Concurrent Boolean Programs*

Gilles Geeraerts
gigeerae@ulb.ac.be

Laurent Van Begin
lvbegin@ulb.ac.be

Université Libre de Bruxelles – Départment d'Informatique
CP 212, Boulevard du triomphe, B1050 Bruxelles

Abstract

In this work, we present a formal language called Concurrent Boolean Programs (CBP for short). The major aim of this language is to describe abstractions of programs written in various widely-used multi-threaded programming languages (in particular: Java, C with POSIX threads, and Ada). Therefore, the language allows one to make use of different communications primitives such as *broadcasts*, *sleeps* and *(asynchronous) rendez-vous*. It can be seen as a multi-threaded extension of Rajamani's Boolean Programs [BR00].

Contents

*Technical Report ULB ???

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 $p \equiv \mathtt{x} \geq 0$, 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.

```java
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: A simple sketch of concurrent program using JAVA's concurrency support.

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.).

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 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 *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 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 is given at Figures 5, 4, 3 and 6.

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: $number(j) = 0 \vee (number(i), i) < (number(j), j)$.

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

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

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

Figure 2: A simple `wait` - `notify` example.

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 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 it has not the highest priority among the threads waiting to be served (*cfr.* 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 thread 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:

$choosing(i) \leftarrow 1$;
$number(i) \leftarrow 1 + \max_{j \neq i} number(j)$;
$choosing(i) \leftarrow 0$;
foreach $j \neq i$ **do**
    waitfor $choosing(j) = 0$;
    waitfor $number(j) = 0 \vee (number(i), i) < (number(j), j)$;
Critical section ;
$number(i) \leftarrow 0$

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.

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

[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/Golbal 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]).

```java
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() throws InterruptedException{
     while (choosing) {
        wait() ;
      }
      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 ;
   }
}
```

Figure 3: Lamport's Bakery algorithm in JAVA (cont'd): The `ticket` class.

```java
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 -- */
   }
}
```

Figure 4: Lamport's Bakery algorithm in Java (cont'd): The `myThread` class.

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

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

   public int getTicket(ticket t) throws InterruptedException {
      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 (while we spend some time choosing a bread)*/
      Thread.sleep(1) ;
      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 exited the bakery */
            if (turn != 0) {
           /* If j is more prioritary than us (because its ticket number is
            lower or because it has the same number but a lower id)...   */
               if ((turn < ticket) ||((turn == ticket) && (j<id))) {
                  synchronized(this) {
                     /*  ..  we wait until it's out of the bakery.*/
                        while (T[j].getValue() != 0) wait() ;
                  }
               }
            }
         }
      }
   }

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

Figure 5: Lamport's Bakery algorithm in Java (cont'd): The `bakery` class.

```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() ;
      }
   }
}
```

Figure 6: Lamport's Bakery algorithm in JAVA : The `main` function.

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, it has been shown in [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 Stoller in [Sto00] 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].

1.4 Plan of the work

Section 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*.

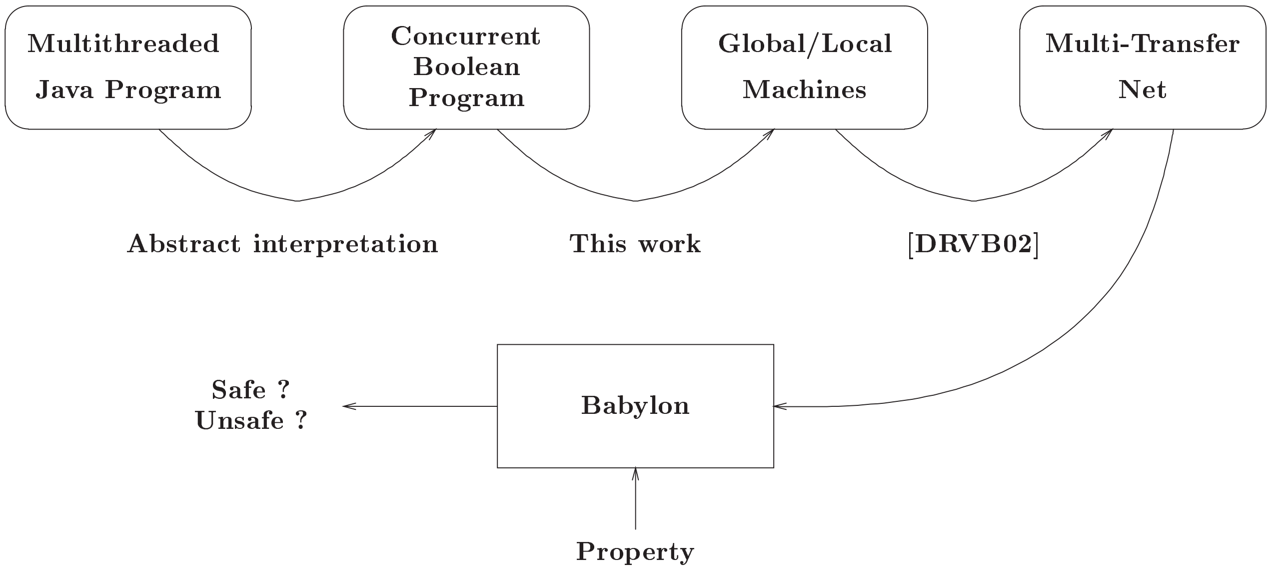

Figure 7: Our framework to verify safety properties on multi-threaded JAVA programs.

Section 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.

Section 4 draws some conclusions and tries to suggest some new paths of research.

1.5 Related works

In [CDH$^+$00], the authors describe BANDERA, a tool for model-checking JAVAprograms. This tool can extract *finite-state* models from JAVA source code. The extracted models can be expressed in the input languages of several classical model-checking tools such as SPIN [Hol97], SMV or Java Path Finder (see bellow).

In [HP00], the authors describe the first generation of a tool called JAVA PATH FINDER. This first version was simply a JAVA-to-Promela translator (Promela in the input language of SPIN). The extracted models are thus finite-state. The next generation of JAVA PATH FINDER is presented in [BHPV00]. This new version uses BANDERA's capabilities to *slice* models of JAVA programs with respect to variables involved in race-conditions. It can perform *predicate abstraction* , using the Stanford Validity Checker. [HP00]

2 The Concurrent Boolean Programs

In this chapter, we introduce the *language of CBPs*[3] which are the cornerstone of our verification framework, as we plan to translate JAVA programs into CBPs and CBPs into GMs.

[3]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...

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 8. 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 9. 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*). 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.

[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	→	LblCommands$^+$
[9]	LblCommands	→	**[label]** Command
[10]		→	Command
[11]	Command	→	Assignment `;`
[12]		→	`if` (Condition) { LblCommands } `else` { LblCommands }
[13]		→	`choice` { GuardedAssign$^+$ }
[14]		→	`while` (Condition) { LblCommands }
[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 8: 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 9: A simple Concurrent Boolean Program.

2.1.1 Well-Formed Concurrent Boolean Programs

We begin this subsection with a set of useful definitions:

Let *Locks*(S), *Global*(S), *Messages*(S) be respectively the set of locks, global variables, and messages declared in S. Let *NameThreads*(S) be the set of names of thread types declared in the preamble of S, and let *Threads*(S) be the set of threads whose code is specified in S. Furthermore, let *Labels*(S) be the set of labels present in program S (that is, for every ℓ in *Labels*(S), there exists a line in S beginning by $[\ell]$, and conversely). Finally, let *Local*(T) be the set of *local variables* declared for the thread type T, and *Body*(T) be the sequence of instructions ('LabeledCommands' in Figure 8) 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 8 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 *Labels*(S) ;

Declarations If a variable v is used in the body of thread type T, v must be either in *Global*(S), either in *Local*(T). Each lock that is used in S (in a `lock`, an `unlock` or a `sleep` instruction) must belong to *Locks*(S). Each thread in *NameThreads*(S) must also be in *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 *Messages*(S).

Thread start For each `start`$(T, \pi_1, \ldots, \pi_n)$ instruction in S, two conditions must hold: 1) $T \in$ *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: $\{$`main`$\} \in$ *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 *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 *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

$$\texttt{lock}(L)\ ;\texttt{if}(E)\ \texttt{then}\{\texttt{unlock}(L)\ ;\}\texttt{else}\{\texttt{skip}\ ;\}$$

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 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 8, 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 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.

$\square$

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

Definition 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 8.

$\square$

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

Example 5 For instance, the type of thread for the thread `t2` of Figure 9 is

$$\langle \underbrace{\{\texttt{l2}\}}_{\text{local variables}} , \underbrace{\texttt{skip ;} \cdot \texttt{accept(msg, l2) ;}}_{\text{sequence of instructions}} \rangle$$

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 8, hence from Definition 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 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 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 8) as follows:
  - $\Gamma(\texttt{true}) = \top$
  - $\Gamma(\texttt{false}) = \bot$
  - $\Gamma(!u) = \bot$ if $\Gamma(u) = \top$
  - $\Gamma(!u) = \top$ if $\Gamma(u) = \bot$
- $\Lambda : \mathcal{L} \mapsto \{\mathit{locked}, \mathit{unlocked}\}$ 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 5 Thread Instantiation.
A **thread instantiation** of $T_i = \langle \mathcal{V}_i, S_i \rangle$ $(1 \le i \le k)$ 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 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 1 to 9) are of the form $G \to G'$, which indicates that the transition from G to G' is legal.

   The inductive case is obtained through the sequence axiom (10) and the interleaving axiom (11). They are of the form $\frac{G_1 \to G_1'}{G_2 \to G_2'}$, which means: 'If the transition $G_1 \to G_1'$ is allowed, then so is $G_2 \to G_2'$'.

   Among these axioms, one is of particular interest: the axiom 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, L and T to denote a message, a lock and a type of thread respectively.

The axioms for $\texttt{rendezvous}(m)$, $\texttt{accept}(m)$ and $\texttt{start}(T)$ have been omitted. They can be deduced from the axioms 16 (for `accept` and `rendezvous`) and 15 (for `start`), with an empty set of parameters.

Assignment Rule

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

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

$$\langle \Gamma, \Lambda, \langle \texttt{if}\ (B)\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\}, \rho\rangle\rangle \rightarrow \begin{cases} \langle \Gamma, \Lambda, \langle S_1, \rho\rangle\rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \vDash B \\ \langle \Gamma, \Lambda, \langle S_2, \rho\rangle\rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \nvDash B \end{cases} \tag{2}$$

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

$$\begin{array}{c}\langle \Gamma, \Lambda, \langle \texttt{choice}\{B_1{:}v_{1,1}, \ldots, v_{1,n_1}{:=}u_{1,1}, \ldots, u_{1,n_1};\ \ldots\ B_m{:}v_{m,1}, \ldots, v_{m,n_m}{:=}u_{m,1}, \ldots, u_{m,n_m};\}, \rho\rangle\rangle \\ \rightarrow \\ \langle \Gamma[v_{i_1} = c_{i_1}, \ldots, v_{i_k} = c_{i_k}], \Lambda, \langle \varepsilon, \rho[v_{i_{k+1}} = c_{ik+1}, \ldots, v_{i_{m_j}} = c_{i_{m_j}}]\rangle\rangle\end{array} \tag{3}$$

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

$$\langle \Gamma, \Lambda, \langle \texttt{while}\ (B)\{S\}, \rho\rangle\rangle \rightarrow \begin{cases} \langle \Gamma, \Lambda, \langle S \cdot \texttt{while}\ (B)\{S\}, \rho\rangle\rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \vDash B \\ \langle \Gamma, \Lambda, \langle \varepsilon, \rho\rangle\rangle & \text{if } B \equiv * \text{ or if } \rho \cup \Gamma \nvDash B \end{cases} \tag{4}$$

Skip Rule

$$\langle \Gamma, \Lambda, \langle \texttt{skip};\ , \rho\rangle\rangle \rightarrow \langle \Gamma, \Lambda, \langle \varepsilon, \rho\rangle\rangle \tag{5}$$

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

Goto Rule

$$\langle \Gamma, \Lambda, \langle \texttt{goto}\ (\ell);\ , \rho\rangle\rangle \to \langle \Gamma, \Lambda, \langle \gamma(S,\ell), \rho\rangle\rangle \tag{6}$$

Where:

- The body of the thread is S ;
- $\gamma(S,\ell)$ is defined as shown in Figure 10. 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.

$$\langle \Gamma, \Lambda, \langle \texttt{lock}\ (L);\ , \rho\rangle\rangle \to \langle \Gamma, \Lambda[L = \textit{locked}], \langle \varepsilon, \rho\rangle\rangle \tag{7}$$

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

Unlock Rule

$$\langle \Gamma, \Lambda, \langle \texttt{unlock}\ (L);\ , \rho\rangle\rangle \to \langle \Gamma, \Lambda[L = \textit{unlocked}], \langle \varepsilon, \rho\rangle\rangle \tag{8}$$

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 12 to 14).

$$\begin{gathered}\langle \Gamma, \Lambda, \langle \texttt{sleep}\ (M, L), \rho\rangle\rangle \\ \rightarrow \\ \langle \Gamma, \Lambda[L = \mathit{unlocked}], \langle [\mathbf{wait}_\ell]\mathit{wait}(M, L), \rho\rangle\rangle\end{gathered} \tag{9}$$

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

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

$$\frac{\langle \Gamma, \Lambda, \langle S_1, \rho\rangle\rangle \rightarrow \langle \Gamma', \Lambda', \langle S_2, \rho'\rangle\rangle}{\langle \Gamma, \Lambda, \langle S_1 \cdot S, \rho\rangle\rangle \rightarrow \langle \Gamma', \Lambda', \langle S_2 \cdot S, \rho'\rangle\rangle} \tag{10}$$

Interleaving Rule

$$\frac{\langle \Gamma, \Lambda, \langle S_i, \rho_i\rangle\rangle \rightarrow \langle \Gamma', \Lambda', \langle T_i, \rho_i'\rangle\rangle}{\langle \Gamma, \Lambda, \Sigma\rangle \rightarrow \langle \Gamma', \Lambda', \Sigma'\rangle} \tag{11}$$

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.

$$\begin{gathered}\langle \Gamma, \Lambda, \{\langle \texttt{wakeup}\ (M) \cdot S_1, \rho_1\rangle, \langle [\mathbf{wait}_\ell]\mathit{wait}(M, L) \cdot S_2, \rho_2\rangle\} \cup \Sigma\rangle \\ \rightarrow \\ \langle \Gamma, \Lambda, \{\langle S_1, \rho_1\rangle, \langle [\mathbf{lock}_\ell]\texttt{lock}\ (L) \cdot S_2, \rho_2\rangle\} \cup \Sigma\rangle\end{gathered} \tag{12}$$

$$\langle \Gamma, \Lambda, \langle \texttt{wakeup}\ (M) \cdot S_1, \rho_1\rangle \cup \Sigma\rangle \rightarrow \langle \Gamma, \Lambda, \langle S_1, \rho_1\rangle \cup \Sigma\rangle \tag{13}$$

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.

$$\langle \Gamma, \Lambda, \langle \texttt{wakeupall}\ (M) \cdot S, \rho \rangle \cup \Sigma_1 \cup \Sigma_2 \rangle \to \langle \Gamma, \Lambda, \langle S, \rho \rangle \cup \Sigma'_1 \cup \Sigma'_2 \rangle \tag{14}$$

where

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

$$\langle \Gamma, \Lambda, \langle \texttt{start}\ (T_i, \pi_1, \ldots, \pi_{|\mathcal{V}_i|}); \cdot S', \rho \rangle \cup \Sigma \rangle \to \langle \Gamma, \Lambda, \{\langle S', \rho \rangle, \langle S_i, \rho' \rangle\} \cup \Sigma \rangle \tag{15}$$

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

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*[4], 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 11.

$$\begin{array}{c} \langle \Gamma, \Lambda, \langle \texttt{accept}\ (M, v_1, \ldots, v_n) \cdot S, \rho \rangle, \langle \texttt{rendezvous}\ (M, u_1, \ldots, u_n) \cdot S', \rho' \rangle \cup \Sigma \rangle \\ \to \\ \langle \Gamma, \Lambda, \{\langle S, \rho[v_1 = \rho'(u_1), \ldots, v_n = \rho'(u_n)] \rangle, \langle S', \rho' \rangle\} \cup \Sigma \rangle \end{array} \tag{16}$$

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 :

[4]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 11: 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`.

Definition 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: $\forall i :\ 1 \leq i \leq n-1 :\ G_i \rightarrow G_{i+1}$ (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 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. □

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 7). 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.[5]

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, fully automatisable [BHPV00, BMMR01] even when predicates relate to variable references [BMR02], is used to construct

[5]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.

```java
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 ;
    }
```

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

Figure 12: A simple example of JAVA class

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, 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 13 and let us consider a program in which several such threads execute concurrently. These threads manipulate a shared `point` object described in Figure 12. 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.

```
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 13: JAVA thread classes manipulating a shared `Point` object

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 practise 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 14 shows the two CBP threads corresponding to the JAVA threads of Figure 13. □

```
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 14: The `inc` and `dec` CBP threads corresponding to the `Inc` and `Dec` JAVA threads of Figure 13

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 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.

Now that we have shown how to translate a JAVA program into a CBP[6], let's show (according to our *verification framework* of Figure 7) how to translate a CBP into a Global/Local Machine.

3.2 From CBP to Global/Local Machines

Let us now introduce a new step toward the verification of properties on CBP(in Figure 7, 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

[6] 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.

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 8 Boolean Formulas

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 9 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. 8), and $\ell \in \Sigma$):

- *Internal action:* $\ell : \varphi$;
- *Rendez-vous:* $\ell! : \varphi$ (sending), and $\ell?$ (reception);
- *Asynchronous Rendez-vous:* $\ell\!\uparrow: \varphi$ (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 10 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 10), in order to handle the special labels $[\mathbf{wait}_\ell]$ and $[\mathbf{lock}_\ell]$, that we have introduced along the axioms 9 and 7 of 2.2.2. The definition should be self-explanatory.

Definition 11 $\widetilde{\gamma}$: The Extension of γ.

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

□

$\widetilde{\gamma}$ 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 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 1 *If we have reached the local state $\langle S_j, \rho_j \rangle$ from $\langle [\ell']$*`goto`*$(\ell); S', \rho_j \rangle$, then, it immediately follows from the* `goto` *axiom (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 (10) we obtain $[\ell'']I \cdot S_3 \cdot [\ell]I \cdot S_2 \rightarrow_{10} [\ell]I \cdot S_2$.

In the second case, using the sequence axiom (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 (6) and the sequence axiom (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 15 should help the reader to get the picture...). S is such that either:

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

(2) $[\ell']\mathtt{if}(E_1)\{S_1\}\ldots\mathtt{elseif}(E_k)\{S_k\}$ 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 $S_{\ell'} = [\ell']\mathtt{while}(E)\{S\} \cdot S''$. Let's apply the `while` axiom (4):

$$S_{\ell'} \rightarrow_4 S \cdot [\ell']\mathtt{while}(E)\ \{S\} \cdot S''$$

and then by the sequence axiom (10):

$$S \cdot [\ell']\mathtt{while}(E)\ \{S\} \cdot S'' \rightarrow_{10} [\ell]I \cdot S_a \cdot \mathtt{while}(E)\ \{S\} \cdot S'' = S_\ell$$

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 $\widetilde{\gamma}$):

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

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

$$S_{\ell'} \rightarrow_2 S_i \cdot S''$$

and by the sequence axiom (10):

$$S_i \cdot S' \rightarrow_{10} [\ell]I \cdot S_a \cdot S'' = S_\ell$$

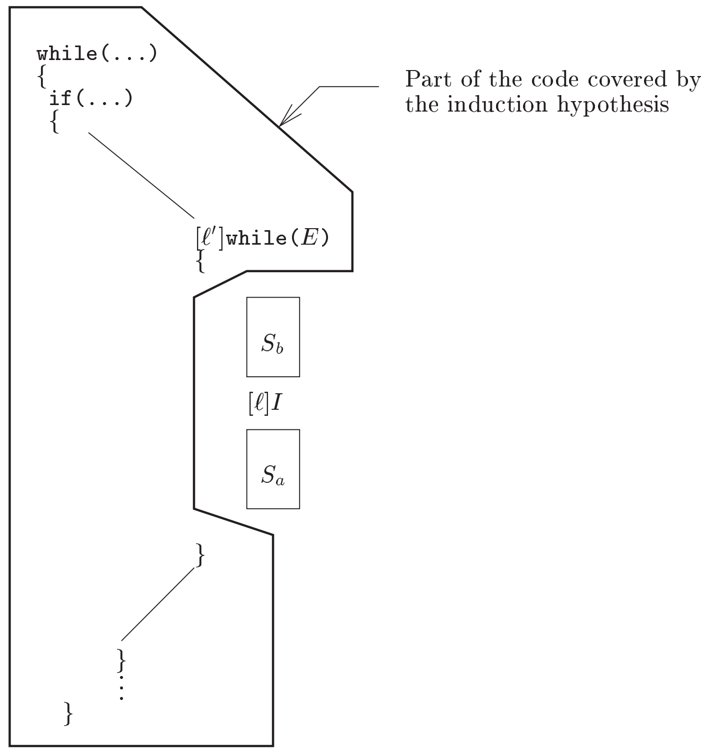

Figure 15: Graphical representation of the inductive proof of Lemma 1.

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:*

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

■

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[7]:

1. $Q_i = \{\langle \theta, \ell \rangle\}$ such that:
   - θ is a valuation of the variables in $\mathcal{V}_i$;
   - $\ell \in \ \mathit{Label}(\mathit{Relabel}(S_i))$
     $\cup \{\mathbf{lock}_{\ell'}, \mathbf{wait}_{\ell'} | \exists \ell' \in \mathit{Label}(\mathit{Relabel}(S_i)) : \gamma(\mathit{Relabel}(S_i), \ell') \equiv [\ell']\texttt{sleep} \cdot S'_i\}$
     $\cup \{\mathbf{begin}, \mathbf{end}\}$
2. δ_i and Σ_i can be constructed as follows: $\forall \ell \in \mathit{Label}(\mathit{Relabel}(S_i)) \cup \{\mathbf{begin}, \mathbf{end}\}$: we distinguish several cases depending on the value of $\gamma(\ell, \mathit{Relabel}(S_i))$. 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 $b_k = \mathit{true}$, else if $\neg v_i \in a$, then $g'_k = u_i$ and $b_k = \mathit{false}$.
   - 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 $b_j = \theta(v_i)$.

[7]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

- 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) $[\ell]\texttt{if}(E)\ \texttt{then}\{[\ell_1]I_1 \cdot S_1\}\texttt{else}\{[\ell_2]I_2 \cdot S_2\} \cdot S'$, 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 $\forall 1 \leq i \leq n : c_i \equiv v_i \vee c_i \equiv\ !v_i \vee c_i \in \{\texttt{true}, \texttt{false}\}$, 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) $[\ell]\texttt{choice}\{E_1{:}u_{1,1}, \ldots, u_{1,m_1}{:=}v_{1,1}, \ldots, v_{1,m_1}; \ldots E_n{:}u_{n,1}, \ldots, u_{n,m_n}{:=}v_{n,1}, \ldots, v_{n,m_n};\} \cdot [\ell']I' \cdot S'$, then, for every guarded assignment $E_i{:}u_{j,1}, \ldots, u_{j,m_j}{:=}v_{j,1}, \ldots, v_{j,m_j};$ $(1 \leq j \leq n)$, we apply the following steps: 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) $[\ell]\texttt{while}(E)\{[\ell_1]I_1 \cdot S_1\} \cdot [\ell_2]I_2 \cdot S_2$, 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`(L) $;\cdot\,[\ell']I'\cdot S''$, then

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

(h) $[\ell]$`unlock`(L) $;\cdot\,[\ell']I'\cdot S''$, then

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

(i) $[\ell]$`sleep`(M,L) $;\cdot\,[\ell']I'\cdot S''$, then

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

(j) $[\ell]$`wakeup`(M) $;\cdot\,[\ell']I'\cdot S''$, then

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

(k) $[\ell]$`wakeupall`(M) $;\cdot\,[\ell']I'\cdot S''$, then

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

(l) $[\ell]$`start`$(T,\pi_1,\ldots,\pi_n)$ $;\cdot\,[\ell']I'\cdot S''$, then

$$\langle\theta,\ell\rangle \xrightarrow{!T\pi_1\ldots\pi_n\ :\ \top} \langle\theta,\ell'\rangle$$

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

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

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

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

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

$$\langle \theta_\top, \mathbf{begin} \rangle \xrightarrow{?Ta_1 \ldots a_n \;:\; \top} \langle \theta_\top[v_1 = a_1, \ldots, v_n = a_n], \ell \rangle$$

for all the possible values of $a_1, \ldots, a_n$ with $a_i \in \{\top, \bot\}$, where T is the identifier of the ith type of thread.

3. Σ_i is constructed together with δ_i. More precisely:
   (a) Each time we encounter a message M as a parameter of a `wakeup`() ;, `wakeupall`() ; or `unlock`() ; during the construction of δ_i, we add M to Σ_i ;
   (b) For each `start`$(T, \pi_1, \ldots, \pi_n)$;, we add $T\pi_1 \ldots \pi_n$ to Σ_i ;
   (c) For each $\langle \theta, \ell \rangle \xrightarrow{!M\theta(u_1)\ldots\theta(u_n) \;:\; \top} \langle \theta, \ell' \rangle$ transition added to δ_i, we add to Σ_i the message $M\theta(u_1) \ldots \theta(u_n)$;
   (d) For each `accept`$(M, v_1, \ldots, v_n)$;, we add $Mv_1 \ldots v_n$ to Σ_i ;
   (e) Finally, for each thread type name T, we add the set $\{Ta_1 \ldots a_n | (a_1, \ldots, a_n) \in \{\top, \bot\}^n\}$ of messages to Σ_i.
4. The initial state I_i is $\langle \theta_\top, \mathbf{begin} \rangle$.

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 12 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 7 The translation of the CBP of Figure 14 into a set of Local Machines can be found in 3.4 (especially Figures 17 and 18). □

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 13 The φ Function

Given a thread instantiation $s = \langle S, \rho \rangle$, $\varphi(s)$ is defined as the tuple $\langle \rho, \ell \rangle$ if $\mathit{Relabel}(S) = [\ell] I \cdot S'$. □

Definition 14 The Φ Function

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

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

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

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 15 The φ' Function

Given a local state $s' = \langle \theta, \ell \rangle$ of the Local Machine *that has been obtained by translating the CBP thread* $T_i = \langle \mathcal{V}_i, S_i \rangle$, then $\varphi'(s')$ is defined as the tuple:

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

□

φ' is, indeed, a function. To compute its value, one needs to know the sequence of instructions S_i in the body of the CBP thread from which the Local Machine has been extracted. As S_i is uniquely fixed, and provided the well-formedness rules have been respected in S_i (that is, there is at most one occurrence of ℓ), one is sure of the unity of $\widetilde{\gamma}(S_i, \ell)$.

Definition 16 The Φ' Function

Given a global state $S' = \langle G, \{s_1, \dots, s_n\} \rangle$ of a GM, $\Phi(S')$ is defined as the tuple $\langle \Gamma, \Lambda, \{\varphi'(s_1), \dots, \varphi'(s_n)\} \rangle$ 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 $\langle \theta_\top, \mathbf{begin} \rangle$ 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 17 The *Live* Function
Given a global state $S' = \langle G, \{s_1, \ldots, s_n\} \rangle$ of a GM, $Live(S') = \langle G, \{live(s_1), \ldots, live(s_n)\} \rangle$ 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 18 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 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 2 *The full version of the proof is to be found in Appendix A. 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 16 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[8]. (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. ■

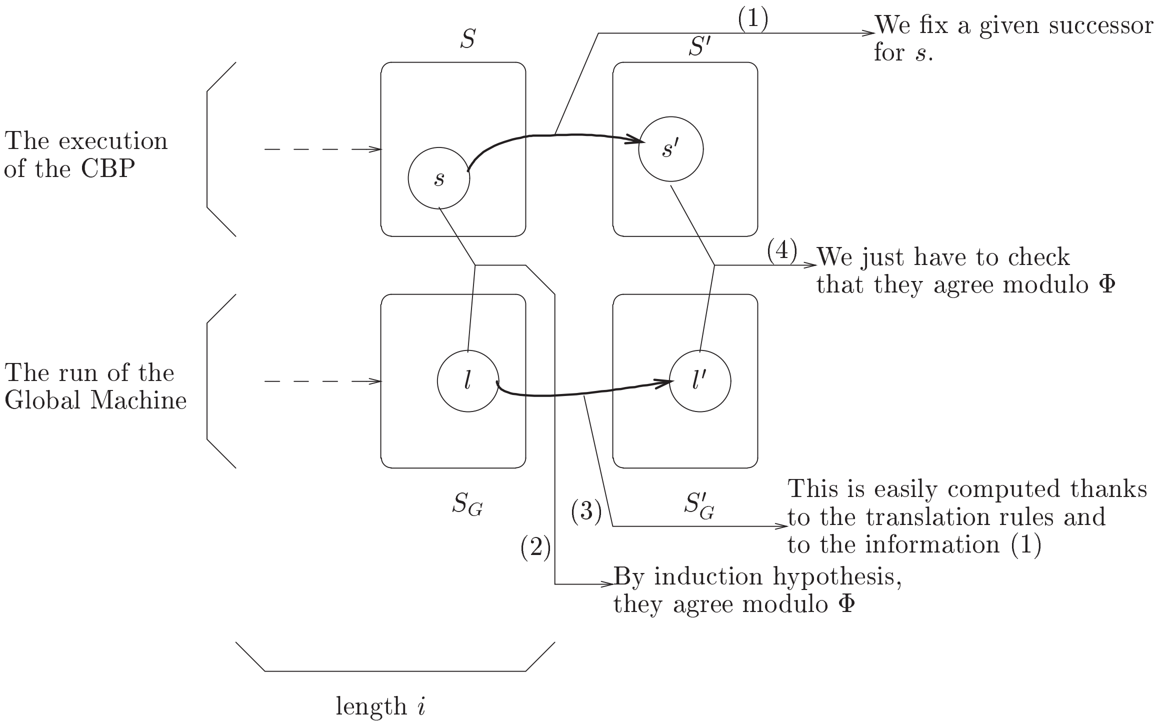

Figure 16: A graphical depiction of the inductive step to prove Lemma 2. The boxes with round corners represent global states/configurations (of the CBP and of the GM respectively). The thick arrows represent the transition relation. The numbers on the straight arrow refer to the different stages of the inductive step.

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 (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 3 *To prove this lemma, one can apply the same reasoning that was previously used for Lemma 2.*

■

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

Theorem 1 (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 4 *Stems from Lemma 2 and Lemma 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 8. Its output is a file describing the corresponding Global Machine, in a format that is suitable as input for the BABYLON toolkit [ADG+02] [9].

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

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.

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

[9] See http://www.ulb.ac.be/di/ssd/lvbegin/CST/index.html for more practical information about the BABYLON project

Example 8 As an example, look at Figure 18 and Figure 17. They present the Local Machines for the two types of threads presented at Figure 13. 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*[10], and is released under the General Public Licence. It is publicly available as a part of the BABYLON project, on the project's homepage[11].

4 Conclusion

In this paper, we have presented a new model for the specification and verification of infinite-state multi-threaded systems: the *Concurrent Boolean Programs.* 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.* The dynamic creation of an unbounded number of threads is also supported. 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.

Our final goal is the fully automatic verification of JAVA programs. Our future research toward it should address the two following problems:

1. *Abstraction of* JAVA *programs* How to obtain more precise models, allowing one to model more complex systems (i.e. systems using recursion or unbounded data structures) ?

2. *Reduction of the model's size* in the spirit of Stoller's or Corbett's works [Sto00, Cor00].

[10] See the web site of the Free Software Foundation (http://www.gnu.org) for more information about free software.

[11] http://www.ulb.ac.be/di/ssd/lvbegin/CST/index.html

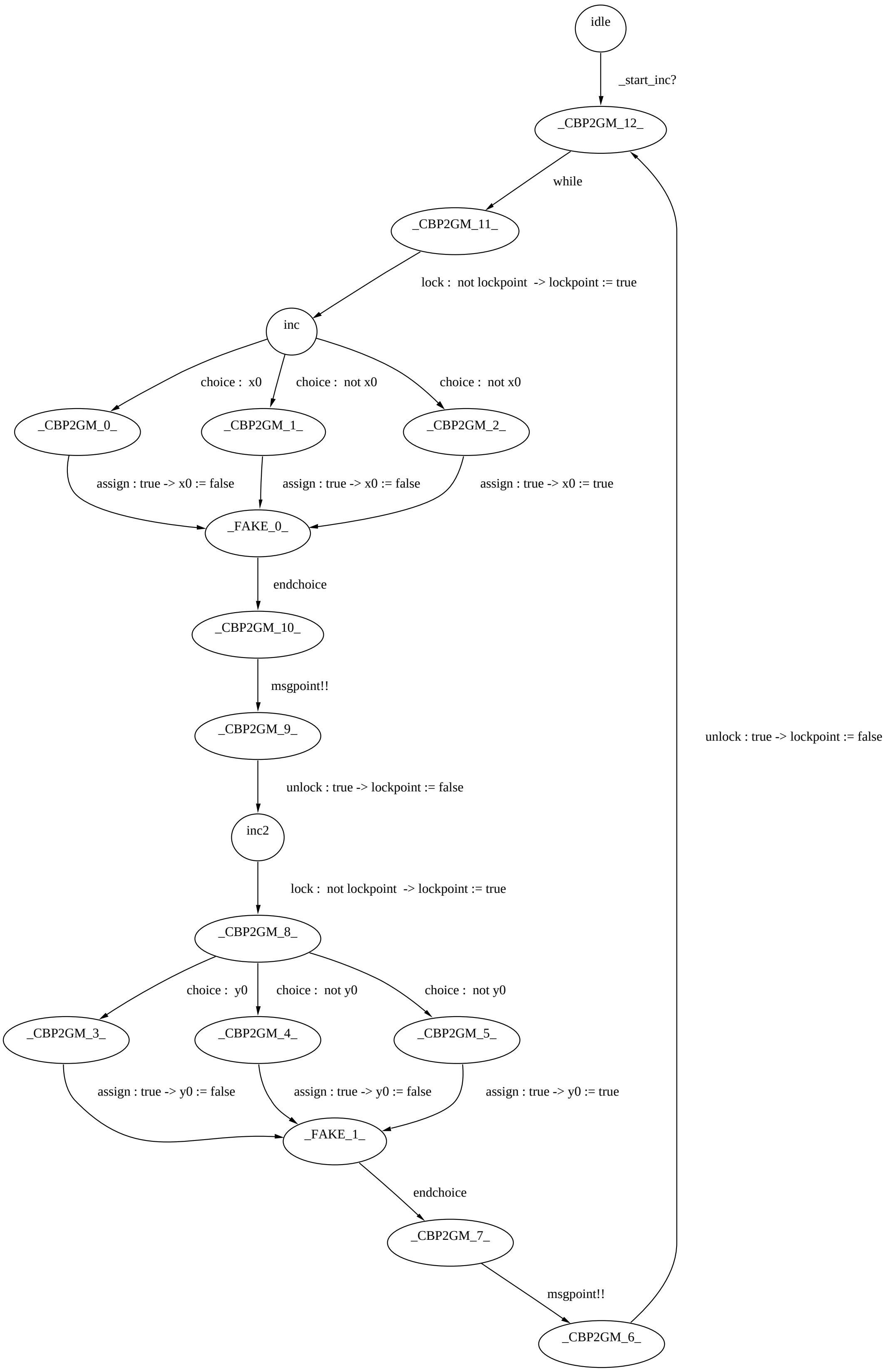

Figure 17: The Local Machine for the thread type `Inc`, generated by `CBP2GM`.

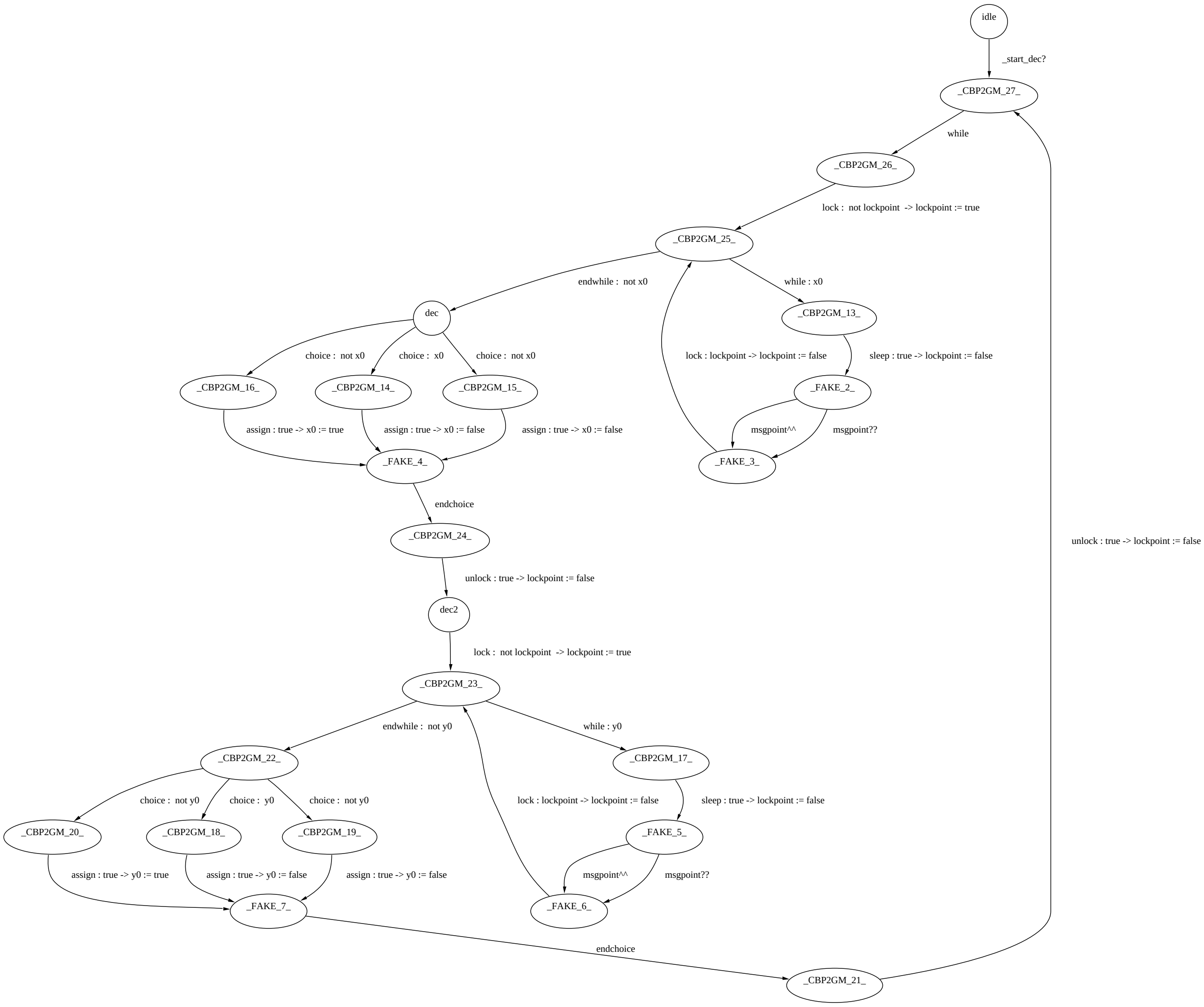

Figure 18: The Local Machine for the thread type `Dec`, generated by `CBP2GM`.

A Proof of Lemma 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, 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 assignment which just have to deal with. The case of the assignment 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)$s_i = \langle [\ell]\texttt{lock}(L); [\ell']I \cdot S, \rho\rangle$ and $s'_i = \langle [\ell']I \cdot S, \rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda[L = \mathit{locked}]$; (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, L 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 L is put to $\top$.

(h) There exists $1 \leq i \leq n$ such that (i)$s_i = \langle [\ell]\texttt{unlock}(L); [\ell']I \cdot S, \rho\rangle$ and $s'_i = \langle [\ell']I \cdot S, \rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$ and $\Lambda' = \Lambda[L = \mathit{unlocked}]$; (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, L is put to $\bot$. It follows that $\Phi(S')$ is exactly $\Phi(S)$ in which $\mathcal{L}_k$ has evolved from l_k to $l_{k'}$ and L is put to $\bot$.

(i) There exists $1 \leq i \leq n$ such that: (i) $s_i = \langle [\ell]\texttt{sleep}(M, L)\ ; \cdot [\ell']I' \cdot S'', \rho\rangle$ and $s'_i = \langle [\textbf{wait}_\ell]\mathit{wait}(M, L) \cdot [\ell']I' \cdot S'', \rho\rangle$; (ii) $n' = n$, $\Gamma' = \Gamma$, $\Lambda' = \Lambda[L = \mathit{unlocked}]$; (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 $\langle \rho, \textbf{wait}_\ell\rangle$. 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):

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

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 $l_{k_1} = \langle \rho_i, \textbf{wait}_\ell\rangle$ 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: $l'_{k_1} = \langle \rho_i, \textbf{lock}_\ell\rangle$, 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) $s_i = \langle [\ell]\texttt{wakeup}(M)\ ; \cdot [\ell']I' \cdot S'', \rho\rangle$ 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 $\{s_1, \dots s_n\}$ in three subsets $\Sigma_1 = \{s_1, \dots, s_k\}$, $\Sigma_2 = \{s_{k+1}, \dots, s_{n-1}\}$ and $\Sigma_3 = \{s_n\}$, such that: (i), for every $1 \leq i \leq k$: s_i is $\langle [\mathbf{wait}_{\ell_i}] wait(M) \cdot [I'_i].S''_i, \rho_i\rangle$; (ii), for every $k < j \leq n$: the program S_j remaining to be executed in s_j does *not* begin by $wait()$ and (iii) s_n is the state of the local machine doing the `wakeupall`. More precisely, $s_n = \langle [\ell_n]\texttt{wakeupall}(M)\ ; \cdot [\ell'_n] I'_n \cdot S''_n, \rho_n\rangle$.

Then (according to axiom 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 $1 \leq i \leq k\ :\ s_i = \langle [\mathbf{lock}_{\ell_i}] wait(M) \cdot [I'_i].S''_i, \rho_i\rangle$, 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 M. 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)$s_i = \langle [\ell]\texttt{start}(T\pi_1, \dots, \pi_m, [])\ ;\ell' I \cdot S, \rho\rangle$ 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 T 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*: $\langle \theta_\top, \mathbf{begin}\rangle$.

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 $T\pi_1 \dots \pi_m$, 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:

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

(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 $M\rho(u_1)\ldots\rho(u_n)$. 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 $\Phi(e' \cdot S) = Live(\sigma)$.

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 $\Phi(e) = Live(\sigma')$.

■

References

[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.

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

[BCR01a] T. Ball, S. Chaki, and S.K. Rajamani. Parameterized Verification of Multi-threaded 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.

[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.

[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.

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

[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.

[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.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

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

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

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

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