Fundamenta Informaticae 85 (2008) 533–548
IOS Press

Towards Verification of Java Programs in $\surd erICS^*$

Andrzej Zbrzezny, Bożena Woźna[†]
IMCS, Jan Długosz University
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland
a.zbrzezny@ajd.czest.pl
b.wozna@ajd.czest.pl

Abstract. VerICS is a tool for the automated verification of timed automata and protocols written in both the Intermediate Language and the specification language Estelle. Recently, the tool has been extended to work with Timed Automata with Discrete Data and with multi-agent systems. This paper presents a prototype Timed Automata with Discrete Data model of Java programs. In addition, we show how to use the model together with the verification core of VerICS to validate the well-known alternating bit protocol written in Java.

1. Introduction

Given a description of a system $\mathcal{S}$ and a property (specification) $\mathcal{P}$ the *verification problem* consists in establishing whether $\mathcal{S}$ satisfies $\mathcal{P}$. This can be achieved by using either *homogeneous* or *heterogeneous* verification methods. The former class of methods assumes that both the system and the specification are given in the same formalism, while the latter allows the system and the specification to be described in different formalisms.

Model checking is a verification technique that was originally developed for temporal logics. Its main idea consists in representing a finite state system, often derived from a hardware or software design, as a labelled transition system (model), representing a specification by a temporal formula, and checking automatically whether the formula holds in the model.

[*] The authors acknowledge partial support from the Ministry of Science and Higher Education under grant number 3T11C 01128.

[†] Address for correspondence: IMCS, Jan Długosz University, Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland

For the last twenty years model checking has became a widely recognised and prominent technique in hardware [16] and protocol verification [11]. Also, during the last two decades surprisingly many efficient verification tools have appeared. The most advanced and popular are SPIN [12], NuSMV [4], Uppaal [20], AVISPA [3]. This paper employs VerICS, a new verification tool developed in the past five years.

VerICS is fully automated and geared toward verification of time dependent and multi-agent systems as well as communication and security protocols. It can be downloaded from [1]. The tool is designed to accept a number of input languages that either are translated into Intermediate Language (IL)[8] or directly into a formalism called Timed Automata with Discrete Data (TADD) [14, 24]. The core of the verification engine of the tool is mainly based on translations of the model checking problem into the SAT problem [9]. Further, VerICS uses state-of-art SAT solvers like ZCHAFF [17] and RSAT [21], and it is also equipped with GUI interfaces. The architecture of VerICS is composed of the following modules:

- **Language Translator:** it takes a system description in Estelle [13], which is an ISO standard specification language designed for describing communication protocols and distributed systems, and it translates this description into IL that allows for describing a system as a set of processes, which exchange information by message passing (via bounded or unbounded channels) or memory sharing (using global variables).
- **Automata Translator:** it constructs timed automata (with discrete data) from an IL program.
- **BMC Analyser:** it verifies ECTLK [15] properties over models for timed automata and TECTL [19] properties over models for TADD.
- **UMC Analyser:** it verifies CTLK properties over models for timed automata.

In this paper, we consider the problem of verification of concurrent Java programs. We consider Java because it is now one of dominant languages for writing concurrent software. Verifying programs written in programming languages like Java is different from verifying hardware or protocols; the state space is often infinite and the relationships between possible states are harder to understand because of asynchronous behaviour and complex underlying semantics of the languages. Further, the size and complexity of software force us to treat model checking rather as a debugging technique in software verification than a fully automated validation process of the software. In particular, for what concerns verification of Java programs, we see model checking as a method that can be applied to the more crucial parts of a Java software.

In order to investigate the challenges that Java programs pose for model checking, we have developed a preliminary Timed Automata with Discrete Data (TADD) model of Java that is accurate enough to detect concurrency errors and yet abstract enough to make model checking tractable. The main aim of the paper is to describe this model, which can be further used to build a verification module of Java programs in VerICS. In addition, we show how to use our model together with the symbolic verification core of VerICS to validate the well-known alternating bit protocol written in Java.

The rest of the paper is organised as follows. The next section provides a discussion on related works. In Section 3 we define a TADD model of a subset of Java. Finally, we use the model to verify the alternating bit protocol by means the model checker VerICS.

2. Related work

Model checking of Java programs has become increasingly popular during the last decade. However, to the best of our knowledge, there are only two existing model checkers that can verify Java programs. The first tool is called JavaPathFinder [10, 23, 18] and the second one is called Bandera [6].

The first release of JavaPathFinder(JPF1) is described in [10]. JPF1 translates Java code into Promela code, which then can be model checked by means of the SPIN model checker [12]. In this realisation Java programs may contain: dynamic creation of objects with data and methods, class inheritance, threads, synchronized statements, the `wait` and `notify` methods, exceptions, thread interrupts and most of the standard programming languages constructs such as assignment statements, conditional statements and loops. However, the translator misses some features, such as packages, overloading, method overriding, recursion, strings, floating point numbers, some thread operations like suspend and resume, and some control constructs such as the continue statement. In addition, arrays are not objects as they are in Java, but are modelled using Promela's own arrays to obtain efficient verification.

The second generation of JavaPathFinder(JPF2) is described in [23, 18]. This version of JPF2 combines model checking techniques with techniques for dealing with large or infinite state spaces. These techniques include a static analysis for supporting partial order reductions of the set of transitions to be explored by the model checker, a predicate abstraction for abstracting the state space, and a runtime analysis such as race condition detection and lock order analysis to pinpoint potentially problematic code fragments. JPF2 techniques operate on the Java bytecode. Currently, the NASA model-checker Java PathFinder is one of the backend model-checkers supported by Bandera.

The Bandera tools [7, 6] are designed to compile a Java program to a finite-state transition system that can be processed by model-checking tools. This is accomplished by first compiling a Java source code into Java byte-code. Second, using the Soot Java compiler framework [22] Java byte-code is translated to an intermediate language called *Jimple*. Jimple is essentially a language of control-flow graphs where:

- statements appear in a three-address-code form, and
- various Java constructs such as method invocations and synchronized statements are represented in terms of their virtual machine counterparts (such as: invokevirtual, monitorenter and monitorexit).

Third, the Jimple code is translated into a guarded command language called BIR (Bandera Intermediate Representation). The BIR model can then be translated into the input languages of different model checking tools. In particular there are translations from BIR to Promela (the input language of SPIN) and the Bogor input language.

There are basically 2 releases of Bandera: 0.x and 1.x. The 0.x releases work but it is hard to experiment with them. Only in this version however there is a translation form BIR to Promela. The 1.x releases are focused on direct work with the Bogor model checking framework only, and they do not support the translators or tools to work with SMV or Spin/Promela.

The Bandera tools give a direct support for unbounded dynamic creation of threads and objects, automatic memory management (garbage collection), virtual methods and exceptions. It supports virtually all of the Java language features. However, Bandera does not currently model full Java standard library. In particular, some of Java library classes were taken from the GNU Classpath library, and still native calls are not modelled.

3. A TADD model of Java programs

This section describes a prototype Timed Automata with Discrete Data (TADDs) model of Java programs. TADDs [24] are standard diagonal timed automata augmented to include integer variables over which standard arithmetic and Boolean expressions can be defined. These automata take as an input a set of initialised integer variables and a set of propositional variables, true at particular states.

We start by describing the *alternating bit protocol* (ABP), a well-known communication protocol that is often used as a test case for automated verification tools, and then we briefly describe concurrency model in Java.

3.1. Alternating Bit Protocol

The protocol involves three active components: a sender, a receiver and a bidirectional communication channel. Each message contains a control bit only and is sent from the sender to the receiver through an unreliable communication channel. The protocol works in the following way. The sender starts sending the bit to the receiver, which is initially silent. The sending of the bit proceeds until the sender receives an acknowledgement. After that, the sender flips the control bit and starts all over again. As soon as the receiver receives the control bit that matches its internal control bit, an acknowledgement is sent to the sender. Then, the receiver flips his internal control bit and waits for another bit.

The communication channel transmits bits both from the sender to the receiver, and from the receiver to the sender. There are four possible situations that can occur. Bits are properly transmitted from the sender (the receiver) to the receiver (the sender), or bits are corrupted or lost during such a transmission.

Listing 1 shows a Java source code of ABP, which will then be modelled as a network of TADDs that run in parallel, communicate with each other via shared variables and perform transitions with shared labels synchronously; we assume here a definition of parallel composition [24] that is an extension of the standard definition of parallel composition [2] that takes into account integer variables.

3.2. Concurrency in Java

In Java, threads are instances either of the class `Thread` or subclasses of the class `Thread`, that is, there are two ways of creating threads: implementing an interface `Runnable` or extending the class `Thread`. We consider the most common way of creating threads, that is, by using the constructor for `Thread` that takes as a parameter any object implementing the interface `Runnable`, which essentially means the object has a method `run()`. Once a thread is started by calling its `start()` method, the thread executes the `run()` method of this object. Although threads may have assigned priorities to control scheduling, in this paper we assume that all threads have equal priorities and are scheduled arbitrarily.

Consider the ABP system shown in Listing 1. In this example, the program starts, as usual, by executing the static method `main()`. As a result there are created: (1) an instance `channel` of `LossyChannel`, (2) instances of `Sender` and `Receiver` that take `channel` as an argument, (3) two instances of `Thread` that take as arguments the object of the class `Sender` and `Receiver`, respectively. Then, all the threads are started, which means that their `run()` methods are executed. The Sender and Receiver threads put/get bit to the shared channel.

Recall that every Java object has an implicit lock. When a thread executes a synchronized method, it must acquire the lock of the object on which the method has been invoked before executing the body

```java
import java.util.Random;
public class AltBitProtocol {
  public static void main(String[] args) {
    LossyChannel channel = new LossyChannel();
    (new Thread(new Sender(channel))).start();
    (new Thread(new Receiver(channel))).start();
  }
}
class LossyChannel {
  private boolean protocolBit, channelEmpty = true, ackBit = true;
  private Random r;
  public LossyChannel() {r = new Random();}
  public synchronized boolean getAckBit() {notify(); return ackBit;}
  public synchronized void putAckBit(boolean ackBit) {
    this.ackBit = ackBit;
    int ignoreBit = r.nextInt(2);
    if (ignoreBit > 0) {this.ackBit = !this.ackBit;} notify();
  }
  public synchronized boolean get() {
    while (!channelEmpty) {try {wait();} catch (InterruptedException e){}}
    channelEmpty = true; notify();  return protocolBit;
  }
  public synchronized void put(boolean protocolBit) {
    while (!channelEmpty) {try {wait();} catch (InterruptedException e){}}
    int ignoreBit = r.nextInt(2);
    channelEmpty = false; if (ignoreBit > 0) channelEmpty = true;
    notify();
  }
}
class Sender implements Runnable {
  private LossyChannel channel;
  public Sender(LossyChannel channel) {this.channel = channel;}
  public void run() {
    boolean protocolBit = false; Random r = new Random();
    while (true) {
      if (protocolBit != channel.getAckBit()) channel.put(protocolBit);
      else protocolBit = !protocolBit;
      try {Thread.sleep(r.nextInt(1500));} catch (InterruptedException e){}
    }
  }
}
class Receiver implements Runnable {
  private LossyChannel channel;
  public Receiver(LossyChannel channel) {this.channel = channel;}
  public void run() {
    Random r = new Random();
    while (true) {
      boolean protocolBit = channel.get(); channel.putAckBit(protocolBit);
      try {Thread.sleep(r.nextInt(1500));} catch (InterruptedException e){}
    }
  }
}
```

Listing 1. Java source code of the alternating bit protocol

of the method, releasing the lock when the body of the method is exited. If the lock is unavailable, the thread will be blocked until the lock is released. In addition, every object also provides the wait-and-notify mechanism that allows threads to be in waiting states. This wait-and-notify technique is a communication mechanism between threads; it allows one thread to communicate to another thread when a particular condition has occurred. In our ABP example callers of `put()` and `putAckBit()` must wait either for an ackBit or for a fresh bit in the channel, callers of `get()` and `getAckBit()` must wait until the channel is nonempty. That is, the precondition for any operation mentioned above is checked, and, if false, the thread blocks itself on the object by executing the `wait()` method, which releases the lock. When a method changes the state of the object in such a way that a precondition might now be true, it executes the `notify()` method, which wakes up a thread waiting on the object.

3.3. A TADD model

We model a concurrent multi-threaded Java program with a network of TADDs. Each state of TADD is an abstraction of the state of the Java program, and each transition represents the execution of code transforming this abstract state. The modelled subset of Java contains: definitions of integer variables, standard programming language constructs like assignment statements, conditional statements and loops, definitions of classes, objects, methods and threads, synchronized and static methods (blocks), and the methods `wait()`, `notify()`, `sleep()` and `random()`.

The method of constructing the TADD model of a Java program is the following. State variables are used to record the current control location of each thread and the values of key program variables and any run-time information necessary to implement the concurrent semantics (e.g., whether each thread is ready, running, or blocked on some object). Each transformation represents the execution of a Java instruction (for example an assignment statement) for some thread. There is one TADD for each thread: one TADD for the `main()` method and `n` identical TADDs for each `run()` method of class C (where n is the number of instances for class C). In this paper we assume all method calls have been inlined; this limits the analysis to programs with statically bounded recursion.

To produce an automaton for a thread, we proceed as follows. First, for all the variables `var` of type `int` or `Boolean` of the considered thread class that are defined in the body of the method `run()`, we introduce corresponding local integer variables `threadName_var`. Next, we model `run()` methods according to the following scheme:

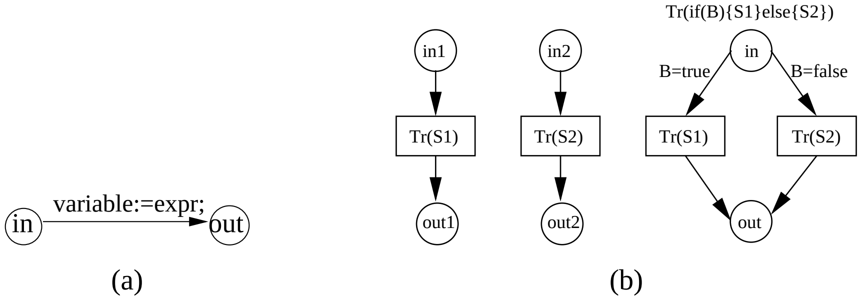


Figure 1. An automata model of: (a) an assignment statement; (b) a conditional statement.

- If there is an assignment statement of the form `variable = expr;`, then we produce an automaton as it is shown in Figure 1a.

- If there is a conditional statement of the form `if(condition B)`$\{$`statement S1;`$\}$ `else` $\{$`statement S2;`$\}$, then we first build automata for $S1$ and $S2$, written $Tr(S1)$ and $Tr(S2)$. Next, we produce an automaton that is a sum of automata $Tr(S1)$ and $Tr(S2)$ such that the initial locations $in1$ and $in2$ become one initial location in, and the final locations $out1$ and $out2$ become one final location out (see Figure 1b).

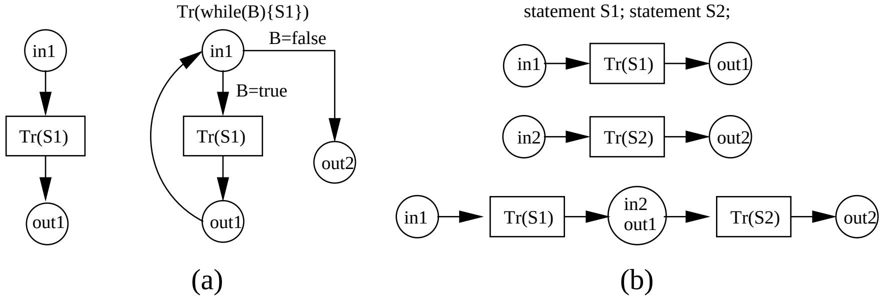
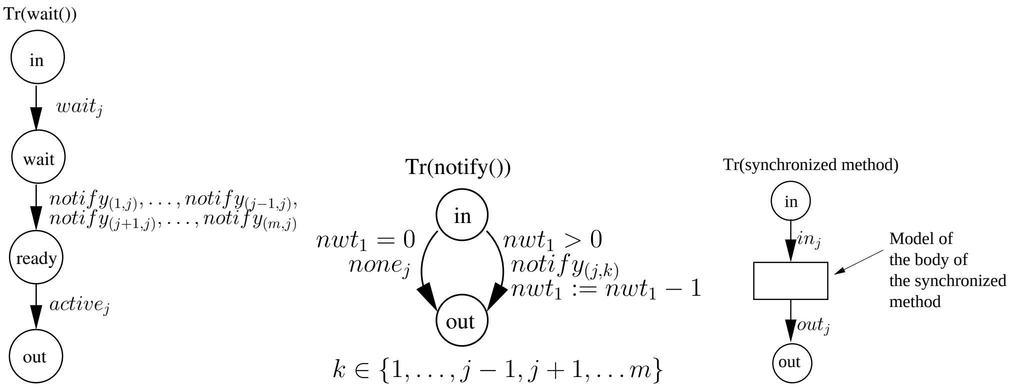


Figure 2. An automata model of: (a) a loop statement; (b) concatenation of two statements.

- If there is a loop statement of the form `while(condition B)`$\{$`statement S1;`$\}$, then we first build an automaton for $S1$, written $Tr(S1)$, and next we produce an automaton as it is shown in Figure 2(a).
- If there are two consecutive statements `statement S1; statement S2;`, then we first build automata for $S1$ and $S2$, written $Tr(S1)$ and $Tr(S2)$. Next, we produce an automaton which is the concatenation of $Tr(S1)$ and $Tr(S2)$, that is, the new automaton has as an initial location the initial location of $Tr(S1)$ (i.e., in1), as a final location the final location of $Tr(S2)$ (i.e., out2), and as a “contact” location a new location (out1-in2), which is a merger of the final location of $Tr(S1)$ (i.e., out1) and the initial location of $Tr(S2)$ (i.e., in2) (see Figure 2(b)).
- A model of synchronized methods is shown in Figure 3(c). The way synchronization is realised is the following. Let l be a number of objects on which synchronized methods are invoked. First, we introduce l binary semaphores and l special variables $nwt_1, \ldots, nwt_l$. The value of variable nwt_i is equal to the numebr of threads waiting on the lock of the object number i. In particular, initially all the variables are set to zero. Then, we build an automaton that has two special locations (in and out), and two special transitions that are labelled with synchronized actions in_j and out_j, respectively, where j denotes the number of a thread which has invoked the synchronized method under consideration, in denotes acquiring a lock of the object on which the method has been invoked and the entrance to the method, and out denotes releasing the lock and exit from the method. The labels in_j and out_j will have the following general form: $x\_threadName\_threadNumber$ with $x \in \{in, out\}$ (for example in_sender_1, out_sender_1 - see the resulting automata for ABP).
- A model of the method `wait()` is as shown in Figure 3(a), that is, we have an automaton of four locations with $(m-1)+2$ transitions (m is the number of threads). The first transition is labelled by action $wait_j$ which denotes the fact that thread number j goes to a *waiting state* and opens the semaphore. In this state the thread is waiting to be notified by any other thread. Here this is represented by the actions $notify_{(1,j)}, \ldots, notify_{(j-1,j)}, notify_{(j+1,j)}, \ldots, notify_{(m,j)}$; these are synchronized actions between threads. Once that action happens, the thread gets into the *ready state.* A thread in this state is ready for execution, but is not being currently executed. Once a



(a) wait() (b) notify() (c) a synchronized method

Figure 3. An automata model of methods *wait()*, *notify()* and a synchronized method for thread j.

thread in the ready state gets access to the CPU, it gets converted to the running state. This is done by invoking a synchronized action $active_j$, which will close the semaphore.

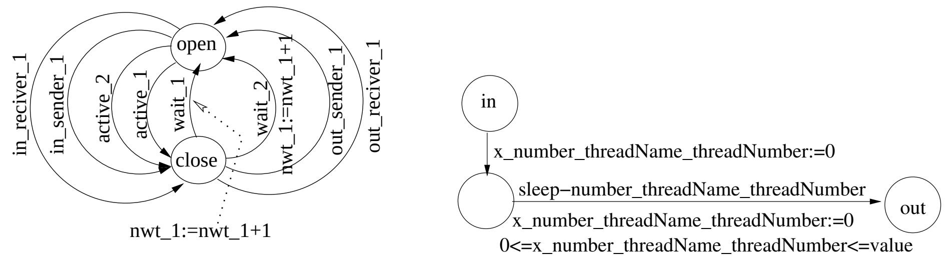
- A model of the method `notify()` that wakes up an arbitrary chosen thread from all the threads waiting on the object's lock is as shown in Figure 3(b). Namely, we have an automaton of two locations with m transitions labelled by $notify_{(j,k)}$, for $k \in \{1, \ldots, j-1, j+1, \ldots, m\}$, and $none_j$, respectively. The synchronized action $notify_{(j,k)}$ represents the fact that thread number j notifies the thread number k, which thereby is moved to the *ready state*. This action can be performed only if the value of the variable nwt_i is grater than 0, where i is the number of the object on which the synchronized method under consideration is invoked. Further, with this action an instructions $nwt_i := nwt_i - 1$ is associated, which means that the number of the waiting threads has been decreased by one. If there are no waiting threads, the `notify()` method has no effect; this is realised by the local action $none_j$, which is enabled only if the value of the variable nwt_i is equal to zero.

- A model of the method *sleep()* consists in introducing a new clock with the general name $x\_number\_threadName\_threadNumber$ (precisely, one clock for all instances of the method sleep() in a thread) and producing an automaton that consists of three locations and two transitions (see Figure 4(b)). The first transition must be labelled by the reset operation of the form $x\_number\_threadName\_threadNumber := 0$; the second transition must be labelled by: (1) an action with general name $sleepNumber\_threadName\_threadNumber$ (for example, $sleep1\_sender\_1$); (2) the reset operation of the form $x\_number\_threadName\_threadNumber := 0$; and (3) a guard of the form $0 \leq x\_number\_threadName\_threadNumber \leq value$, where *value* is a parameter of the method *sleep()* (for example, $0 \leq x1\_sender\_1 \leq 1500$).

- A model of the method *random()* is simulated by an automaton that has two locations and n transitions (n is a value of the parameter of the method random()) decorated with the instructions of the form $x := i$, where x is an integer variable and $i \in \{0, \ldots, n-1\}$.

- Values that are returned by methods of a non-void type are stored in integer variables with the general name $threadName\_threadNumber\_methodName$.

- Passing parameters of the type *int* and *boolean* to a function is realised in the following two steps. First, we introduce as many new integer variables as the function under consideration has parameters; the general name of the new variables is $className\_methodName\_parameterName$. Second, we introduce a set of assignment instructions of the form: $className\_methodName\_parameterName := value$, where $value$ is the value of a parameter with the name $parameterName$. This instruction is attached to a transition labelled with the action $in\_threadName\_threadNumber$, if the function under consideration is a synchronized function. Otherwise, this instruction is attached to a transition labelled with the action $in\_className\_methodName$.


open
close
in_reciver_1
in_sender_1
active_2
active_1
wait_1
wait_2
nwt_1:=nwt_1+1
out_sender_1
out_reciver_1
nwt_1:=nwt_1+1
in
x_number_threadName_threadNumber:=0
sleep−number_threadName_threadNumber
x_number_threadName_threadNumber:=0
0<=x_number_threadName_threadNumber<=value
out

(a) (b)

Figure 4. Automata for: (a) semaphore (b) method *sleep()*.

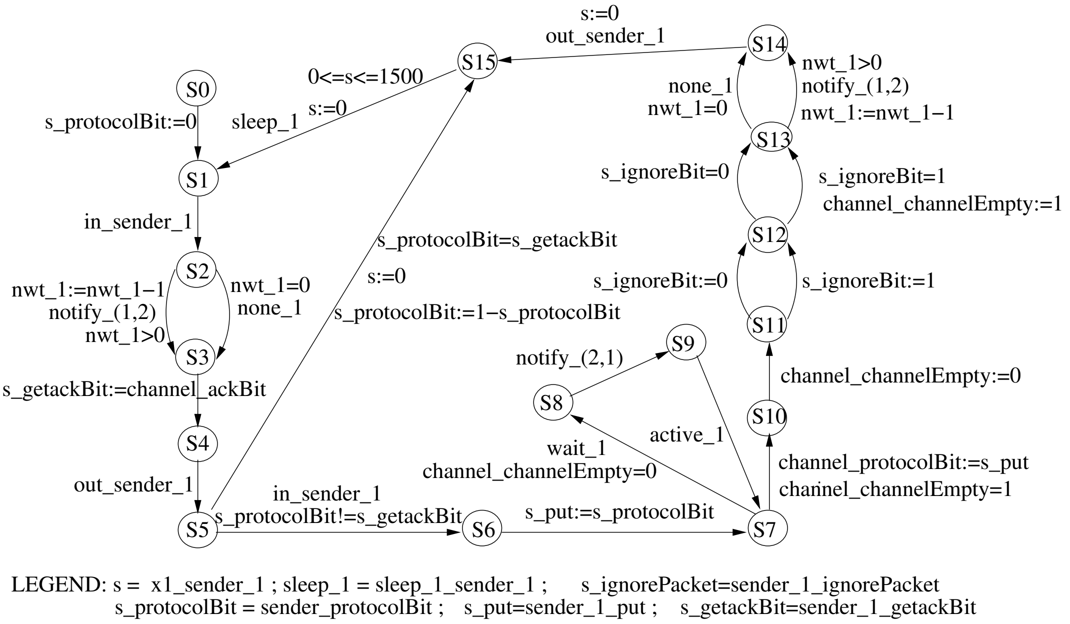


Figure 5. An automaton for Sender

Given the above it is easy to infer the network of timed automata with discrete data that model Alternating Bit Protocol; this is shown in Figures 4(a), 5 and 6.

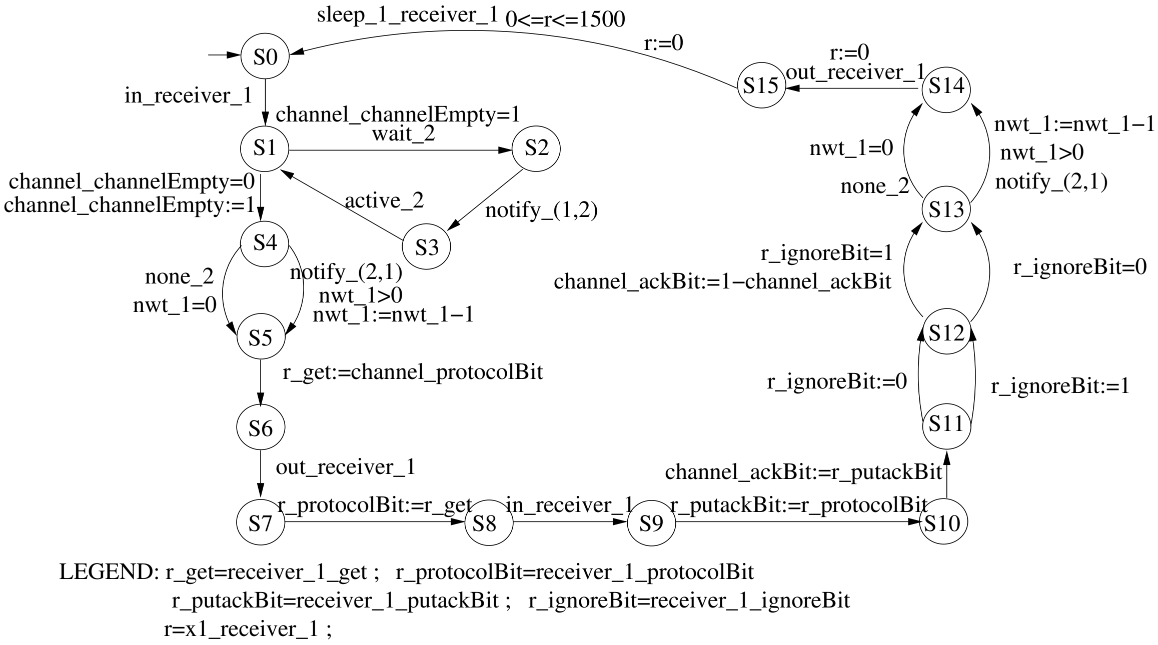


Figure 6. An automaton for Receiver

4. Experimental results

In this section we show how to use our model together with the verification core of VerICS (in fact the BMC module) to validate the Alternating Bit Protocol written in Java; the tested code is shown in Listing 1 in Section 3. All the experiments have been performed on a computer equipped with the processor Intel Celeron M 1.30GHz, 768 MB RAM and the operating system Linux 2.6.24.

Since the BMC module is designed to look for errors, we have introduced in our Java code the following errors: in the class *LossyChannel* we have set the variable *channelEmpty* to false (i.e., we have put *channelEmpty=0*), and we have exchanged the condition of the while loop in the method *get()* to the following one: `while (!channelEmpty){...}`.

These two small changes cause the protocol to stop working nearly immediately, which means that a deadlock has occurred, because the both threads are waiting for two locks to be freed and the circumstances in the program are such that the locks are never freed. To find the cause of this behaviour, we have checked the modified Java code of Alternating Bit Protocol for the existence of deadlocks. The simplest way to find deadlocks is to verify whether the generated network of TADDs admits finite runs only. To this end, it is enough to check whether the ECTL formula $EFtrue$ [5] is true in the model of the considered network for any possible length of a run. In our example, it is possible to verify that there is no run of length longer than 24. The generated witness shows the reason for such a situation: the protocol gets into a state where both threads wait for a notification from another thread, but none of them can invoke the method *notify()*. Table 1 reports this witness; experimental results are reported in Table 2.

Length	**locations**	**values of variables**	**Sender's clock**	**Receiver's clock**
0	$< 0,0,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$0\ \frac{0}{16384}$	$0\ \frac{0}{16384}$
1	$< 0,0,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$571\ \frac{13487}{16384}$	$571\ \frac{13487}{16384}$
2	$< 0,1,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$571\ \frac{13487}{16384}$	$571\ \frac{13487}{16384}$
3	$< 0,1,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$881\ \frac{11118}{16384}$	$881\ \frac{11118}{16384}$
4	$< 0,2,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$881\ \frac{11118}{16384}$	$881\ \frac{11118}{16384}$
5	$< 0,2,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$1289\ \frac{10857}{16384}$	$1289\ \frac{10857}{16384}$
6	$< 1,2,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$1289\ \frac{10857}{16384}$	$1289\ \frac{10857}{16384}$
7	$< 1,2,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$2175\ \frac{14820}{16384}$	$2175\ \frac{14820}{16384}$
8	$< 2,2,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$2175\ \frac{14820}{16384}$	$2175\ \frac{14820}{16384}$
9	$< 2,2,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$2478\ \frac{15509}{16384}$	$2478\ \frac{15509}{16384}$
10	$< 3,3,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$2478\ \frac{15509}{16384}$	$2478\ \frac{15509}{16384}$
11	$< 3,3,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$3089\ \frac{5652}{16384}$	$3089\ \frac{5652}{16384}$
12	$< 4,3,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$3089\ \frac{5652}{16384}$	$3089\ \frac{5652}{16384}$
13	$< 4,3,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$4349\ \frac{9746}{16384}$	$4349\ \frac{9746}{16384}$
14	$< 5,3,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$4349\ \frac{9746}{16384}$	$4349\ \frac{9746}{16384}$
15	$< 5,3,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$5674\ \frac{7304}{16384}$	$5674\ \frac{7304}{16384}$
16	$< 5,1,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$5674\ \frac{7304}{16384}$	$5674\ \frac{7304}{16384}$
17	$< 5,1,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,0 >$	$6049\ \frac{2753}{16384}$	$6049\ \frac{2753}{16384}$
18	$< 5,2,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$6049\ \frac{2753}{16384}$	$6049\ \frac{2753}{16384}$
19	$< 5,2,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$6952\ \frac{3525}{16384}$	$6952\ \frac{3525}{16384}$
20	$< 6,2,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$6952\ \frac{3525}{16384}$	$6952\ \frac{3525}{16384}$
21	$< 6,2,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$8194\ \frac{15846}{16384}$	$8194\ \frac{15846}{16384}$
22	$< 7,2,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$8194\ \frac{15846}{16384}$	$8194\ \frac{15846}{16384}$
23	$< 7,2,1 >$	$< 0,0,0,0,0,1,0,0,1,0,0,1 >$	$8738\ \frac{7928}{16384}$	$8738\ \frac{7928}{16384}$
24	$< 8,2,0 >$	$< 0,0,0,0,0,1,0,0,1,0,0,2 >$	$8738\ \frac{7928}{16384}$	$8738\ \frac{7928}{16384}$

Table 1: The witness

k	variables	clauses	BMC sec	BMC MB	RSAT sec	RSAT MB	SAT
0	416	619	0.0	3.0	0.0	1.3	YES
2	3482	9088	0.1	3.3	0.0	2.0	YES
4	6686	18018	0.3	3.7	0.0	2.6	YES
6	10039	27361	0.4	4.1	0.1	3.4	YES
8	13359	36608	0.6	4.5	0.1	4.2	YES
10	16745	46033	0.7	4.9	0.1	4.9	YES
12	20484	56446	0.9	5.3	0.2	5.8	YES
14	24049	66361	1.0	5.8	0.3	6.5	YES
16	27680	76454	1.2	6.2	0.5	7.3	YES
18	31377	86725	1.4	6.6	1.6	8.3	YES
20	35140	97174	1.6	7.1	0.9	9.1	YES
22	39491	109281	1.8	7.6	2.2	10.3	YES
24	43433	120220	2.0	8.0	1.7	11.1	YES
26	47441	131337	2.1	8.5	0.4	11.8	NO
		In total:	14.1	8.5	8.1	11.8	

Table 2: Property EF$true$.

In Table 1 the first column shows the length of the witness, the second column shows locations of Sender, Receiver and Semaphore, respectively, the third column shows values of integer variables with the meaning: z0: channel_channelEmpty, z1: channel_protocolBit, z2: r_get, z3: r_protocolBit, z4: r_put_ackBit, z5: channel_ackBit, z6: r_ignoreBit, z7: s_protocolBit, z8: s_getackBit, z9: s_put, z10: s_ignoreBit, z11: nwt_1 and the last two columns show the values of clocks for Sender and Receiver, respectively.

For the second experiment, we have introduced in our Java code only one error by modifying the method *get()* as illustrated above. This small change causes the protocol either to stop working, or to perform some infinite loop.

The reason for the deadlock is the following: the protocol can reach a state where both threads wait for a notification from another thread, but none of them can invoke the method *notify()*. We can show formally that such a situation indeed exists by checking reachability of a state in our TADD model which contains ”wait” locations (i.e., locations to which automata for Sender and Receiver can get in, if actions *wait_1* and *wait_2* are performed). We have done such a reachability test, and we have found a witness of length 46 which shows that both Sender and Receiver can both move to a waiting state, and there is no way for them to move to a ready state; for this witness see Table 3; experimental results are presented in Table 4.

The reason the modified Java program may go into an infinite loop can be found by examining the witnesses for the reachability property mentioned above: the Sender can continuously send messages, but he never gets an acknowledgement.

Length	**locations**	**values of variables**	**Sender’s clock**	**Receiver’s clock**
0	$< 0,0,0 >$	$< 1,0,0,0,0,1,0,0,1,0,0,0 >$	$0\ \frac{0}{33554432}$	$0\ \frac{0}{33554432}$
1	$< 0,0,0 >$	$< 1,0,0,0,0,1,0,0,1,0,0,0 >$	$638\ \frac{31095471}{33554432}$	$638\ \frac{31095471}{33554432}$

2	$< 1, 0, 0 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$638 \frac{31095471}{33554432}$	$638 \frac{31095471}{33554432}$
3	$< 1, 0, 0 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$1421 \frac{28288043}{33554432}$	$1421 \frac{28288043}{33554432}$
4	$< 2, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$1421 \frac{28288043}{33554432}$	$1421 \frac{28288043}{33554432}$
5	$< 2, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2712 \frac{14254826}{33554432}$	$2712 \frac{14254826}{33554432}$
6	$< 3, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2712 \frac{14254826}{33554432}$	$2712 \frac{14254826}{33554432}$
7	$< 3, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2825 \frac{16790440}{33554432}$	$2825 \frac{16790440}{33554432}$
8	$< 4, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2825 \frac{16790440}{33554432}$	$2825 \frac{16790440}{33554432}$
9	$< 4, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$3248 \frac{5394119}{33554432}$	$3248 \frac{5394119}{33554432}$
10	$< 5, 0, 0 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$3248 \frac{5394119}{33554432}$	$3248 \frac{5394119}{33554432}$
11	$< 5, 0, 0 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$3356 \frac{870022}{33554432}$	$3356 \frac{870022}{33554432}$
12	$< 6, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$3356 \frac{870022}{33554432}$	$3356 \frac{870022}{33554432}$
13	$< 6, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$3936 \frac{132865}{33554432}$	$3936 \frac{132865}{33554432}$
14	$< 7, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$3936 \frac{132865}{33554432}$	$3936 \frac{132865}{33554432}$
15	$< 7, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$5270 \frac{16845056}{33554432}$	$5270 \frac{16845056}{33554432}$
16	$< 10, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$5270 \frac{16845056}{33554432}$	$5270 \frac{16845056}{33554432}$
17	$< 10, 0, 1 >$	$< 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$6665 \frac{15302656}{33554432}$	$6665 \frac{15302656}{33554432}$
18	$< 11, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$6665 \frac{15302656}{33554432}$	$6665 \frac{15302656}{33554432}$
19	$< 11, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$8161 \frac{32515834}{33554432}$	$8161 \frac{32515834}{33554432}$
20	$< 12, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$8161 \frac{32515834}{33554432}$	$8161 \frac{32515834}{33554432}$
21	$< 12, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$9646 \frac{17719040}{33554432}$	$9646 \frac{17719040}{33554432}$
22	$< 13, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$9646 \frac{17719040}{33554432}$	$9646 \frac{17719040}{33554432}$
23	$< 13, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$10018 \frac{14559156}{33554432}$	$10018 \frac{14559156}{33554432}$
24	$< 14, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$10018 \frac{14559156}{33554432}$	$10018 \frac{14559156}{33554432}$
25	$< 14, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$11039 \frac{18366376}{33554432}$	$11039 \frac{18366376}{33554432}$
26	$< 15, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$0 \frac{0}{33554432}$	$11039 \frac{18366376}{33554432}$
27	$< 15, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$991 \frac{20979606}{33554432}$	$12031 \frac{5791550}{33554432}$
28	$< 1, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$0 \frac{0}{33554432}$	$12031 \frac{5791550}{33554432}$
29	$< 1, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$616 \frac{27694219}{33554432}$	$12647 \frac{33485769}{33554432}$
30	$< 2, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$616 \frac{27694219}{33554432}$	$12647 \frac{33485769}{33554432}$
31	$< 2, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$1524 \frac{5345448}{33554432}$	$13555 \frac{11136998}{33554432}$
32	$< 3, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$1524 \frac{5345448}{33554432}$	$13555 \frac{11136998}{33554432}$
33	$< 3, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$1681 \frac{32498720}{33554432}$	$13713 \frac{4735838}{33554432}$

34	$< 4, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$1681\ \frac{32498720}{33554432}$	$13713\ \frac{4735838}{33554432}$
35	$< 4, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2161\ \frac{22637552}{33554432}$	$14192\ \frac{28429102}{33554432}$
36	$< 5, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2161\ \frac{22637552}{33554432}$	$14192\ \frac{28429102}{33554432}$
37	$< 5, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2225\ \frac{9449313}{33554432}$	$14256\ \frac{15240863}{33554432}$
38	$< 6, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2225\ \frac{9449313}{33554432}$	$14256\ \frac{15240863}{33554432}$
39	$< 6, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2861\ \frac{7524704}{33554432}$	$14892\ \frac{13316254}{33554432}$
40	$< 7, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$2861\ \frac{7524704}{33554432}$	$14892\ \frac{13316254}{33554432}$
41	$< 7, 0, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 >$	$3621\ \frac{8083799}{33554432}$	$15652\ \frac{13875349}{33554432}$
42	$< 8, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1 >$	$3621\ \frac{8083799}{33554432}$	$15652\ \frac{13875349}{33554432}$
43	$< 8, 0, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1 >$	$4724\ \frac{26668746}{33554432}$	$16755\ \frac{32460296}{33554432}$
44	$< 8, 1, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1 >$	$4724\ \frac{26668746}{33554432}$	$16755\ \frac{32460296}{33554432}$
45	$< 8, 1, 1 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1 >$	$5436\ \frac{27239241}{33554432}$	$17467\ \frac{33030791}{33554432}$
46	$< 8, 2, 0 >$	$< 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 2 >$	$5436\ \frac{27239241}{33554432}$	$17467\ \frac{33030791}{33554432}$

Table 3: The witness

k	variables	clauses	BMC sec	BMC MB	RSAT sec	RSAT MB	SAT
0	422	637	0.0	3.0	0.0	1.3	NO
2	3492	9118	0.1	3.3	0.0	2.0	NO
4	6700	18060	0.3	3.7	0.0	2.7	NO
6	10057	27415	0.4	4.1	0.0	3.4	NO
8	13381	36674	0.6	4.5	0.1	4.2	NO
10	16771	46111	0.7	4.9	0.1	4.9	NO
12	20514	56536	0.9	5.3	0.1	5.8	NO
14	24083	66463	1.1	5.8	0.2	6.6	NO
16	27718	76568	1.2	6.2	0.2	7.4	NO
18	31419	86851	1.4	6.6	0.3	8.3	NO
20	35186	97312	1.6	7.1	0.3	9.1	NO
22	39541	109431	1.8	7.6	0.5	10.0	NO
24	43487	120382	1.9	8.0	0.5	11.0	NO
26	47499	131511	2.1	8.5	0.6	11.8	NO
28	51577	142818	2.3	9.0	0.9	12.7	NO
30	55721	154303	2.5	9.4	1.7	13.7	NO
32	59931	165966	2.7	9.9	1.6	14.5	NO
34	64207	177807	2.9	10.4	2.3	15.5	NO
36	68549	189826	3.1	11.0	1.6	16.5	NO
38	72957	202023	3.3	11.5	1.9	17.4	NO
40	77431	214398	3.5	12.0	2.6	18.5	NO

42	81971	226951	3.7	12.5	3.3	19.2	NO
44	87616	242636	4.0	13.3	2.5	20.8	NO
46	92335	255679	4.2	13.8	18.8	24.5	YES
		In total:	46.2	13.8	40.1	24.5	

Table 4: Property $EF(wait\ locations)$

5. Summary

We have proposed a preliminary Timed Automata with Discrete Data model of Java programs, which is going to be a base for a verification module of the tool VerICS. Our method exploits two basic constructs of the concurrency model in Java: data accessible by only one thread (mutual exclusion) and semaphore (a classic concurrency control construct).

The process of extracting TADD models from the Java source code, to some degree, depends on the language. Although, many aspects of our method are more widely applicable and could be used to model programs written in other languages, for example C/C++, ADA.

We have also provided preliminary experimental results. Although we have constructed the TADD model for the source code of the Alternating Bit Protocol by hand, we trust that this processes can be done in a full automatic way and effectively; in fact, the method is currently being implemented as part of the tool VerICS.

We cannot compare our results with existing tools - Bandera, JPF2 - since both of them analyse the bytecode of a Java program, but not the Java program itself as we do.

References

[1] VerICS. http://verics.ipipan.waw.pl/verics/.

[2] R. Alur. Timed Automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[3] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, J. Mantovani, S. Moedersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 281–285, Edinburgh, Scotland, UK, 2005. Springer-Verlag.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of the 11th International Computer Aided Verification Conference (CAV'99)*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, 1999.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[6] J. Corbett, M. Dwyer, J. Hatcliff, Robby C. Pasareanu, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering(ICSE '00)*, pages 439–448, New York, NY, USA, 2000. ACM Press.

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

[8] A. Doroś, A. Janowska, and P. Janowski. From specification languages to Timed Automata. In *Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'02)*, volume 161(1) of *Informatik-Berichte*, pages 117–128. Humboldt University, 2002.

[9] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability (SAT) problem: a survey. In *Satisfiability Problem: Theory and Applications*, volume 35 of *Discrete Mathematics and Theoretical Computer Science(DIMASC)*, pages 19–152. American Mathematical Society, 1996.

[10] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, V2(4):366–381, March 2000.

[11] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[12] G. J. Holzmann. The model checker SPIN. *IEEE transaction on software engineering*, 23(5):279–295, 1997.

[13] ISO/IEC 9074(E), Estelle - a formal description technique based on an extended state-transition model. International Standards Organization, 1997.

[14] A. Janowska and P. Janowski. Slicing of timed automata with discrete data. *Fundamenta Informaticae*, 72(1-3):181–195, 2006.

[15] M. Kacprzak, A. Lomuscio, and W. Penczek. From bounded to unbounded model checking for temporal epistemic logic. *Fundamenta Informaticae*, 63(2,3):221–240, 2004.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[17] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.

[18] C. Pasareanu and W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proceedings of SPIN'04*, volume 2989 of *LNCS*, pages 164–181. Springer-Verlag, 2004.

[19] W. Penczek, B. Woźna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.

[20] P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[21] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[22] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)*, page 13. IBM Press, 1999.

[23] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE)*, September 2000.

[24] A. Zbrzezny and A. Pólrola. Sat-based reachability checking for timed automata with discrete data. *Fundamenta Informaticae*, 79(3–4):579–593, 2007.