Jan Długosz University in Częstochowa
Scientific Issues, Mathematics XIV, Częstochowa 2009

MODEL CHECKING OF JAVA PROGRAMS USING NETWORKS OF FADDS

Bożena Woźna, Andrzej Zbrzezny

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

Abstract
In the paper we present the current theoretical base of the `J2FADD` tool, which translates a Java program to a network of finite automata with discrite data (FADDs). The reason for building the tool is that to model check a concurrent program written in Java by means of the tools like `Uppaal` or `VerICS` (the module `VerICS`), an automata model of the Java program must be build first. This is because these tools verify only systems modeled as networks of automata, in particular, systems modeled as networks of FADDs. We also make an attempt to evaluate the `J2FADD` tool by comparison of it with the two well known Java verification tools: `Bandera` and `Java PathFinder`.

1. Introduction

Developing and writing multi-threaded programs in Java, one of the modern programming languages, is not an easy task. But even more hard it is to detect errors due to multi-threading. This is because subtle program errors can result from unforeseen interactions among multiple threads and they often depend on the non-deterministic behaviour of the scheduler and the environment. Therefore, it is desirable to provide tools for software developers that automatically detect errors due to multi-threading.

In the last decade various Java verification approaches and tools have been developed. In particular, the following environment for Java verification and testing have been built: `Java PathFinder` (JPF) [11] and `Bandera` [5].

Both tools can model check Java programs on deadlocks and limited Java assertions only, and they analyse the Java bytecode.

Other works related to the model checking of Java programs include [7, 10]. In the paper [10] a SAL (Symbolic Analysis Laboratory, [2] based Java model checker is presented. The tool uses both the SAL intermediate language and the `Soot` compiler framework, and it works as follows. First, a Java program is compiled to the Java bytecode. Then, the resulting bytecode is converted to a `Jimple` code, one of the `Soot` output formalisms. Next, the `Jimple` code is translated to SAL; this should be done by a Jimple to SAL translator, but it is not available on the web. Thus, it is not possible to run and evaluate the [10] Java model checker. Moreover, it seems that the work is not continued since the year 2000.

In the paper [7] a tool Java Formal Analysis (`JavaFAN`) is presented. The tool is based on rewriting logic, implemented in the Maude language, and it supposes to formally analyse multi-threaded Java programs at source code and/or bytecode levels. According to [7] the `JavaFAN` allows for the following types of analysis: `symbolic simulation`, `safety violations` (via BFS search), and `LTL model checking of rewriting theories`. Similarly to the SAL based model checker, `JavaFAN` is not available on the web. Moreover, it seems that the work is not continued since the year 2006.

The model checking tools like `Uppaal` [3] and `VerICS` [1] accept a description of a network of `finite automata with discrete data` (FADDs) as input. Thus, to verify a concurrent system written in Java by means of these tools, first a FADD model of the system must be build, which is accurate enough to detect concurrency errors and yet abstract enough to make model checking tractable.

A year ago we have started developing the `J2FADD` tool that translates a Java program to a network of FADDs. At the very beginning this translator was an implementation of a FADD model of Java programs that was proposed in [16]. Currently, `J2FADD` implements the solution presented in [16] with some major corrections in the translation of the `notify()` method plus lots of new futures that will be described in Section 3. Other words, the paper presents the new theoretical base of the `J2FADD` tool.

Our case study have been done for a Java program that implement the well-known problem of `dining philosophers`; as verification engines we have used `Uppaal` and the BMC module of `VerICS` [6] that has been adapted to work with the `J2FADD` translator. The module implements the Bounded Model Checking method with properties expressible in ECTL [12] and system described as a network of FADDs. We compare our results with the other available Java verification tools, i.e. `JPF` and `Bandera`.

The rest of the paper is organised as follows. In the next section we describe the FADD formalism, and we briefly discuss an architecture of `J2FADD` - a translator of a Java code to a network of FADDs. In section 3 we show the theoretical base of our `J2FADD` tool. Finally, we discuss the dining philosophers problem.

2. Preliminaries

2.1. Finite automata with discrete data

The model-checkers `Uppaal` and `VerICS` are based on the theory of finite automata [9] and their automata modelling language offers an additional feature: bounded integer variables; `Uppaal` and `VerICS` do in-fact accept timed automata with discrete data as its input. The properties to be checked are specified in a subset of CTL (computation tree logic) [4]. In this section we briefly describe **`finite automata with discrete data (FADD)`**, the automata modelling languages of these tools.

A `FADD` is a finite-state machine extended with bounded discrete variables. It uses a discrete-time model where a discrete variable evaluates to an integer value. All the discrete variables are used as in programming languages: they are read, written, and are subject to common arithmetic operations. In both `Uppaal` and `VerICS`, a system is modelled as a network of several FADDs that run in parallel and communicate via shared actions. A state of the system is defined by the locations of all automata and the values of the discrete variables. Every automaton may fire an edge (transition) separately or synchronise with another automaton, which leads to a new state.

Automata for `Uppaal` generated by the `J2FADD` tool use two kinds of synchronisation: `binary` and `broadcast`. In the binary synchronisation one sender c! synchronises with non-deterministically chosen receivers c?. If there are no receivers, then the sender cannot execute the c! action. In the broadcast synchronisation one sender c! can synchronise with an arbitrary number of receivers c?. Any receiver than can synchronise in the current state must do so. If there are no receivers, then the sender can still execute the c! action, i.e. broadcast sending is never blocking. Automata for `VerICS` generated by the `J2FADD` tool use the multi-way synchronisation model, i.e. it requires that each automaton of a given network that contains a transition labelled by a joint action has to execute it.

2.2. The J2FADD tool

The `J2FADD` tool implements the translation of a Java program to a network of FADDs according to the theory discussed in the next section. `J2FADD` is written in JAVA and it can be run as a standalone command line tool on any platform supporting Java 1.5 or later, and can be downloaded from [1]. `J2FADD` accepts a number of options to specify the output format and valuation; all of them can be seen by running the `h` (from help) option.

To implement the above mentioned translation, first a Java code is translated to an internal assembler. This step involves lexical analysis and parsing of the input, semantic check and static analysis. Once the assembler code is generated, two stages remain to convert the assembler to FADDs: `interpreting` and `generating transitions`. The role of the interpreter is to initialise variables, load needed classes, create objects and threads. For the detailed description of the internal assembler and translation from the assembler to a network of FADDs see [13, 14].

3. A FADD model of Java programs

This section describes a translation of a concurrent multi-threaded Java program into a network of FADDs, which has been implemented as a `J2FADD` tool. Each location of the generated FADDs is used to record the current control state 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, or dead). Each transition (or a set of transitions) represents the execution of a Java instruction for some thread. There is one FADD for each instance of a started thread and one FADD for each shared object. In this paper we assume all method calls have been inlined, and because the translator does not detect a statically bounded recursion, direct or indirect recursion is not allowed.

The subset of Java that can be translated to FADDs and has been described in [16] contains: definitions of integer variables, standard programming language constructs like assignments, expressions with most operators, conditional statements and loops (`for`, `while`, `do while`), definitions of classes, objects, constructors and methods, static and non–static methods, and synchronisation methods and blocks. There were also standard thread creation constructs recognised and special methods: `Thread.wait()`, `Thread.notify()`, and `Random.nextInt(int)`. The currently handled subset of Java contains additionally: instructions `break` and `continue` without labels, the methods `Thread.join(Thread)` and `Thread.notifyAll()`, nested synchronised methods and blocks (this involves a new translation for the `wait()` and `notify()` methods), static objects and comment tags (annotations); in the sequel we present FADD models of the new elements of the chosen subset of Java.

Threads. In Java there are two ways of creating threads: implementing an interface `Runnable` or extending the class `Thread`. Our translation handles both methods.

We produce one FADD for each instance of a started thread. Namely, when a thread is started by calling its `start()` method, one FADD that is the translation of the body of the run method is produced.

Synchronized methods and blocks. Java supports mutually exclusive access to objects via `synchronized methods` and `synchronized blocks`, that is, if a thread calls a synchronized method on an object, then no other thread can call synchronized methods on the same object as long as the first thread has not terminated its call or has not suspended its execution. Other words, when a thread executes a synchronized method, it must acquire the lock of the object (every Java object has an implicit lock) on which the method has been invoked before executing the body of the method. The lock is released, when the body of the method is exited. If the lock is unavailable, the thread will be blocked until the lock is released.

In the FADD formalism mutual exclusion between threads is realised as follows. Assuming that N is a number of objects on which synchronized methods are invoked, we introduce N binary semaphores as shown in Figure 1 and N special variables $locVar_1, \ldots, locVar_N$, one per each semaphore, initialised with the value zero; the $locVar_i$ variable is a counter that keeps track of the number of threads blocked on object i due to a call of a `wait()` method. Then, we build a two state automaton with two special transitions that are labelled with synchronized actions in_{i_x} and out_{i_x}, respectively, where x denotes the name of a thread that has invoked a given synchronized method on the object i, *in* denotes acquiring a lock of the object i and the entrance to the method, and *out* denotes releasing the lock and exit from the method.

Wait-notify mechanism. In addition to synchronized blocks and methods, the Java standard library provides the methods `wait()`, `notify()` and `notifyAll()` defined in class Object. Rather than continuously test if the state of an object has changed (i.e. if the lock of an object is released), a thread can suspend itself by calling the `wait()` method, until another thread awakens it by calling the `notify()` or `notifyAll()` method. This wait-notify technique is a communication mechanism between threads.

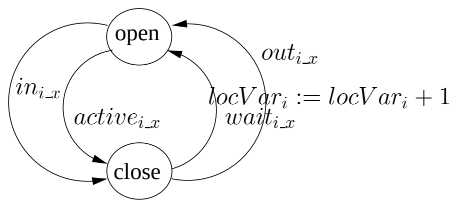

Figure 1: An automata model of binary semaphore for the i-th shared object. In the semaphore we have as many transitions labelled with the action *in* (*out*) as there are threads that call synchronized methods on object i. Transitions with labels *active* and *wait* are added to the semaphore, if a synchronised method called on object i contains the method `wait()`. When action *wait* is performed, a value of the $locVar_i$ is increased by one. Index x denotes the name of a calling thread.

Let us assume that m is a number of threads waiting on the lock of object i. The method `notify()` called by thread j, which wakes up an arbitrary chosen thread from all the threads waiting on the lock of object i, is modelled by an automaton that consists of two locations, one transition with a local action $none_{i\_j}$ and $m-1$ transitions labelled by synchronized actions $notify_{(i,j,k)}$, where $k \in \{1, \ldots, j-1, j+1, \ldots, m\}$. If there are no waiting threads, the `notify()` method has no effect and this is modelled by the local action $none_{i\_j}$ that is enabled only if $locVar_i = 0$. The action $notify_{(i,j,k)}$ can be performed only if $locVar_i > 0$ and it represents the fact that thread j notifies thread k that waits on object i, which thereby is moved to the *ready state*. With this action an instruction $locVar_i := locVar_i - 1$ is also associated. It means that after performing the action, the number of the waiting threads is decreased by one. Note that if there is at least one thread blocked due to a call of a `wait()` method on object i, then the value of the $locVar_i$ counter is greater than zero. Thus, since the transition with action $none_{i\_j}$ is labelled with the guard $locVar_i = 0$, all the transitions labelled with action $notify_{(i,j,k)}$ can be performed with the guard *true*.

The method `notifyAll()` called by thread j, which wakes up all the threads waiting on the lock of object i, is modelled by an automaton that consists of two locations and one transition labelled by the broadcast synchronized action $notify_{(i,j)}$. The action is performed only if $locVar_i \geq 0$. With this action an instruction $locVar_i := 0$ is performed, which means that after execution the action, there is no waiting threads on the lock of object i.

The method `wait()` called by thread k, which suspends the thread, is modelled by an automaton that consists of four locations (called *in*, *wait*, *ready*, *out*) and $2(m-1)+2$ transitions (m is the number of threads waiting on the lock of object i). The first transition (*in* to *wait*) is labelled by a local action $wait_{i\_k}$ denoting the fact that thread k goes to a *waiting state* and opens semaphore i. In this state the thread is waiting to be notified by any other thread, what is represented either by $m-1$ transitions (*wait* to *ready*) labelled with actions $notify_{(i,j,k)}$ or by $m-1$ transitions (*wait* to *ready*) labelled with actions $notify_{(i,j)}$, for $j \in \{1, \dots k-1, k+1, \dots, m\}$. These $notify$ actions are synchronized actions between threads and use three different models of synchronisation. Actions $notify_{(i,j,k)}$ are performed according to the rules of binary synchronisation (in `Uppaal`) or multi-way model of synchronisation (in `VerICS`). Actions $notify_{(i,j)}$ are performed according to the rules of the broadcast synchronisation. Once one of these 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 moved to the running state. This is done by invoking a synchronized action $active_{i\_k}$, which closes semaphore i (transition *ready* to *out*).

The join() method. A thread invokes the `join()` method on another thread in order to wait for the other thread to complete its execution. In the FADD formalism the method `join()` called by thread j on a thread i is modelled as follows. First, to the last location of the automaton for thread i, say l_f, a loop transition (l_f, l_f) is added. It is labelled with a synchronized action $join_{i\_j}$. Then a two state automaton with one transition labelled with the action $join_{i\_j}$ is constructed. It is a part of the automaton for thread j.

3.1. Valuation for FADDs

In order to introduce a valuation into the network of FADDs generated by `J2FADD` and formally describe properties of a given Java program, interesting places of the considered Java program have to be marked by a special `comment tags`. The tags should be contained within comments, to not interfere with other translators, for example with a Java compiler.

Syntax of comment tags. A comment tag can be put into any type of comment. The tag is proceeded with @, then by an optional pair of parentheses containing a zero or more of the tag's modifiers, and finally the parentheses are followed by a name of the tag; currently only names `observable`, `generateHead` and `generateTail` are recognised. Whitespace is allowed only within the parentheses. The name is terminated by a whitespace or the end of the comment.

Comment tags can vary in scope. A tag put before a class contains all methods of the class within its scope. A tag in a comment before a method or a block contains the method or the block, respectively, within its scope. There are also statement-wide tags. Modifiers are a series of comma-separated strings. There is an exception though - compiler modifiers that are prefixed with a dot. These modifiers control how a tag is applied within its scope.

By default, a tag is applied to each operation within the tag's scope. But, if there is a modifier of the form `.annotation @Annotation`, the tag is applied only to the operations that contain the given annotation. Another compiler modifier begins with `.inline`. It controls how the tag is recursively applied to inlined methods. The modifier `.inline infinite` means that the recursion within which the tag should be applied is infinite. A modifier of the form `.inline` $< n >$ defines a maximum recursion depth n, $(n > 0)$. For example, $n = 0$ means that a respective tag is lost along with the replaced method call that contains the tag. For $n = 1$, the tag is propagated from a replaced method call only to the directly inlined method m, and it does not propagate to the methods inlined within m. The modifier `.inline first` means that the tag should be applied only at the first operation of the inlined code.

Tags are ignored in the part of code that is only interpreted.

Special annotations. Special annotations listed below can be used to apply comment tags more selectively.

- @@IN - It marks the target locations of both the "in" transition appearing in the translation of any synchronised method (block) and the "active" transition appearing in the translation of the method `wait()`;
- @@OUT - It marks the target locations of both the "out" transition appearing in the translation of any synchronised method (block), and the "wait" transition appearing in the translation of the method `wait()`;
- @@NOTIFY_THREAD - It marks the source location of "notify" transitions of the translation of the methods `wait()` and `notify()`;
- @@DO_NOTHING - It marks the source location of the transition that does not notify any thread in the translation of the method `notify()`.
- @@NOTIFY - It marks all the locations of the translation of the methods `notify()`;
- @@WAIT - It marks all the locations of the translation of the methods `wait()`.

- @@HEAD - It marks the first location of the translation of a method. The same can be gained by using a comment tag `@generateHead`.
- @@TAIL - It marks the last location of the translation of a method. The same can be gained by using a comment tag `@generateTail`

Comment tags and valuation. When valuation of observables (i.e. the option -vo in our J2FADD tool) is chosen, only locations that have at least a single accompanying operation that contain the comment tag have the valuation variables assigned. Example:

```
@(.inline infinite, .annotation @@WAIT, .annotation
           @@IN)observable put(packet);
```

Apply the discussed tag to every operation (o) that is within the flattened code of the recursively inlined methods that replace the call put(), but only if (o) contains the annotations @@WAIT and @@IN. In effect, the tag is applied only to operations that accompany the transitions "active" of the translation of the methods `wait()`.

4. Example: Dining Philosophers

In this section we consider the well known example of concurrent programming and we model check it by means of the tools: `Uppaal`, `VerICS`, `JPF` and `Bandera`. Since, in fact, `JPF` and `Bandera` can search for deadlocks only, for the considered example we search for deadlocks, which can be defined formally as follows [15]: *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.*

4.1. Problem Description

The description of the dining philosophers problem (DPP) we provide below is based on that in [8]. 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.

4.2. Possible solutions

We have implemented a possible solution of the DPP problem that could lead to a deadlock (see Listing 2). 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**
J2TADD + BMC4TADD	5	12 217	279.2
JPF	4	2.21	3.7
JPF	5	-	-
J2TADD + Uppaal	60	1.16	41.9
Bandera	60	117.02	3.3

Table 1. Dining Philosophers. Deadlock.

Assume now another solution for DPP (see Listing 3), 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	-	-
JPF	4	16.47	3.7
JPF	5	-	-
Bandera	2	76.31	4.6
Bandera	3	-	-

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.

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

Figure 2. Java source code of the DP problem (3 Philosophers).
The main class.

```
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();
      //@(.inline infinite, .annotation @@NOTIFY_THREAD) observable
      right.acquire(); right.release();
      left.release(); s.release();
    }
  }
}
```

Figure 3. Java source code of the DP problem (3 Philosophers) with lackey. The main class.

5. Conclusions

In the paper we have presented a theoretical base of the `J2FADD` translator which together with the verification core of `Uppaal` and `VerICS` allows to validate the well-known concurrency example written in Java: `dining philosophers`. The translator performs a number of optimisations to decrease the often high memory and time requirements of model checking.

The experiments confirm that our approach provides a valuable aid for Java software verification. Moreover, we have compared our results with JPF and Bandera, and it turned out that our tool works much more efficient.

References

[1] http://www.ajd.czest.pl/∼modelchecking

[2] Symbolic Analysis Laboratory (SAL). http://sal.csl.sri.com/ (2008).

[3] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, C. Weise. New generation of Uppaal. In: *Proc. Int. Workshop on Software Tools for Technology Transfer*, T. Margaria, B. Steffen (Eds.), pp. 43–51, 1998.

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

[5] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, H. Zheng. Bandera: Extracting finite-state models from Java source code. In: *Proc. 22nd Int. Conf. on Software Engineering (ICSE '00)*, ACM Press, New York, pp. 439–448, 2000.

[6] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, A. Zbrzezny. VerICS: A tool for verifying timed automata and Estelle specifications. In: *Proc. 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, H. Garavel, J. Hatcliff (Eds.), Lecture Notes in Computer Sience, vol. 2619, pp. 278–283, Springer, Berlin 2003.

[7] A. Farzan, F. Chen, J. Meseguer, G. Roşu. Formal analysis of Java programs in JavaFAN. In: *Proc. 16th Int. Conf. on Computer-aided Verification (CAV'04)*, R. Alur, D.A. Peled (Eds.), Lecture Notes in Computer Sience, vol. 3114, pp. 501–505, Springer, Berlin 2004.

[8] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice Hall, London 1985.

[9] J. E. Hopcroft, J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Massachusetts 1979.

[10] D. Park, U. Stern, J.U. Skakkebaek, D.L. Dill. Java model checking. In: *Proc. 15th IEEE Int. Conf. on Automated Software Engineering (ASE'2000)*, pp. 253–256, IEEE, 2000.

[11] C. Păsăreanu, W. Visser. Verification of Java programs using symbolic execution and invariant generation. In: *Model Checking Software, Proc. SPIN'04*, S. Graf, L. Mounier (Eds.), Lecture Notes in Computer Sience, vol. 2989, pp. 164–181, Springer, Berlin 2004.

[12] W. Penczek, B. Woźna, A. Zbrzezny. Bounded model checking for the universal fragment of CTL, *Fundamenta Informaticae*, **51**, 135–156, 2002.

[13] A. Rataj, B. Woźna, A. Zbrzezny. A translator of Java programs to TADDs. In: *Proc. Int. Workshop on Concurrency, Specification and Programming (CS&P'08)*, pp. 524–535, 2008.

[14] A. Rataj, B. Woźna, A. Zbrzezny. A translator of Java programs to TADDs. *Fundamenta Informaticae*, **95**, 305-324, 2009.

[15] A.S. Tanenbaum. *Modern Operating Systems*, Prentice Hall, Amsterdam 2001.

[16] A. Zbrzezny, B. Woźna. Towards verification of Java programs in VerICS. *Fundamenta Informaticae*, **85**, 533–548, 2008.