

# Towards verification of Java programs in $\sqrt{erICS}^*$

Andrzej Zbrzezny<sup>1</sup>, Bożena Woźna<sup>1</sup>,  
Maciej Orzechowski<sup>1</sup> and Franco Raimondi<sup>2</sup>

<sup>1</sup> IMCS, Jan Długosz University of Częstochowa  
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland  
email: {m.orzechowski,b.wozna,a.zbrzezny}@ajd.czyst.pl

<sup>2</sup> DCS, University College London  
Gower Street, London WC1E 6BT, UK  
email: F.Raimondi@cs.ucl.ac.uk

**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 translation from a subset of the Java programming language to Timed Automata with Discrete Data. In addition, we show how to use the translator 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.

For the last twenty years model checking has become 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 year.

---

\* The authors acknowledge partial support from the Ministry of Science and Higher Education under grant number 3T11C 01128 and from the EU project PLASTIC.

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)[7] or directly into a formalism called Timed Automata with Discrete Data (TADD) [14, 23]. 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 MINISAT [8], 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.

Verifying programs written in programming languages like Java or C/C++ 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 software poses for model checking, we have developed a preliminary Java to TADD translator that will become a part of the tool VerICS, which, as one can see from the architecture presented above, does not support verification of Java programs yet. Therefore, the main aim of the present paper is to describe this translator, which will allow for verification of Java programs in VerICS. To support the claim, in addition, we show how to use our translator together with the 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 translations of a subset of Java to TADD. Finally, we test the translation on the alternating bit protocol.

## 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, 21, 18] and the second one is called Bandera [5].

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 [21, 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.

The papers [6, 5] provide an overview of the Bandera tool. In particular, they describe a theory of translating Java programs into transition models, making use of a static pointer analysis to aid virtual coarsening, which reduces the size of the models. The Bandera tool is designed to be a bridge between a non-finite-state software system expressed as Java source code and tools like NuSMV and SPIN.

### 3 Translation

This section describes a prototype translator of Java programs into Timed Automata with Discrete Data (TADD). These automata are standard diagonal timed automata [22] augmented to include integer variables over which a standard arithmetic and Boolean expressions can be defined. Moreover these automata take as an input a set of initialised integer variables and a set of propositional variables true at particular states. Since each real Java software is written as a set of communicating processes (classes), it is reasonable to translate classes of the software into a net of TADDs that run in parallel, communicate with each other via shared variables and perform transitions with shared labels synchronously; we assume here a standard definition of parallel composition [2].

We start by describing our running example: the *Alternating Bit Protocol* (ABP), a well-known communication protocol that is often used as a test case for automated verification tools.

### 3.1 Alternating Bit Protocol

The protocol involves three active components: a sender, a receiver and a bidirectional communication channel. Each message that is sent from the sender to the receiver goes through an unreliable communication channel and contains a data part together with a one-bit identifier.

```
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();
    }
}
```

**Fig. 1.** Java source code of the main class

```
import java.util.Random;
public class Sender implements Runnable {
    private LossyChannel channel;
    public Sender(LossyChannel channel) {
        this.channel = channel;
    }
    public void run() {
        boolean protocolBit = false;
        Random random = new Random();
        while (true) {
            if (protocolBit != channel.getAckBit()) {
                channel.put(protocolBit);
            } else {protocolBit = !protocolBit;}
            try {Thread.sleep(random.nextInt(1500));}
            catch (InterruptedException e) {}
        }
    }
}
```

**Fig. 2.** Java source code of Sender

The protocol works in the following way. The sender reads a message at his input and extends it with a control bit to form a frame. Then it starts sending the frame to the receiver, which is initially silent. The sending of the frame 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 frame with the control bit that matches its internal control bit, the message in the frame is sent to the output port, and an acknowledgement is sent to the sender. Then the receiver flips his internal control bit and waits for another frame.

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

Figures 1, 3, 2 and 4 show a Java source code of ABP, which will then be translated into a network of TADDs.

```
import java.util.Random;
public class Receiver implements Runnable {
    private LossyChannel channel;
    public Receiver(LossyChannel channel) {
        this.channel = channel;
    }
    public void run() {
        Random random = new Random();
        while (true) {
            boolean protocolBit = channel.get();
            channel.putAckBit(protocolBit);
            try {Thread.sleep(random.nextInt(1500));}
            catch (InterruptedException e) {}
        }
    }
}
```

**Fig. 3.** Java source code of Receiver

### 3.2 A Translation of Java programs to TADD

The subset of Java, which we use in our translator contains: definitions of integer variables, standard programming language constructs like assignment statements, conditional statements and loops, definitions of classes, objects, methods and threads (we allow to create threads in the main class only), synchronized methods, and the methods `wait()`, `notify()`, `sleep()` and `random()`.

In order to translate the Java code into timed automata with discrete data, we proceed in the following way. First we partition the input Java code into several modules that contain respectively the application class, thread classes and classes that do not extend the thread classes. Then, we parse each module in accordance with the Java subset we have chosen (this is currently performed by hand), and we start a step-by-step analysis of the application class.

The translation of the application class consists in:

- introducing integer variables for all the variable declarations of the type *int* and *Boolean*;
- introducing a set of integer variables for each Java declaration of the type *className variable*. These variables represent the fields of the object *variable*, also those inherited from the base classes. For example, the declaration *LossyChannel channel* introduces the following four integer variables: *channel\_protocolBit*, *channel\_channelEmpty*, *channel\_ackBit* and *channel\_random*.
- producing one TADD for each thread.

```

import java.util.Random;
public class LossyChannel {
    private boolean protocolBit, channelEmpty=true, ackBit=true;
    private Random random;
    public LossyChannel() {random = new Random();}
    public synchronized boolean getAckBit() {
        notify(); return ackBit;
    }
    public synchronized void putAckBit(boolean ackBit) {
        this.ackBit = ackBit;
        int ignoreBit = random.nextInt(5);
        if (ignoreBit > 2) {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){}}
        this.protocolBit = protocolBit;
        channelEmpty = false;
        int ignorePacket = random.nextInt(5);
        if (ignorePacket > 2) {channelEmpty = true;}
        notify();
    }
}

```

**Fig. 4.** Java source code of the communication channel

All the variables introduced above are shared by all the resulting automata.

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 translate *run* methods according to the following scheme:

- If there is an assignment statement of the form `variable = expr;`, then we produce an automaton as it is shown in Figure 5a.
- 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 5b).

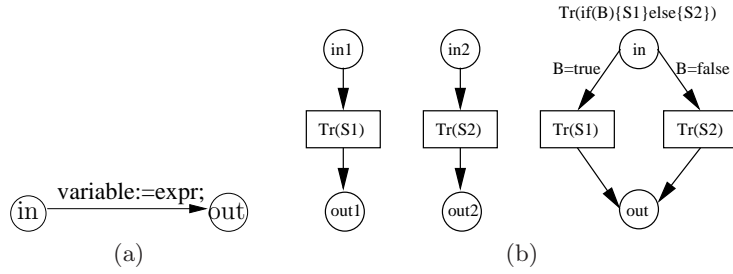


Fig. 5. A translation of: (a) an assignment statement; (b) a conditional statement.

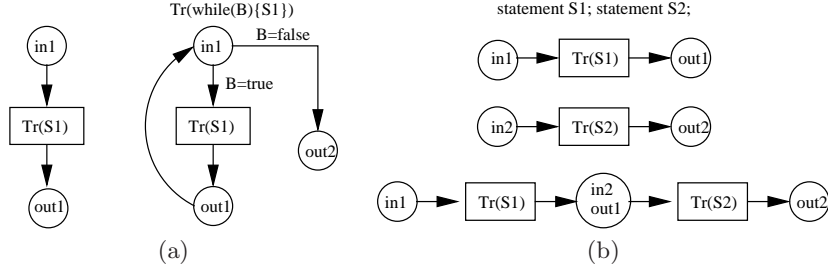
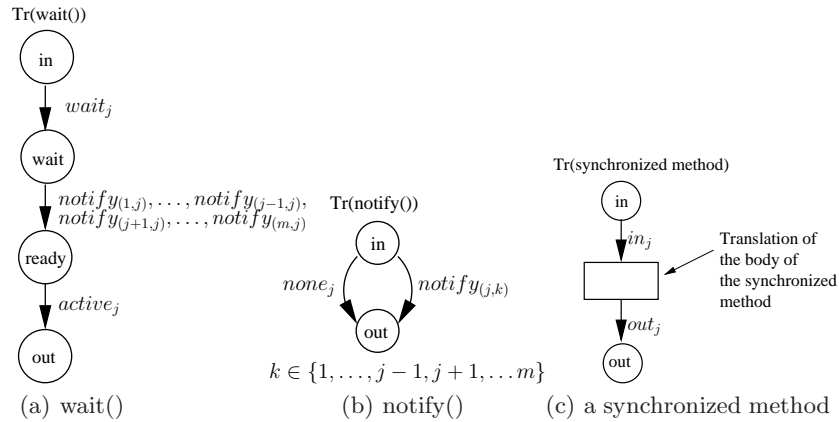


Fig. 6. A translation of: (a) a loop statement; (b) concatenation of two statements.

- 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 6(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 a 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 6(b)).
- If there is a method `wait()`, then the translation is as shown in Figure 7(a), that is, we have an automaton of four locations with  $(m - 1) + 2$  transitions, where  $m$  is the number of threads. The first transition is labelled by action  $wait_j$  which denotes the fact that a 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  $notify_{(1,j)}, \dots, notify_{(j-1,j)}, notify_{(j+1,j)}, \dots, notify_{(m,j)}$ ; 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 thread in the ready state gets access

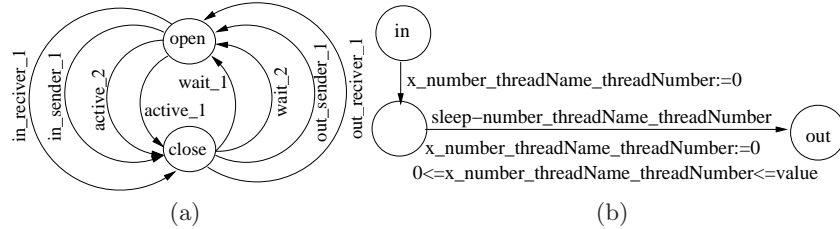
to the CPU, it gets converted to the running state. This is done by invoking a synchronized action  $active_j$ , which will close the semaphore.



**Fig. 7.** A translation of methods  $wait()$ ,  $notify()$  and a synchronized method for thread  $j$ .

- If there is a method  $notify()$  that wakes up an arbitrary chosen thread from all the threads waiting on the object's lock, then the translation is as shown in Figure 7(b). Namely, we have an automaton of two locations with two transitions labelled by  $notify_{(j,k)}$ , for  $k \in \{1, \dots, j-1, j+1, \dots, m\}$ , 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*. If there are no waiting threads, the  $notify()$  method has no effect; this is realised by the local action  $none_j$ .
- A translation of synchronized methods is shown in Figure 7(c). As one can see, 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 the entrance to the method, and *out* denotes the exit from the method. The synchronization process is realised by using a binary semaphore.  
Note that in the translation process the labels  $in_j$  and  $out_j$  will have respectively the following general form:  $in\_threadName\_threadNumber$  (for example  $in\_sender\_1$ ) and  $out\_threadName\_threadNumber$  (for example  $out\_sender\_1$ ) (see the resulting automata for the alternating bit problem).
- A translation of the method  $sleep()$  consists in introducing a new clock with the general name  $x\_number\_threadName\_threadNumber$  (precisely, one clock per one instance of the method  $sleep()$  in a thread) and producing an automaton that consists of three locations and two transitions (see Figure 8(b)). The first transition must be decorated by the reset opera-

- tion of the form  $x\_number\_threadName\_threadNumber := 0$ ; the second transition must be decorated by: (1) an action with general name  $sleep - number\_threadName\_threadNumber$  (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 translation 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  $i \in \{0, \dots, n - 1\}$ .
  - Values that are returned by methods of a non-void type are stored in integer variables with the general name  $className\_threadNumber\_methodName$ .
  - 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  $in\_className\_methodName$ .



**Fig. 8.** (a) An automaton for semaphore; (b) a translation of the method  $sleep()$ .

Given the above it is easy to infer the network of timed automata with discrete data that encode Alternating Bit Protocol; this is shown in Figures 9, 10 and 8.

## 4 Experimental results

In this section we show how to use our translator 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 Figures 1, 2, 3 and 4 in Section 3. All the experiments have been performed on a computer equipped with the processor

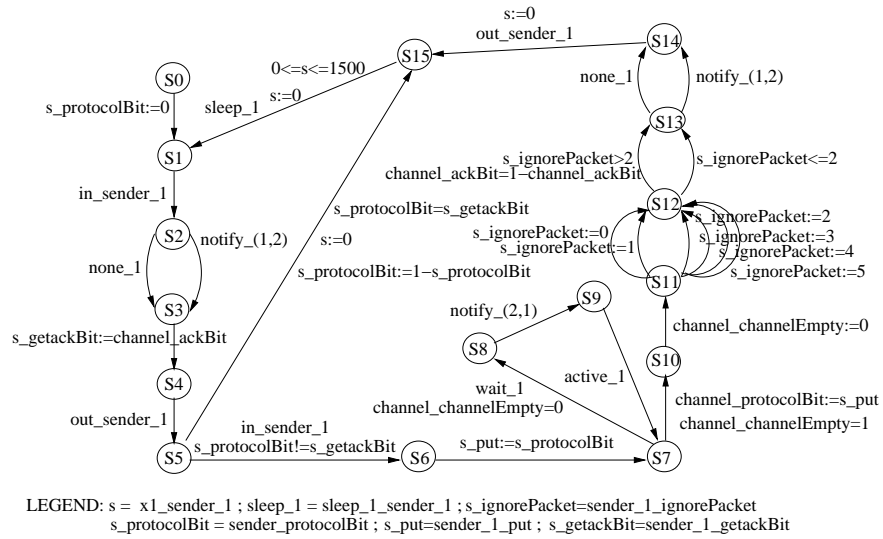


Fig. 9. An automaton for Sender

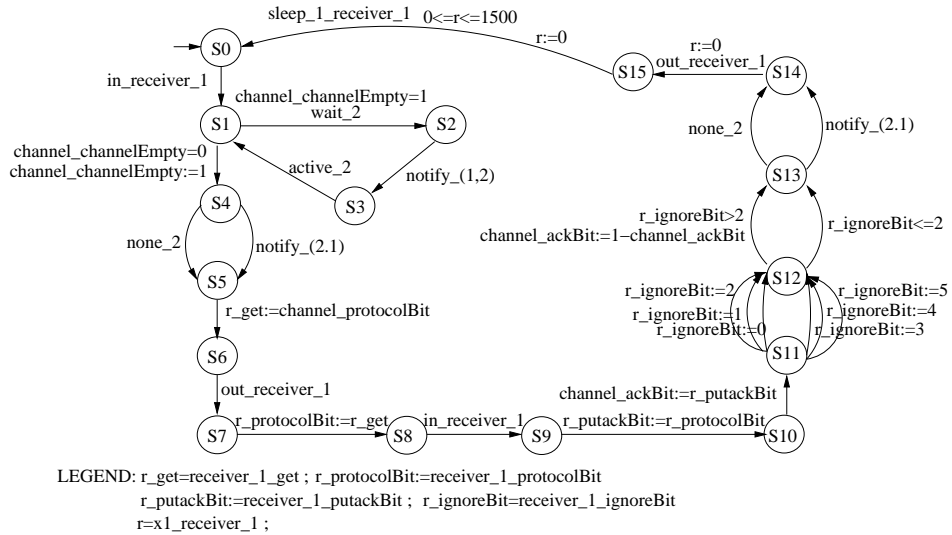


Fig. 10. An automaton for Receiver

AMD Athlon XP 1800 (1544 MHz), 1GB main memory and the operating system Linux 2.6.21.

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 zero (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; indeed, none of the threads can do anything. To find the cause of this behaviour, we have checked this modified Java code of ABP 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  $EFtrue$  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>	<0,0/4>	<0,0/4>
1:	<0,0,0>	<0,0,0,0,0,1,0,0,1,0,0>	<1471,3/4>	<1471,3/4>
2:	<1,0,0>	<0,0,0,0,0,1,0,0,1,0,0>	<1471,3/4>	<1471,3/4>
3:	<1,0,0>	<0,0,0,0,0,1,0,0,1,0,0>	<2069,2/4>	<2069,2/4>
4:	<1,1,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2069,2/4>	<2069,2/4>
5:	<1,1,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2312,0/4>	<2312,0/4>
6:	<1,2,0>	<0,0,0,0,0,1,0,0,1,0,0>	<2312,0/4>	<2312,0/4>
7:	<1,2,0>	<0,0,0,0,0,1,0,0,1,0,0>	<2817,0/4>	<2817,0/4>
8:	<2,2,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2817,0/4>	<2817,0/4>
9:	<2,2,1>	<0,0,0,0,0,1,0,0,1,0,0>	<3516,0/4>	<3516,0/4>
10:	<3,3,1>	<0,0,0,0,0,1,0,0,1,0,0>	<3516,0/4>	<3516,0/4>
11:	<3,3,1>	<0,0,0,0,0,1,0,0,1,0,0>	<0,1/4>	<0,1/4>
12:	<4,3,1>	<0,0,0,0,0,1,0,0,1,0,0>	<0,1/4>	<0,1/4>
13:	<4,3,1>	<0,0,0,0,0,1,0,0,1,0,0>	<514,1/4>	<514,1/4>
14:	<5,3,0>	<0,0,0,0,0,1,0,0,1,0,0>	<514,1/4>	<514,1/4>
15:	<5,3,0>	<0,0,0,0,0,1,0,0,1,0,0>	<2016,0/4>	<2016,0/4>
16:	<5,1,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2016,0/4>	<2016,0/4>
17:	<5,1,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2016,0/4>	<2016,0/4>
18:	<5,2,0>	<0,0,0,0,0,1,0,0,1,0,0>	<2016,0/4>	<2016,0/4>
19:	<5,2,0>	<0,0,0,0,0,1,0,0,1,0,0>	<2180,2/4>	<2180,2/4>
20:	<6,2,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2180,2/4>	<2180,2/4>
21:	<6,2,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2887,1/4>	<2887,1/4>
22:	<7,2,1>	<0,0,0,0,0,1,0,0,1,0,0>	<2887,1/4>	<2887,1/4>
23:	<7,2,1>	<0,0,0,0,0,1,0,0,1,0,0>	<3296,2/4>	<3296,2/4>
24:	<8,2,0>	<0,0,0,0,0,1,0,0,1,0,0>	<3296,2/4>	<3296,2/4>

Table 1: The witness

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\_ignorePacket, and the last two columns show the values of clocks for Sender and Receiver, respectively.

k	variables	clauses	BMC sec	BMC MB	MiniSAT sec	MiniSAT MB	SAT
0	439	637	0.0	1.5	0.0	3.5	YES
4	6230	16756	0.8	2.2	0.1	4.4	YES
8	12022	32878	1.5	2.9	0.1	5.2	YES
12	17814	49000	2.2	3.6	0.3	6.0	YES
16	23606	65122	3.3	4.2	0.7	7.1	YES
20	29398	81244	3.5	5.0	1.6	7.9	YES
24	35190	97366	4.7	5.6	0.9	8.8	YES
26	38086	105427	5.2	6.0	0.4	8.9	NO
	In total:		35.8	6.0	7.3	8.9	

Table 2: Property  $EFtrue$ .

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 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>	<0,0/4>	<0,0/4>
1:	<0,0,0>	<1,0,0,0,0,1,0,0,1,0,0>	<877,1/4>	<877,1/4>
2:	<1,0,0>	<1,0,0,0,0,1,0,0,1,0,0>	<877,1/4>	<877,1/4>
3:	<1,0,0>	<1,0,0,0,0,1,0,0,1,0,0>	<2102,2/4>	<2102,2/4>
4:	<2,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<2102,2/4>	<2102,2/4>
5:	<2,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<3186,0/4>	<3186,0/4>
6:	<3,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<3186,0/4>	<3186,0/4>
7:	<3,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<273,2/4>	<273,2/4>
8:	<4,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<273,2/4>	<273,2/4>
9:	<4,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<941,3/4>	<941,3/4>
10:	<5,0,0>	<1,0,0,0,0,1,0,0,1,0,0>	<941,3/4>	<941,3/4>
11:	<5,0,0>	<1,0,0,0,0,1,0,0,1,0,0>	<2440,0/4>	<2440,0/4>
12:	<6,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<2440,0/4>	<2440,0/4>
13:	<6,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<3637,3/4>	<3637,3/4>
14:	<7,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<3637,3/4>	<3637,3/4>
15:	<7,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<40,1/4>	<40,1/4>
16:	<10,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<40,1/4>	<40,1/4>
17:	<10,0,1>	<1,0,0,0,0,1,0,0,1,0,0>	<1331,1/4>	<1331,1/4>
18:	<11,0,1>	<0,0,0,0,0,1,0,0,1,0,0>	<1331,1/4>	<1331,1/4>
19:	<11,0,1>	<0,0,0,0,0,1,0,0,1,0,0>	<1486,3/4>	<1486,3/4>
20:	<12,0,1>	<0,0,0,0,0,1,0,0,1,0,1>	<1486,3/4>	<1486,3/4>
21:	<12,0,1>	<0,0,0,0,0,1,0,0,1,0,1>	<1817,0/4>	<1817,0/4>
22:	<13,0,1>	<0,0,0,0,0,1,0,0,1,0,1>	<1817,0/4>	<1817,0/4>

23:	<13,0,1>	<0,0,0,0,0,1,0,0,1,0,1>	<2211,0/4>	<2211,0/4>
24:	<14,0,1>	<0,0,0,0,0,1,0,0,1,0,1>	<2211,0/4>	<2211,0/4>
25:	<14,0,1>	<0,0,0,0,0,1,0,0,1,0,1>	<2805,2/4>	<2805,2/4>
26:	<15,0,0>	<0,0,0,0,0,1,0,0,1,0,1>	<2805,2/4>	<2805,2/4>
27:	<15,0,0>	<0,0,0,0,0,1,0,0,1,0,1>	<3713,0/4>	<3713,0/4>
28:	<15,1,1>	<0,0,0,0,0,1,0,0,1,0,1>	<3713,0/4>	<3713,0/4>
29:	<15,1,1>	<0,0,0,0,0,1,0,0,1,0,1>	<112,2/4>	<112,2/4>
30:	<1,1,1>	<0,0,0,0,0,1,0,0,1,0,1>	<112,2/4>	<112,2/4>
31:	<1,1,1>	<0,0,0,0,0,1,0,0,1,0,1>	<281,2/4>	<281,2/4>
32:	<1,2,0>	<0,0,0,0,0,1,0,0,1,0,1>	<281,2/4>	<281,2/4>
33:	<1,2,0>	<0,0,0,0,0,1,0,0,1,0,1>	<1540,3/4>	<1540,3/4>
34:	<2,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<1540,3/4>	<1540,3/4>
35:	<2,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<2816,0/4>	<2816,0/4>
36:	<3,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<2816,0/4>	<2816,0/4>
37:	<3,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<75,2/4>	<75,2/4>
38:	<4,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<75,2/4>	<75,2/4>
39:	<4,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<1016,0/4>	<1016,0/4>
40:	<5,2,0>	<0,0,0,0,0,1,0,0,1,0,1>	<1016,0/4>	<1016,0/4>
41:	<5,2,0>	<0,0,0,0,0,1,0,0,1,0,1>	<1277,3/4>	<1277,3/4>
42:	<6,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<1277,3/4>	<1277,3/4>
43:	<6,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<2049,0/4>	<2049,0/4>
44:	<7,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<2049,0/4>	<2049,0/4>
45:	<7,2,1>	<0,0,0,0,0,1,0,0,1,0,1>	<3008,0/4>	<3008,0/4>
46:	<8,2,0>	<0,0,0,0,0,1,0,0,1,0,1>	<3008,0/4>	<3008,0/4>

Table 3: The witness

k	variables	clauses	BMC sec	BMC MB	MiniSAT sec	MiniSAT MB	SAT
0	445	655	0.1	1.5	0.0	3.5	NO
4	6244	16798	0.7	2.1	0.0	4.4	NO
8	12044	32944	1.4	2.9	0.1	5.2	NO
12	17844	49090	2.1	3.6	0.2	6.0	NO
16	23644	65236	2.6	4.2	0.4	6.9	NO
20	29444	81382	3.2	5.0	0.6	7.7	NO
24	35244	97528	4.0	5.6	1.5	8.7	NO
28	41044	113674	4.5	6.3	1.9	9.7	NO
32	46844	129820	5.3	7.0	1.7	10.8	NO
36	52644	145966	6.4	7.7	3.1	11.3	NO
40	58444	162112	6.4	8.3	4.1	11.9	NO
44	64244	178258	7.5	9.1	6.3	13.2	NO
46	67144	186331	7.5	9.4	11.3	13.8	YES
	In total:	91,7	9.4	47,9	13.8		

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

## 5 Summary

In this paper we have proposed a preliminary Java to Timed Automata with Discrete Data translator, which is going to be a part of the tool VerICS. We have also provided preliminary experimental results. We are currently working on examples to compare the performance of our method with Bandera and JPF.

**Acknowledgement:** The authors wish to thank an anonymous reviewer for constructive comments on this paper.

## References

1. VerICS. <http://verics.ipipan.waw.pl/verics/>.
2. R. Alur. Timed Automata. In *Proc. of CAV'99*, volume 1633 of *LNCS*, pp. 8–22. Springer-Verlag, 1999.

3. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hanks 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 *Proc. of CAV'05*, volume 3576 of *LNCS*, 2005.
4. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. of CAV'99*, volume 1633 of *LNCS*, pp. 495–499. Springer-Verlag, 1999.
5. J. Corbett, M. Dwyer, J. Hatcliff, Robby C. Pasareanu, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. of ICSE '00*, pp. 439–448. ACM Press, 2000.
6. James C. Corbett. Constructing compact models of concurrent java programs. In *International Symposium on Software Testing and Analysis*, pp. 1–10, 1998.
7. A. Doroś, A. Janowska, and P. Janowski. From specification languages to Timed Automata. In *Proc. of CS&P'02*, volume 161(1) of *Informatik-Berichte*, pp. 117–128. Humboldt University, 2002.
8. N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. of SAT'03*, volume 2919 of *LNCS*, pp. 502–518. Springer Berlin/Heidelberg, 2004.
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 *DIMASC*, pp. 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*, 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 *Proc. of DAC'01*, pp. 530–535, June 2001.
18. C. Pasareanu and W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pp. 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. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of ASE'00*, September 2000.
22. A. Zbrzezny. SAT-based reachability checking for timed automata with diagonal constraints. *Fundamenta Informaticae*, 67(1-3):303–322, 2005.
23. A. Zbrzezny and A. Pólrola. Sat-based reachability checking for timed automata with discrete data. *Fundamenta Informaticae*, 79(3–4):579–593, 2007.