Catholic University in Ružomberok
Scientific Issues, Mathematica II, Ružomberok 2009

Verification of Java programs using networks of finite automata with discrete data.

Bożena Woźna, Andrzej Zbrzezny

Institute of Mathematics and Computer Science
Jan Długosz University of Czstochowa
al. Armii Krajowej 13/15, 42-200 Czstochowa, Poland
e-mail: {b.wozna, a.zbrzezny}@ajd.czest.pl

Abstract. In the paper we show that automatic verification of Java programs is feasible. In particular, we show that for a given Java program it is possible to build a network of Finite Automata with Discrete Data (FADDs) that models the behaviour of the program under consideration. Such a model enables to verify some interesting properties of the considered program by means of tools like `Uppaal` and `VerICS`.

1 Introduction

The model checking tools like `VerICS` [4] or `Uppaal` [1] accept a description of a network of *finite automata with discrete data* (FADDs) [7] as input. This FADD formalism is accurate enough to detect concurrency errors and yet abstract enough to make model checking tractable. So, in order to automatically verify Java programs one has to bridge the semantic gap between a non-finite-state system expressed as a Java source code, and those tools input languages. This requires the application of sophisticated transformation techniques, and one of such methods is shown in this paper; it has been implemented as a `J2FADD` module of the tool `VerICS`.

We illustrate our approach by presenting the automata models of the two well known solutions (written in Java) of the dining philosophers problem

(DPP) - an illustrative example of a common computing problem in concurrency. We also provide some experimental results that help to evaluate our method.

The subset of Java that can be translated to FADDs contains: definitions of integer variables, standard programming language constructs like assignments, expressions with most operators, conditional statements and loops (`for`, `while`, `do while`), instructions `break` and `continue` without labels, definitions of classes, (static) objects, constructors and methods, static and non–static methods, and synchronisation methods and blocks. There are also standard thread creation constructs recognised and special methods: `Thread.wait()`, `Thread.notify()`, and `Random.nextInt(int)`, the methods `Thread.join(Thread)` and `Thread.notifyAll()`. For more details we refer to [8].

To the best of our knowledge, there are only two other model checkers for verification of Java programs: *JavaPathFinder* [5] and *Bandera* [2]. However, both of them operate on the Java bytecode, and do not produce a low-level models like finite automata with discrete data.

2 Dining Philosophers Problem (DPP)

In this section we show FADD models of the two well known solutions of DPP, and we model check it by means of the tools: `Uppaal`, `VerICS`. In particular, we search for deadlocks, which can be defined formally as follows [6]: *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

FADDs are standard 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.

2.1 Problem description

The description of the dining philosophers problem (DPP) we provide below is based on that in [3]. Consider n ($n \geq 2$) philosophers. Each philosopher has a room in which he engages in his professional activity of thinking. There is also a common dining room, furnished with a circular table, surrounded by n chairs, each labelled by the name of the philosopher who is to sit in it. On the left of each philosopher there is a fork, and in the centre stands a large bowl of spaghetti, which is constantly replenished. Whenever a philosopher eats he has to use both forks, the one on the left and the other on the right of his plate. A philosopher is expected to spend most of his time thinking,

but when he feels hungry, he goes to the dining room, sits down on his own chair, and picks up the fork on his left provided it is not used by the other philosopher. If the other philosopher uses it, he just has to wait until the fork is available. Then the philosopher tries pick up the fork on his right. When a philosopher has finished he puts down both his forks, exits dining-room and continues thinking.

2.2 Possible solutions

We have implemented a possible solution of the DPP problem that could lead to a deadlock (see Listing 1). The deadlock can happen, if every philosopher sits down on his own chair at the same time and picks up his left fork. Then all forks are locked and none of the philosophers can successfully pick up his right fork. As a result, every philosopher waits for his right fork that is currently being locked by his right neighbour, and hence a deadlock occurs. The results for the deadlock property are in Table 1.

Tools	**No. Ph**	**sec.**	**MB**
J2FADD + VerICS	5	12 217	279.2
J2FADD + Uppaal	60	1.16	41.9

Table 1: Dining Philosophers. Deadlock.

Assume now another solution for DPP (see Listing 2), where there is a lackey who ensures that at most $n-1$ philosophers can be present in the dining room at the same time. This lackey ensures that no deadlock is possible (see Table 2 for the results).

Tools	**No. Ph**	**sec.**	**MB**
J2TADD + Uppaal	5	52.22	418.7
J2TADD + Uppaal	6	-	-

Table 2: Dining Philosophers. Absence of deadlocks.

All of the experiments have been performed on a computer equipped with the processor Intel Core 2 Duo (2 GHz), 2 GB main memory and the operating system Linux.

2.3 Automata models for the solution without lackey

In this section we present an automata model for DPP with three philosophers (see Listing 1). The model consists of a network of the following six FADDs: one FADD for each `Philosopher`, and one FADD for each `Fork`. On Figure 2 we show an automaton for `Philosopher` $p0$, and on Figure 1 an automaton for the `Fork` $fork0$. The automata for philosophers $p1$ and $p2$ are analogous

```
public class College3 {
  public static void main(String args []) {
    Fork fork0 = new Fork(false);    Fork fork1 = new Fork(false);
    Fork fork2 = new Fork(false);
    Philosopher p0 = new Philosopher(0, fork0, fork1);
    Philosopher p1 = new Philosopher(1, fork1, fork2);
    Philosopher p2 = new Philosopher(2, fork2, fork0);
    (new Thread(p0)).start(); (new Thread(p1)).start();
    (new Thread(p2)).start();
  }
}
class Fork {
  private boolean unavailable;
  public Fork(boolean unavailable) {this.unavailable = unavailable;}
  public synchronized void acquire() {
    while (unavailable) {
        try {wait();} catch(InterruptedException e){}
    }
    unavailable = true;
  }
  public synchronized void release() {unavailable = false; notify();}
}
class Philosopher implements Runnable {
  private int nr;
  private Fork left, right;
  public Philosopher(int nr, Fork left, Fork right) {
    this.nr = nr;   this.left = left;  this.right = right;
  }
  public void run() {
      while (true) {
         left.acquire(); right.acquire();
         right.release(); left.release();
      }
  }
}
```

Listing 1: Java source code of DPP (3 Philosophers). The main class.

to that of `Philosopher` $p0$, and the automata for forks $fork1$ and $fork2$ are analogous to that of `Fork` $fork0$.

The automaton for a philosopher reflects the method `run` of the class `Philosopher`. Namely, the fragment consisting of the locations: $v11$, $v12$, $v13$, $v10$, $v9$, $v14$, $v15$, $v16$ reflects the instruction `left.acquire()`; the fragment consisting of the locations: $v16$, $v17$, $v18$, $v8$, $v7$, $v19$, $v20$, $v21$ reflects the instruction `right.acquire()`; the fragment consisting of the locations: $v21$, $v22$, $v23$, $v24$, $v25$ reflects the instruction `right.release()`; and the fragment consisting of the locations: $v25$, $v26$, $v27$, $v28$, $v11$ reflects the instruction `left.release()`.

The automaton for a fork consists of two locations, operates as a binary semaphore, and implements critical sections.

```
public class College3L {
  public static void main(String args []) {
    Fork f0 = new Fork(false); Fork f1 = new Fork(false);
    Fork f2 = new Fork(false); Lackey s = new Lackey(2);
    Philosopher  p0 = new Philosopher (0,f0,f1,s);
    Philosopher  p1 = new Philosopher (1,f1,f2,s);
    Philosopher  p2 = new Philosopher (2,f2,f0,s);
    (new Thread(p0)).start(); (new Thread(p1)).start();
    (new Thread(p2)).start();
  }
}
class Fork {
  private boolean unavailable;
  public Fork(boolean unavailable) {  this.unavailable = unavailable; }
  public synchronized void acquire() {
    while (unavailable) { try {wait(); } catch (Exception e){} }
    unavailable = true;
  }
  public synchronized void release() {  unavailable = false; notify(); }
}
class Lackey {
  private int m; private int max;
  public Lackey(int max) {this.max = max;}
  public synchronized void acquire() {
    while (m >= max) { try {wait();} catch (Exception e) {} }  ++m;
  }
  public synchronized void release() { --m; notify(); }
}
class Philosopher  implements Runnable {
  private int nr; private Lackey s;
  private Fork left, right;
  public Philosopher (int nr, Fork left, Fork right, Lackey s) {
    this.nr = nr; this.s = s;  this.left = left; this.right = right;
  }
  public void run() {
    while (true) {
      s.acquire(); left.acquire(); right.acquire();
      right.release(); left.release(); s.release();
    }
  }
}
```

Listing 2: Java source code of DPP (3 Philosophers). The main class.

2.4 Automata models for the solution with lackey

In this section we present an automata model for DPP with lackey and three philosophers (see Listing 2). The model consists of a network of the following seven FADDs: one FADD for each `Philosopher`, one FADD for each `Fork`, and one FADD for the `Lackey`. On Figure 4 we show an automaton for `Philosopher` $p0$, and on Figure 3 automata for the `Fork` $fork0$ and for the `Lackey` s. The automata for philosophers $p1$ and $p2$ are analogous to that of `Philosopher` $p0$, and the automata for forks $fork1$ and $fork2$ are analogous

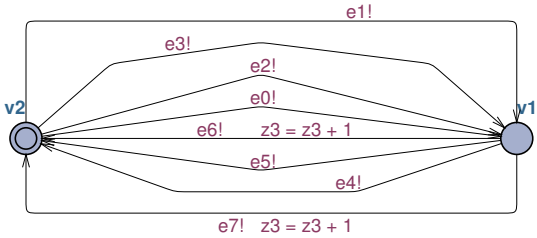


Figure 1: The automaton for `Fork` $fork0$.

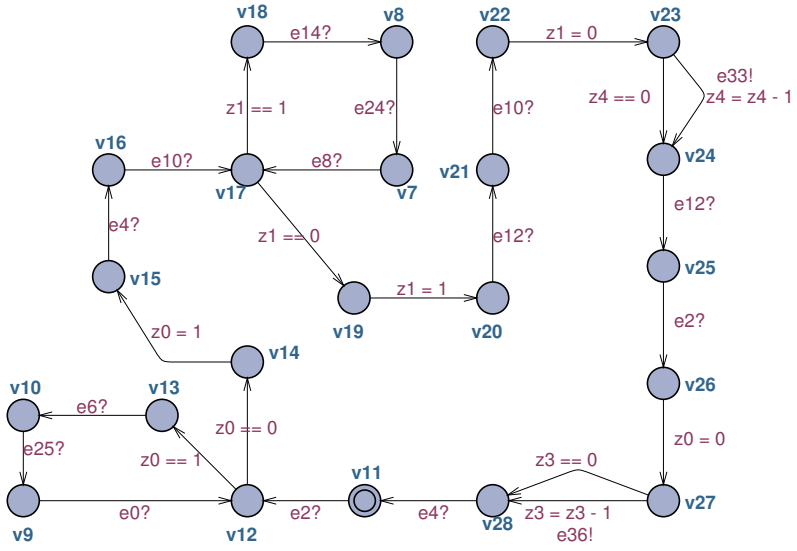


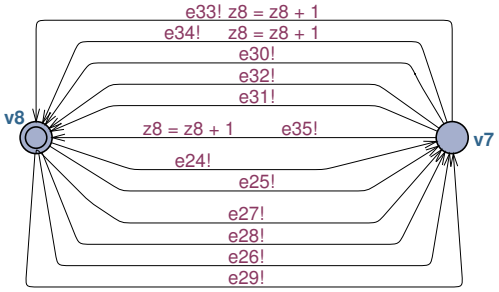
Figure 2: The automaton for `Philosopher` $p0$.

to that of `Fork` $fork0$.

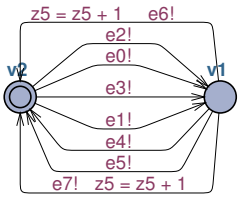
The automaton for a philosopher reflects the method `run` of the class `Philosopher`. Namely, the fragment consisting of the locations: $v15$, $v16$, $v17$, $v14$, $v13$, $v18$, $v19$, $v20$ reflects the instruction `s.acquire()`; the fragment consisting of the locations: $v20$, $v21$, $v22$, $v23$, $v11$, $v12$, $v24$, $v25$ reflects the instruction `left.acquire()`; the fragment consisting of the locations: $v25$, $v26$, $v27$, $v28$, $v10$, $v9$, $v29$, $v30$ reflects the instruction `right.acquire()`; the fragment consisting of the locations: $v30$, $v31$, $v32$, $v33$, $v34$ reflects the instruction `right.release()`; the fragment consisting of the locations: $v34$, $v35$, $v36$, $v37$, $v38$ reflects the instruction `left.release()`; and the fragment consisting of the locations: $v38$, $v39$, $v40$, $v41$, $v15$ reflects the instruction

`s.release()`.

The automaton for a fork consists of two locations, operates as a binary semaphore, and implements locking a fork. The automaton for a lackey consists of two locations, operates as a bounded semaphore, and implements locking the dining room.



The automaton for `Lackey` s;



The automaton for `Fork` $fork0$.

Figure 3:

3 Conclusions

In the paper we have shown that model checking of Java programs by means of tools `Uppaal` and `VerICS` that accept a description of a network of FADDs is feasible. We have presented the method using the dining philosophers problem, a classic multi-process synchronisation problem

The experiments confirm that our approach provides a valuable aid for Java software verification.

References

[1] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of Uppaal. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*, 1998.

[2] 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.

[3] C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall, 1985.

[4] A. Niewiadomski W. Penczek A. Pólrola M. Szreter B. Woźna M. Kacprzak, W. Nabialek and A. Zbrzezny. VerICS 2007 - a model

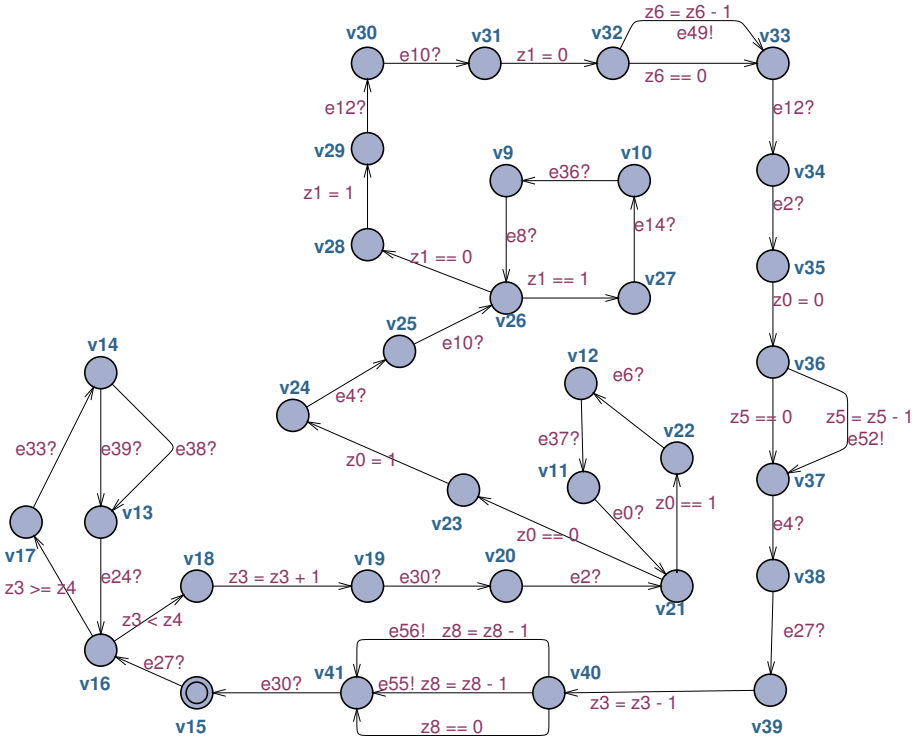


Figure 4: The automaton for `Philosopher` $p0$.

checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1-4):313–328, 2008.

[5] 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.

[6] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Vrije University, Amsterdam, The Netherlands, 2/e edition, 2001.

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

[8] A. Zbrzezny and B. Woźna. Towards verification of Java programs in VerICS. *Fundamenta Informaticae*, 85(1-4):533–548, 2008.