Fundamenta Informaticae XX (2009) 1–20
IOS Press

A translator of Java programs to TADDs

Artur Rataj
IT&ACS, Polish Academy of Sciences
ul. Bałtycka 5, 44-100 Gliwice, Poland
email: {arataj@iitis.gliwice.pl}

Bożena Woźna
IM&CS, Jan Długosz University
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland
email: b.wozna@ajd.czest.pl

Andrzej Zbrzezny
IM&CS, Jan Długosz University
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland
email:a.zbrzezny@ajd.czest.pl

Abstract. The model checking tools `Uppaal` and `VerICS` accept a description of a network of Timed Automata with Discrete Data (TADDs) as input. Thus, to verify a concurrent program written in Java by means of these tools, first a TADD model of the program must be build. Therefore, we have developed the `J2TADD` tool that translates a Java program to a network of TADDs; the paper presents this tool.

The `J2TADD` tool works in two stages. The first one consists in translation of a Java code to an internal assembly language (IAL). Then, the resulting assembly code is translated to a network of TADDs. We exemplify the use of the translator by means of the following well-known concurrency examples written in Java: *race condition problem*, *dining philosophers problem*, *single sleeping barber problem* and *readers and writers problem*.

1. Introduction

Model checking [3] is a widely recognised and prominent automatic verification technique both in hardware [10] and protocol verification [8]. It does not rely on complicated interaction with the user for incremental property proving. If a property does not hold, the model checker automatically generates a

counterexample. In model checking, the system to be verified is modelled as a finite state machine (for example as a network of timed automata), which further can be represented by a transition system, and temporal logics are used for specifying the system properties.

Typical examples of finite systems, for which model checking has successfully been applied, are digital sequential circuits and communication protocols, and typical examples of checked properties are reachability properties. In this paper, we consider the model checking problem for concurrent programs written in Java, one of the dominant high-level programming languages, and we test these programs for the properties mentioned above.

Verifying programs written in programming languages like Java is different from verifying digital sequential circuits 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 whole software. In particular, for what concerns verification of Java programs, we see model checking as a method that can be applied to the crucial parts of a Java software.

In order to investigate the challenges that Java programs pose for model checking, we have developed a `J2TADD` tool that translates a Java code to a network of *Timed Automata with Discrete Data* (TADDs) [9, 14]. `J2TADD` implements the translation that has been shown in [15] and there one can find a detailed description of how the concepts of concurrent programming in Java are translated into TADDs.

The TADD formalism is accurate enough to detect concurrency errors and yet abstract enough to make model checking tractable. Moreover, it is an input language of the model checking tools `VerICS`[5] and `Uppaal` [2], so `J2TADD` compiler together with these two tools can be used to validate concurrent programs written in Java. `J2TADD` is a pure Java application that can be run as a standalone command line tool on any platform supporting Java 1.5 or later.

Our case studies have been done for a Java program that suffers from a *race condition* and for Java programs that implement the following well-known concurrency examples: *dining philosophers problem*, *single sleeping barber problem*, and *readers and writers problem*. As verification engines we have used `Uppaal` and `BMC4TADD` - a new version of the BMC module of the tool `VerICS` that has been adapted to work with the `J2TADD` translator. `BMC4TADD` implements the Bounded Model Checking method, where systems are described as a network of TADDs and properties are the reachability one.

Over the last ten years various approaches and tools have been developed to automatically verify Java programs. However, the majority of work has been devoted to analysis of Java bytecode and building new Java Virtual Machines (JVMs) that interpret Java bytecode. In particular, the following two verification and testing environments for Java have been built: `Java PathFinder` (JPF) [12] and `Bandera` [4]. Both tools can model check Java programs on deadlocks and limited Java assertions only.

Apart from the two above tools one can find the following works related to the model checking of Java programs [11, 6]. The paper [11] describes a Java model checker that is based on the SAL (Symbolic Analysis Laboratory) [1] intermediate language and the `Soot` compiler framework. According to the paper the architecture of the tool is the following. First, a Java program is compiled to Java bytecode. Then, the result 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 [11] Java model checker. Moreover, it seems that the work is not continued since the year 2000.

The paper [6] presents Java Formal Analysis (`JavaFAN`), a tool to simulate and formally analyse

multithreaded Java programs at source code and/or bytecode levels. The tool is based on rewriting logic, implemented in the Maude language, and supporting analyses both the Java language and JVM bytecode. It 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 rest of the paper is organised as follows. The next section shortly describes the idea of our translation of a Java code to a network of TADDs. Then, in Section 3 we discuss an assembly language that is used internally by our `J2TADD` translator. In Section 4 we present two stages of our translation (interpreting and generating transitions) that are required to convert the internal assembly code to TADDs. In Section 5 we show a simple Java program that encodes the dining philosophers problem, the resulting internal assembly code and the final TADD specification resulting from it. Finally, we present a case study that confirms that our approach provides a valuable aid for verification of Java programs.

2. Translation from JAVA to TADD

In this section we shortly describe the idea of our translation of a concurrent multithreaded Java program to a network of Timed Automata with Discrete Data (TADDs) [9, 14]; a detailed description of the translation can be found in [15]. TADDs 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.

Each state of a TADD is an abstraction of a state of a given Java program, and each transition represents an execution of a code transforming this abstract state. The subset of Java that can be translated to a network of TADDs 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, objects, constructors, static and non–static methods and synchronisation of methods and blocks. Also, standard thread creation constructs and the following special methods: `Thread.wait()`, `Thread.notify()`, `Thread.join(Thread)`, `Thread.sleep(int)` and `Random.nextInt(int)` are recognised.

A theoretical method of constructing a network of TADDs that models a Java program is shown in [15]. Here we only recall that in TADDs locations are 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). Each transition represents the execution of a Java instruction (for example an assignment statement) for some thread. There is one TADD for each thread, but the main thread: one TADD for each instance of a started thread, and one TADD for each used semaphore. The main thread is treated differently: it is executed by a built–in interpreter instead of being translated like the other threads. The reason for this is twofold:

- to allow for performing computations whose results are normally in Java only known at run–time, so these results can be used at compile–time when constructing TADDs;
- to interpret allocations that are normally absent in the TADD formalism.

In this paper we assume that direct or indirect recursion is not allowed in the considered fragment of the Java language. The following operations are allowed in the interpreted thread: allocations of new objects, including static initialisation, creation and starting of new threads, assignments and method calls,

arithmetic expressions that can be statically determined.

To implement the translation shown in [15], we first translate a Java code to an internal assembly language (see the next section). Than, the resulting assembly code is translated to a network of TADDs; this is presented in Section 4.

3. Internal Assembly Language (IAL)

This section describes an assembly language that is used internally by our `J2TADD` translator; in Section 3.6 we provide a formal grammar of a single operation of the language.

We start with an explanation of what we mean by *dereferences* and *literals* that we use in our internal assembly language. Then, we describe how the Java static and non–static methods are handled. Next, we discuss polymorphism and annotations. Finally, we define allowed operations and show their connections with object allocation, method calls, synchronisation and standard programming language constructs like assignment, expressions or loops.

3.1. Dereferences and literals

A *dereference* is a pair *(object, field)* of two variables, where the first one must be an object reference, including the null reference. Both *object* and *field* variables are always either local or static variables, and the variable *object* can be *null*. If the variable *object* is not null, then this is a dereference of a respective non–static field *field* of the Java object *object*, and this is the only case when there is performed an actual variable dereference. Textual representation of such a dereference is `object::field`. If the variable *object* is null, then the dereference refers to a local variable or a static field. Textual representation of such a reference is `field`. The type of a dereference in each case is the same as that of *field*.

Literals are constant values and can have one of the following types: *void*, *char*, *byte*, *int*, *short*, *long*, *float*, *double* and a Java type; TADDs currently support only integer types, though.

In our assembly language we refer to dereferences and literals as to values, i.e., we have ⟨value⟩ ::= ⟨literal⟩|⟨dereference⟩.

3.2. Methods

There is no difference between static and non–static Java methods in our assembly language apart from one: a non–static method has an additional argument called `this`, which is of the type of the class containing the method. The argument references an instance of the object "this", that defines the non–static context of the method. A constructor is simply a non–static method, but it is named after the class it constructs.

3.3. Instantiation and polymorphism

A run-time Java object gets its non–static fields initialised in its method `*object`. A class we treat as a special object. The static fields of a class are initialised in the method `*static`.

In our assembly language we define only one polymorphism rule. Namely, if the class B extends class A, then an object of the class B can be used when an object of the class A is required. A run-time object class type, as opposed to a variable compiled type, decides about which non–static method is

called. The other possible polymorphism rules can be enforced during parsing, semantic check or code generation, the assembly language itself does not define or enforce them.

3.4. Annotations

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate. Annotations have a number of uses, among them: information for the compiler, compiler-time and deployment-time processing, or run-time processing (some annotations are available to be examined at run-time).

In our assembly language methods can have annotations like in Java. There are also internal special annotations that begin with '@@' and annotate special methods:

- `@@MARK` – do not call, the method is just a marker; it is replaced by a *no operation*, see the description of *no operation* for details;
- `@@START_THREAD` – start a thread;
- `@@IGNORE` – ignore any call of the method; it works like `@@MARK` but it is included for clarity.

Back-ends and interpreters (see Section 4) can define their own annotations, and use them along with these two described in this section. For example, the interpreter used by the translator uses an annotation `@@START_THREAD` to start a new thread, and the TADD backend uses an annotation `@@RANDOM` to mark a method that generates random numbers. Both annotations are used together with the annotation `@@MARK`. Normally, the user of the translator does not need to know or use these special annotations. Instead, there is a special library of classes like `Object` or `Thread`, in which certain methods are marked with these annotations. For example, `Thread.start()` is annotated with `@@START_THREAD` in the library.

3.5. Operations

The following operations are defined in the internal assembly language:

- Allocation: `<dereference> = new <constructor> (<argument>*)`. It instantiates an object, calls the object's method `*object` and then the respective constructor. Example:

  ```
  #c3 = new ->Sender(mch.abp.v1.Sender, mch.abp.v1.LossyChannel) channel
  ```

  An object `Sender` is created using a constructor that takes an argument of the type `mch.abp.v1.LossyChannel`. The argument is the copy of a local variable `channel` of the caller method. The other argument is the new object and provides the "this" local variable for the non–static context. It is, thus, of the type of the created object.
- An assignment: `<dereference> = <value>`. Example: `this::ackBit = ackBit`. This operation copies the value of the local variable `ackBit` to the field of the object of a non–static method.
- A binary expression: `<dereference> = <value> <operator> <value>`. It allows for standard Java binary operators and also for a concatenation of a string to any other value. Here the standard Java rules apply to determine the type of the reference variable; for example, an integer and a float produce a float. Only the following binary operators are currently supported by the TADDs, though: plus, minus, multiply, divide, modulus. Example: `#c5 = ignoreBit > 2`. Test if a local variable `ignoreBit` is greater than a constant `2`, and store the result in a local Boolean variable `#c5`.

- A conditional branch: `<value> ? <label> : <label>`. This condition `value` can be a dereference or a constant. The labels point to operations to jump to if the condition is, respectively, true or false. A label can be null what means that there is no respective jump. Example: `#c5?<null>:10`. If `#c5` is true, then go to the next operation. Otherwise, jump to the operation of index 10.
- A call to a method: `<dereference> = <method> (<argument>*)`. If the method returns void, the dereference is null. As in Java, the non–static methods are virtual. Argument "this" of non–static methods is always the first one. Example:

```
ignoreBit = ->nextInt(java.util.Random, int)
  @@MARKER @@RANDOM (this)this::random 5.
```

  Call to a non–static method `nextInt(int)` of the object referred by a field `random` belonging to `int` turn to the object of the calling non–static method. The additional argument of the type `java.util.Random` passes a reference to "this" of the called method. A value of `5` is copied to the other parameter. The called method has two internal annotations: `@@MARKER` and `@@RANDOM`.
- An unconditional jump: `goto <label>`. Example: `goto 4`. Go to the operation at index 4.
- No operation: `no op [<dereference> = ] <annotation>+ (<argument>*)`. It can be used as a holder of a label or of an annotation. Also, despite its name, it can perform something if the annotation specifies some special operation, that can not be expressed by the regular assembly language operations, but that is needed for some particular translation. In that case, the operation may optionally have arguments, and a dereference to hold the result of the operation. Any call to a method annotated with `@@MARKER` is replaced by no operation. Example:

```
no op @@MARKER @@SLEEP (i#0mch.abp.v2.Sender_run__#c8).
```

  Perform the special operation "sleep" on a thread referenced by `i#0mch.abp.v2.Sender_run__#c8`. The operation does not return anything, so the part `[<dereference> = ]` is absent.
- `return` – returns from a method.
- `sync begin|end <dereference>` – either the begin or the end of a synchronised code. The dereference references the semaphore. Example: the code of a synchronised method always starts with `sync <begin> this` and there is `sync <end> this` before each `return` operator in the method.
- An unary expression: `<dereference> = <operator> <value>`. It allows for standard Java unary operators. Only the following unary operators are currently supported by the TADDs, though: arithmetic negation and Boolean negation. Example: `#c7::ackBit = !#c7::ackBit`. Negate the field ackBit of the object referred by the variable `#c7`.

We further assume that:

- A return value of a method is in a local variable named `*retval`.
- A label is simply an index of the target operation, or, in the case of a conditional branch, optional null meaning no jump, but execution of the following operation instead.
- A signature, like in Java, is an unique key that defines a method. A signature contains the method's name and types of arguments. It does not contain the type of "this" argument of non–static methods.

3.6. Grammar of an operation

This section defines a formal grammar of our assembly language operation ⟨operation⟩, provided in the BNF notation.

- ⟨allocation⟩ ::= ⟨dereference⟩ = new ⟨method call⟩ (⟨argument⟩*)
- ⟨annotation⟩ ::= ⟨identifier⟩
- ⟨assignment⟩ ::= ⟨dereference⟩ = ⟨value⟩
- ⟨binary_expression⟩ ::= ⟨dereference⟩ = ⟨value⟩ ⟨binary_operator⟩ ⟨value⟩
- ⟨binary_operator⟩ ::= PLUS| MINUS | MULTIPLY| DIVIDE | MODULUS| INCLUSIVE_OR| EXCLUSIVE_OR| AND| EQUAL| CONDITIONAL_OR| CONDITIONAL_AND| INEQUAL| LESS| GREATER| LESS_OR_EQUAL| GREATER_OR_EQUAL| SHIFT_LEFT| SHIFT_RIGHT| SIGNED_SHIFT_RIGHT|
- ⟨conditional_branch⟩ ::= ⟨value⟩ ? ⟨label⟩ : ⟨label⟩
- ⟨call⟩ ::= ⟨dereference⟩ = ⟨method call⟩ (⟨argument⟩*)
- ⟨expression⟩ ::= ⟨unary_expression⟩|⟨binary_expression⟩
- ⟨identifier⟩ is a string of characters.
- ⟨jump⟩ ::= ⟨unconditional_jump⟩| ⟨conditional_branch⟩
- ⟨label⟩ is an index of the target operation. Indices start at 0.
- ⟨literal⟩ is a constant.
- ⟨method_call⟩ ::= → ⟨method_signature⟩ ((⟨dereference⟩)*) (⟨annotation⟩*)
- ⟨method_signature⟩ is the signature of a method, see Sec.3.5 for details.
- ⟨no_op⟩ ::= no op [⟨dereference⟩ =] (⟨annotation⟩*)
- ⟨operation⟩ ::= ⟨allocation⟩| ⟨assignment⟩| ⟨expression⟩| ⟨jump⟩| ⟨method_call⟩| ⟨return⟩|⟨sync⟩|
- ⟨dereference⟩ ::= [⟨variable_name⟩::]⟨variable_name⟩
- ⟨return⟩ ::= return
- ⟨sync⟩ ::= sync begin|end ⟨dereference⟩
- ⟨unary_expression⟩ ::= ⟨dereference⟩ = ⟨unary_operator⟩ ⟨value⟩
- ⟨unary_operator⟩ ::= NEGATION | CONDITIONAL_NEGATION | BITWISE_COMPLEMENT
- ⟨unconditional_jump⟩ ::= goto ⟨label⟩
- ⟨value⟩ ::= ⟨literal⟩|⟨dereference⟩
- ⟨variable_name⟩ ::= ⟨identifier⟩

4. Translation from IAL to TADDs

Once the assembly code is generated, two stages remain to convert the code to TADDs: interpreting the main thread and then generating transitions from the other threads.

4.1. Interpreter

The role of the interpreter is to initialise variables, load needed classes, create objects and threads.
Threads. As in Java, the start method of the main thread is the main method of the translated application. Only a single main method within the application is allowed by the translator.

The interpreter is started with this single thread and ends once the main thread ends. All other threads, called here TADD threads, created by the interpreter when executing the main thread are translated to TADDs, one TADD per thread.

Library and annotations. The interpreter uses a TADD–specific library that defines the needed Java classes like `Object` or `Thread`. As already discussed, some methods in the library are marked with special annotations beginning with double '@'. Let us describe these annotations in detail.

The annotations `@@IGNORE`, `@@MARKER` and `@@START_THREAD` are used as defined by the assembly language, and were already described in Sec.3.4. Additional TADD–specific annotations defined in the library are as follows:

- `@@NOTIFY` – perform `notify()` on an object, used with marker methods, usually only with `Thread.notify()`.
- `@@RANDOM` – generate a random number in the range 0 ... method's integer argument - 1, used with marker methods, usually only with `Random.nextInt(int)`.
- `@@SLEEP` – sleep by a given number of milliseconds, used with marker methods, usually only with `Thread.sleep(int)`.
- `@@WAIT` – perform `wait()` on an object, used with marker methods, usually only with `Thread.wait()`.

4.2. Building TADDs

This section describes the process of generating TADDs after interpreting the main thread.

Virtual call resolution. The first step is a virtual call resolution. In Java, a called non–static method is determined not by compile–time type of an object, but by the run-time type of an object. After the interpreter is already run and run-time types are known, some calls can be resolved. If it is not possible because of a null variable or a variable not initialised at the interpreter stage, an error is reported.

Inlining calls. Once the calls are resolved, it is possible to 'flatten' the code by the translator. To do so, the code is recursively inlined with a thread, beginning at each thread's method `run()`, so that, in effect, there should be no calls left in the 'flattened' code. The operation is performed because no stack is available in the generated TADDs for calling methods, and also because inlined code allows for more efficient optimisations like reduction of variables and assignments.

Optimisation. The next step is an optimisation of the code like: value propagation, optimising branches with static conditions, compacting jump and assignment chains, checking for dead code, removal of assignments and locals that are unused. This step is performed even if the code had already been optimised before the discussed inlining of calls. It is because the inlining may allow for substantial further optimisations, as mentioned above.

Ambiguity check. The `J2TADD` translator allows for assignments of reference variables within the TADD threads, if the translator can determine that these assignments do not cause a reference to be ambiguous at compile time. The translator checks only the local reference variables for ambiguity, and thus, assignments to non–local reference variables within TADD threads are forbidden. Ambiguity means here that the dereferenced variable can have more than a single value at the time of being dereferenced, or at least that it was not possible to determine statically, by tracing execution paths within a method, that there is only one such value possible. Such an ambiguity of a reference variable would make it impossible to statically dereference the variable at compile time. This in turn would require the actual dereferences to be performed at run–time. TADDs, though, do not support dereferences and a run-time emulation of dereferences is not implemented.

Generating transitions. Most assembly language operations are straightforwardly translated to TADDs: assignments, jumps, binary and unary expressions and conditional branches. However, there are some

sequences of assemble language operations that are treated as a whole and as such transformed to respective TADD transitions:

- a conditional expression followed by a conditional branch, which uses the result of the conditional expression, is transformed to two opposite conditional expressions on two transitions. Example:

```
0 c = a < 1
1 c ? <null> : 3
```

  is translated to two conditional expressions: $a < 1$ on a transition from state 1 to state 2 and $a \geq 1$ on a transition from state 1 to state 3.
- a marker method `@RANDOM` followed by a marker method `@SLEEP` that uses the result of the first marker method is transformed to a respective clock condition. Example:

```
0 c = @@MARKER @@RANDOM (random 150)
1 no op @@MARKER @@SLEEP (c)
```

  is translated to a clock condition $(x \geq 0) \wedge (x < 150)$.

Annotations `@NOTIFY`, `@WAIT` and synchronisation operations cause a new semaphore TADD to be created for the respective synchronisation object if the semaphore does not exist yet. Transitions are added to the semaphore as necessary. As required by Java, within a single thread, synchronisation operations on run-time object k are ignored if they are nested within synchronisation operations that are also on the object k.

5. Example translation

Let us discuss an example translation of a Java program, which implements a possible solution of the well known dining philosophers problem (for the discussion of the problem see 6.2), to our assembly language (IAL), and then to TADDs. The translated class (program) `College3` extends `Thread` and objects of the class represents threads in the output network of automata. The source of the class is given in Fig. 1. IAL representation of one of the actual threads is shown in Fig. 2. The IAL code is eventually translated into one of the automata. The transitions of the automaton, written in the `Verics` format, are shown in Fig. 3; the notation `transition x y z` means that there is a transtion from location `x` to location `y` labelled by `z`. Because of the length of the representation, comments and lines containing only the keywords `reset` and `end`, were removed from the file.

Lines in each of the three representations are prefixed with `s` for the source file, `i` for the IAL file and `t` for the output file, as seen in the figures, and so the lines will further be referred to in this section. Counterparts of each line in the different representations are shown in Table 1. If a method is called twice, its code is translated twice. In the table, a group of lines related to such a single call is separated from possible other groups with a semicolon.

Let us now discuss the example translation in detail.

The thread begins with calling `run()`, that begins in `s61`. The method is non–static, and thus, its local variable `this` has to be set to point to the method's object. As can be seen from the mapping, there is a related assignment operation in `i0`. The method itself begins in `s62` with a condition that can be evaluated at compile time. Such operations, and several others within the thread, do not have any

```
 s1 public class College3 {
 s2
 s3   public static void main(String args []) {
 s4     Fork fork0 = new Fork(false);
 s5     Fork fork1 = new Fork(false);
 s6     Fork fork2 = new Fork(false);
 s7
 s8     Philosopher p0 = new Philosopher(0,
          fork0, fork1);
 s9     Philosopher p1 = new Philosopher(1,
          fork1, fork2);
s10     Philosopher p2 = new Philosopher(2,
          fork2, fork0);
s11
s12     (new Thread(p0)).start();
s13     (new Thread(p1)).start();
s14     (new Thread(p2)).start();
s15   }
s16 }
s17
s18 class Fork {
s19
s20   private boolean unavailable;
s21
s22   public Fork(boolean unavailable) {
s23     this.unavailable = unavailable;
s24   }
s25
s26   public synchronized void acquire() {
s27     while (unavailable) {
s28       try {
s29         wait();
s30       } catch (InterruptedException e) {
s31       }
s32     }
s33     unavailable = true;
s34   }
s35
s36   public synchronized void release() {
s37     unavailable = false;
s38     notify();
s39   }
s40
s41 }
s42
s43 class Philosopher implements Runnable {
s44   private int nr;
s45   private Fork left, right;
s46
s47   public Philosopher(int nr, Fork left,
          Fork right) {
s48     this.nr = nr;
s49     this.left = left;
s50     this.right = right;
s51   }
s52
s53   // @generateHead
s54   private void eating() {
s55       try {
s56         Thread.sleep(1);
s57       } catch (InterruptedException e) {
s58       }
s59   }
s60
s61   public void run() {
s62     while (true) {
s63       try {
s64         Thread.sleep(5);
s65       } catch (InterruptedException e) {
s66       }
s67       left.acquire();
s68       // @(.inline infinite, .annotation
            @@NOTIFY_THREAD)observable
s69       right.acquire();
s70       eating();
s71       right.release();
s72       left.release();
s73     }
s74   }
s75 }
```

Figure 1. Java source code of the program `College3`.

counterparts in both IAL and the TADDs formalism, as there is no need to execute anything related to operations like that at run time.

The first operation that is actually translated to a network of TADDs is in `s64`. As it is a call to a marker method, its counterpart in IAL is an annotated no–operation. The annotations `@@MARKER @@SLEEP` and the argument `5` describes the original operation. The no–operation is in turn translated to transitions `t1`, `t2`, `t10` and `t11` that contain a reset of the clock `x1` and a condition `x1 >= 5`. The `catch` block at `s65` and `s66` is empty. While the translator does not support exceptions, it accepts exception–related constructs that do not effectively contain any exception–related code, assuming, though, that no exceptions will actually be thrown.

The next operation is the call to a non–static synchronized method `acquire()` in `s69`. In IAL, translation of the call begins with the assignment of the local variable `this`, like it has already been

```
 i0 i#0<default>.Philosopher_run__this = this::runnable
 i1 no op @@MARKER @@SLEEP (5)
 i2 i#0<default>.Fork_acquire__this = i#0<default>.Philosopher_run__this::left
 i3 sync begin i#0<default>.Fork_acquire__this
 i4 i#0<default>.Fork_acquire__this::unavailable ? <null> : 7
 i5 no op @@MARKER @@WAIT (i#0<default>.Fork_acquire__this)
 i6 goto 4
 i7 i#0<default>.Fork_acquire__this::unavailable = true
 i8 sync end i#0<default>.Fork_acquire__this
 i9 <@(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    i#0<default>.Fork_acquire__this = i#0<default>.Philosopher_run__this::right
i10 <@(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    sync begin i#0<default>.Fork_acquire__this
i11 <@(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    i#0<default>.Fork_acquire__this::unavailable ? <null> : 14
i12 <@(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    no op @@MARKER @@WAIT (i#0<default>.Fork_acquire__this)
i13 <@(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    goto 11
i14 <@(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    i#0<default>.Fork_acquire__this::unavailable = true
i15 <@(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    sync end i#0<default>.Fork_acquire__this
i16 <@()generateHead @(.inline infinite, .annotation @@NOTIFY_THREAD)observable>
    no op @@HEAD ()
i17 <@()generateHead> no op @@MARKER @@SLEEP (1)
i18 i#0<default>.Fork_release__this = i#0<default>.Philosopher_run__this::right
i19 sync begin i#0<default>.Fork_release__this
i20 i#0<default>.Fork_release__this::unavailable = false
i21 no op @@MARKER @@NOTIFY (i#0<default>.Fork_release__this)
i22 sync end i#0<default>.Fork_release__this
i23 i#0<default>.Fork_release__this = i#0<default>.Philosopher_run__this::left
i24 sync begin i#0<default>.Fork_release__this
i25 i#0<default>.Fork_release__this::unavailable = false
i26 no op @@MARKER @@NOTIFY (i#0<default>.Fork_release__this)
i27 sync end i#0<default>.Fork_release__this
i28 goto 1
```

Figure 2. IAL code of one of the threads of objects of the class `College3`.

discussed. Then, as the call is not a marker method like `Thread.sleep`, but a method with an actual code, inlining of the code begins. A code in a synchronized method, if the call operation is not already synchronized with a given lock, needs to be preceded with a synchronization begin operation, in this case in `i3`, and a matching synchronization end operation, in the case in `i8`. As shown in the mapping, there are equivalent automata transitions in `t13` and `t21`. There are also lines `i10`, `i15`, `t22` and `t31` shown in the array, yet after semicolons. They contain the synchronization code as well, but related to another call of the method `acquire()`.

The condition in `s27` is translated to `i4` and then to two transitions with conditions in `t14`, `t15`, `t16` and `t17`. The two conditions are opposite and, as it can be seen in the automata code, split the thread into two branches.

Within the inlined method, there is a call to a marker method in `s29`, translated, like the previous call to a marked method, into an annotated no–operation, but the annotations are different this time – `@@MARKER @@WAIT`, and there is no argument. The operation is translated to a set of automata transitions,

```
 t1 transition 0 6 24
 t2 reset x1
 t3 transition 1 19 25
 t4 x1 >= 1
 t5 reset x1
 t6 transition 2 13 8
 t7 transition 3 2 26
 t8 transition 4 8 0
 t9 transition 5 4 27
t10 transition 6 7 28
t11 x1 >= 5
t12 reset x1
t13 transition 7 8 2
t14 transition 8 9 29
t15 cond z0 = c1
t16 transition 8 10 30
t17 cond z0 = c0
t18 transition 9 5 6
t19 transition 10 11 31
t20 assign c1 + c0 to z0
t21 transition 11 12 4
t22 transition 12 13 10
t23 transition 13 14 32
t24 cond z1 = c1
t25 transition 13 15 33
t26 cond z1 = c0
t27 transition 14 3 14
t28 transition 15 16 34
t29 assign c1 + c0 to z1
t30 transition 17 18 35
t31 transition 16 17 12
t32 transition 19 20 10
t33 transition 18 1 36
t34 reset x1
t35 transition 21 22 37
t36 assign z4 - c1 to z4
t37 transition 21 22 38
t38 cond z4 = c0
t39 transition 20 21 39
t40 assign c0 + c0 to z1
t41 transition 23 24 2
t42 transition 22 23 12
t43 transition 25 26 40
t44 assign z3 - c1 to z3
t45 transition 25 26 41
t46 cond z3 = c0
t47 transition 24 25 42
t48 assign c0 + c0 to z0
t49 transition 26 0 4
```

Figure 3. Shortened TADDs representation of one of the threads of the objects `College3`.

as described in the mapping. The assignment operation in `s37` has equivalent assignments in `i20`, `t39` and `t40`.

The next call in `run()`, in the line `s69`, differs from the previous one in a different value assigned to the called method's `this`, and in being tagged with `@observable`. The two differences are seen in the lines `i9` – `i15`, when compared to the already discussed `i2` – `i8`. The IAL operations are tagged this time, and in the TADDs formalism, a different variable is assigned – compare `t39`, `t40` with `t47`, `t48`. This is because the field `unavailable` belongs to two different objects in each of the two calls in, respectively, `s67` and `s69`.

Even that all operations within `i9` – `i15` are tagged, the TADD backend, while generating transitions from these operations, will check for the filter `.annotation NOTIFY_THREAD` and will mark only these transitions as observable that are annotated with `@@NOTIFY_THREAD`.

The next operation in the method `run()` is a call to a not synchronized method `eating()`. The method is tagged with `generateHead`, what is reflected by an additional transition in `t30`.

The next two calls are to `release()`. It is a non–static synchronized method, as `acquire()`. The method contains a call to a marker method `notify()`. A counterpart IAL code is seen for example in `i21`. The code of this particular IAL operation is then translated to transitions shown in lines `t35`, `t36`, `t37`, `t38`.

The last operation in `run()` is a jump in `s73` to the condition of the `while` loop. The jump is translated to a `goto` operation in `i28`, which in turn is translated to a transition in `t49`. As seen in the mapping, to this transition has also been translated the synchronization end operation at `i27`.

6. Case study

The main aim of this paper was to show a new model checking method for verification of concurrent programs written in Java. This method consists in translating a Java program to a network of TADDs, and then applying the existing model checking tools accepting a description of a network of TADDs as input. As we have already mentioned at the very beginning of the paper, the model checking tools like

Java	IL	Verics
s26	i3; i10	t13; t22
s27	i4; i11	t14, t15, t16, t17; t23, t24, t25, t26
s29	i5; i12	t18, t9; t27, t7
s32	i6; i13	t8; t6
s33	i7; i14	t19, t20; t28, t29
s34	i8; i15	t21; t31
s36	i19; i24	t32; t41
s37	i20; i25	t39, t40; t47, t48
s38	i21; i26	t35, t36, t37, t38; t43, t44, t45, t46
s39	i22; i27	t42; t49
s53	i16	t30
s56	i17	t33, t34, t3, t4, t5
s61	i0	-
s64	i1	t1, t2, t10, t11, t12
s67	i2	-
s68, s69	i9	-
s71	i18	-
s72	i23	-
s73	i28	t49

Table 1. Mutual dependencies between the lines in each representation.

`Uppaal` or `VerICS` accept networks of TADDs as input. Thus, to verify concurrent programs written in Java using those tools, we had to develop the `J2TADD` tool that translates a Java program to a network of TADDs.

In this section, we report on results that we have got when evaluating the effectiveness of the proposed verification method by means of four well known concurrency examples written in Java. We do this by comparing our results with those returned by the tools `JPF` and `Bandera`.

For all the considered examples we have searched for deadlock states, and additionally, we have tested a *race condition* by using `JPF`, `Uppaal`, and the `BMC4TADD` module of `VerICS`. A state is a deadlock state if there are no outgoing action transitions neither from the state itself or any of its delay successors.

In order to search for a deadlock state `JPF` tests for every non-end state if there is any runnable thread left. In `Uppaal`, in order to search for a deadlock state, a special state formula $E\Diamond deadlock$ that is satisfied for all deadlock states is used. It is worth to mention that using `Uppaal` one can also handle the following temporal properties: possibility, invariants, potentially always, eventually, if a request occurs then it will be eventually acknowledge.

In `BMC4TADD` to search for a deadlock state we test reachability of a state satisfying certain (usually

undesired) property. For this the transition relation of a given network of TADDs is unfolded up to some depth k, and encoded as a propositional formula. Then, the reachability property to be tested is encoded as a propositional formula as well, and satisfiability of the conjunction of these two formulae is checked using a SAT–solver. If the conjunction is satisfiable, one can conclude that a counterexample (a path to an undesirable state) was found. Otherwise, the value of k is incremented. The above process can be terminated when the value of k is equal to the diameter of the system, i.e., to the maximal length of a shortest path between its two arbitrary states.

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. Moreover, we have set the time–out limit to 15min for RSAT–solver to get the answer.

6.1. Race condition

Problem description. In practical multithreaded applications, it is common that two or more threads need to share access to the same objects. However, if two threads have access to the same object and each calls a method that modifies the state of the object at the same time, then the result can be partly what one thread wrote and partly what the other thread wrote. Depending on the order in which the object was accessed, a corrupted object can result. Such a situation is called a *race condition*.

An example of a race condition. In our example program *RaceCondition4.java* (see Listing 1) there are two threads that run concurrently and access to a shared variable that is initially set to 0. Each thread gets a value of the shared variable, increase this value by one and write back the updated value to the variable. The above operation is repeated n times by both threads. Therefore, one could expect that the final value of the shared variable will be equal to $2n$. However, one can observe that there exit executions of the program which end with the value of the shared variable that is less than $2n$. This is because these threads do not lock the shared variable while it is being accessed. In a proper realisation these threads should lock the shared variable while it is being accessed and then should unlock it when they are finished.

We are able to detect the above race condition by translation of the Java code of our example program to a network of TADDs and then checking reachability of a state in which the final value of the shared variable is less than $2n$; the reachability checking was done by the tools `Uppaal` and `BMC4TADD`. W have also tested the race condition property by means of the the `JPF` tool; the `Bandera` is not able to check this property. The results for this property are in Table 2.

Tools	n	sec.	MB
J2TADD + BMC4TADD	4	2797.2	103.6
J2TADD + BMC4TADD	5	-	-
JPF	$<=$ 16000	$<=$ 57.79	$<=$ 471
J2TADD + Uppaal	$<=$ 16000	$<=$2.9	$<=$80.4

Table 2: Race condition

6.2. Dining Philosophers Problem

Protocol Description. The description of the dining philosophers problem (DPP) we provide below is based on that in [7]. 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

```
1  public class RaceCondition4 {
2    public static void main (String[] args) {
3      Variable variable = new Variable();
4      Thread b1 = new Thread (new Beaver (variable, 4));
5      Thread b2 = new Thread (new Beaver (variable, 4));
6      b1.start();  b2.start();
7      (new Thread (new Observer (b1, b2, variable, 8))).start();
8    }
9  }
10
11 class Variable {
12   private int value;
13   Variable() { this.value = 0;}
14   public synchronized int getValue() { return value; }
15   public synchronized void setValue (int value) { this.value = value; }
16 }
17
18 class Observer implements Runnable {
19   private Thread b1, b2;
20   private Variable variable;
21   private int expResult;
22   private boolean error;
23   public Observer (Thread b1, Thread b2, Variable variable, int expResult) {
24     this.b1 = b1;   this.b2 = b2;
25     this.variable = variable;
26     this.expResult = expResult;
27     this.error = false;
28   }
29   public void run() {
30     try {  b1.join(); } catch (Exception e) {}
31     try {  b2.join(); } catch (Exception e) {}
32     if (variable.getValue() != expResult) {
33       /* @observable */
34       error = true;
35     }
36   }
37 }
38
39 class Beaver implements Runnable {
40   private Variable variable;
41   private int limit;
42   public Beaver (Variable variable, int limit) {
43     this.variable = variable;
44     this.limit = limit;
45   }
46   public void run() {
47     for (int j = 0; j < limit; ++j) {
48       int tmp = variable.getValue() + 1;
49       variable.setValue (tmp);
50     }
51   }
52 }
```

Listing 1. Java source code of the race condition problem

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.

We have implemented a possible solution of the DPP problem that could lead to a deadlock; one can get the implementation by removing from Listing 2 the class `Lackey` and all the occurrences of instructions containing the variable `s`. 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 3.

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 4 for the results).

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 3: Dining Philosophers. Deadlock.

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 4: Dining Philosophers. Absence of deadlocks.

6.3. Single Sleeping Barber Problem

Consider a hypothetical barber shop that has one barber, one barber chair, and a waiting room with several chairs for customers. When a barber finishes cutting a customer's hair, he fetches another customer from the waiting room if there is a customer, or the barber sits in his chair and sleeps if there are no customers. A customer who needs a haircut enters the waiting room. If the waiting room is full, the newly arrived

```
1  public class College5L {
2    public static void main(String args []) {
3      Fork f0 = new Fork(false); Fork f1 = new Fork(false);
4      Fork f2 = new Fork(false); Fork f3 = new Fork(false);
5      Fork f4 = new Fork(false);
6      Lackey s = new Lackey(4);
7      Phil p0 = new Phil(0,f0,f1,s); Phil p1 = new Phil(1,f1,f2,s);
8      Phil p2 = new Phil(2,f2,f3,s); Phil p3 = new Phil(3,f3,f4,s);
9      Phil p4 = new Phil(4,f4,f0,s);
10     (new Thread(p0)).start(); (new Thread(p1)).start();
11     (new Thread(p2)).start(); (new Thread(p3)).start();
12     (new Thread(p4)).start();
13   }
14 }
15 class Fork {
16   private boolean unavailable;
17   public Fork(boolean unavailable) {  this.unavailable = unavailable; }
18   public synchronized void acquire() {
19     while (unavailable) { try {wait(); } catch (Exception e){} }
20     unavailable = true;
21   }
22   public synchronized void release() {  unavailable = false; notify(); }
23 }
24 class Lackey {
25   private int m; private int max;
26   public Lackey(int max) {this.max = max;}
27   public synchronized void acquire() {
28     while (m >= max) { try {wait();} catch (Exception e) {} }
29     ++m;
30   }
31   public synchronized void release() { --m; notify(); }
32 }
33 class Phil implements Runnable {
34   private int nr; private Lackey s;
35   private Fork left, right;
36   public Phil(int nr, Fork left, Fork right, Lackey s) {
37     this.nr = nr; this.s = s;
38     this.left = left; this.right = right;
39   }
40   public void run() {
41     while (true) {
42       s.acquire(); left.acquire();
43       // @(.inline infinite, .annotation @@NOTIFY_THREAD)observable
44       right.acquire(); right.release();
45       left.release(); s.release();
46     }
47   }
48 }
```

Listing 2. Java source code of the dining philosophers problem

customer simply leaves. If the barber is busy but in the waiting room there is a vacant chair available, the customer takes a seat. If the waiting room is empty and the barber is sleeping, the customer sits in the barber chair and wakes the barber up.

Implementing a wrong solution of the problem can lead either to deadlock. The deadlock can happen, if the barber waits on a customer and a customer waits on the barber. We have taken a proper implementation of the problem from the Bandera website; it is based on a solution discussed in book [13] and we have checked the solution for absence of deadlocks and the results are in Table 5.

Tools	No. Customers	sec.	MB
J2TADD+Uppaal	4	107	42.7
JPF	3	18.13	3.7
JPF	4	-	-
Bandera	2	612	3.3
Bandera	3	-	-

Table 5: Single Sleeping Barber. Absence of deadlocks.

6.4. Readers and writers

The readers and writers problem has two types of threads accessing the shared data. The first type, called readers, only wants to read the shared data. The second type, called writers, may want to modify the shared data. If a writer is accessing the shared data, then no other writer or reader can do this.

We have implemented a possible solution of the above problem for n writers and n readers, $n \leq 1$ (see Listing 3). These programs consist of at least 2 threads: one reader and one writer. To increase the size of the problem, additional readers and writers can be added. This solution does not admit deadlocks; the results are in Table 6.

Tools	No. of Readers and Writers	sec.	MB
J2TADD+Uppaal	2	0.02	1.5
J2TADD+Uppaal	5	36.9	138.4
J2TADD+Uppaal	6	-	-
JPF	2	3.59	4.9
JPF	3	-	-
Bandera	2	150.3	115.54
Bandera	3	-	-

Table 6: Readers and writers. Absence of deadlocks.

7. Summary

The `J2TADD` translator is not a self–contained verification system, but instead it provides an output for a verification system like `Uppaal` or `VerICS`, and it can be used as a part of a modular chain of such tools. Thus, users can reuse their knowledge of `VerICS`, `Uppaal` or other tools that as a input take a network of TADDs to use the translator to validate various concurrent programs written in Java. The translator

```
1  public class R2W2 {
2    public static void main(String args []) {
3      ReadingRoom readingRoom = new ReadingRoom();
4      Reader r0 = new Reader(0, readingRoom);
5      Reader r1 = new Reader(1, readingRoom);
6      Writer w0 = new Writer(0, readingRoom);
7      Writer w1 = new Writer(1, readingRoom);
8      r0.start();  r1.start();  w0.start();  w1.start();
9    }
10 }
11 class ReadingRoom {
12   private int numberOfReaders = 0;
13   public synchronized void startReading() {
14     while (numberOfReaders < 0) { try { wait(); } catch (Exception e) {} }
15     ++numberOfReaders;
16   }
17   public synchronized void stopReading() {
18     --numberOfReaders;
19    if (numberOfReaders == 0) { notify(); }
20   }
21   public synchronized void startWriting() {
22     while (numberOfReaders != 0) { try { wait(); } catch (Exception e) {} }
23     numberOfReaders = -1;
24   }
25   public synchronized void stopWriting() {
26     numberOfReaders = 0; notify();
27   }
28 }
29 class Reader extends Thread {
30   private int nr;
31   private ReadingRoom readingRoom;
32   public Reader(int nr, ReadingRoom readingRoom) {
33     this.nr = nr;  this.readingRoom = readingRoom;
34   }
35   public void run() {
36     while (true) { readingRoom.startReading();  readingRoom.stopReading(); }
37   }
38 }
39 class Writer extends Thread {
40   private int nr;
41   private ReadingRoom readingRoom;
42   public Writer(int nr, ReadingRoom readingRoom) {
43     this.nr = nr;  this.readingRoom = readingRoom;
44   }
45   /* @generateHead */
46   private void writing() { }
47   public void run() {
48     while (true) {
49       /* @(.inline infinite, .annotation @@NOTIFY_THREAD)observable */
50       readingRoom.startWriting();
51       /* @observable */
52       writing();  readingRoom.stopWriting();
53     }
54   }
55 }
```

Listing 3. Java source code of the readers and writers problem

performs a number of optimisations to decrease the often high memory and time requirements of model checking.

We have provided four examples of some well–known concurrency problems and and the experiments confirm that our approach provides a valuable aid for Java software verification. In particular, we have shown that our translator together with the tool `Uppaal` or the SAT-based reachability module `BMC4TADD` of `VerICS` works as good as the `Bandera` of `JPF` tools (see 3) or it performs better (see Tables).

References

[1] Symbolic analysis laboratory (SAL). http://sal.csl.sri.com/, 2008.

[2] 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.

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

[4] 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.

[5] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS: A tool for verifying Timed Automata and Estelle specifications. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 278–283. Springer-Verlag, 2003.

[6] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of java programs in javafan. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004.

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

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

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

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

[11] D. Park, U. Stern, J. U. Skakkebaek, and D. L. Dill. Java model checking. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000)*.

[12] 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.

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

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

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