A translator of Java programs to TADDs[⋆]

Artur Rataj[1], Bożena Woźna[2], Andrzej Zbrzezny[2]

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

[2] IMCS, Jan Długosz University of Częstochowa
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland
email: {b.wozna,a.zbrzezny}@ajd.czest.pl

Abstract. The paper presents a `Java`^{TM}`2TADD` translator of the Java programming language. The translator works in two stages. The first one consists in translation of a Java code to an internal assembly language. Then, the resulting assembly code is translated to Timed Automata with Discrete Data, a formalism which is an input language of the tool `VerICS`. We exemplify the use of the translator by means of the following well-known problems written in Java: *alternating bit protocol*, *race condition problem* and *dining philosophers protocol*.

1 Introduction

Model checking [3] is a widely recognised and prominent automatic verification technique both in hardware [9] and protocol verification [7]. 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 reachability properties.

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

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

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 implemented a `Java`^{TM}`2TADD` compiler that translate a Java code to a network of *Timed Automata with Discrete Data* (TADDs) [8, 11]. This 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 tool `VerICS`[2], so we can use this tool to validate a concurrent program written in Java. In particular, we validate the following well-known problems: *alternating bit protocol*, *race condition problem* and *dining philosophers protocol*.

Model checking of Java programs has become increasingly popular during the last decade. However, to the best of our knowledge, there are only two existing model checkers that can verify Java codes: *JavaPathFinder (JPF)* [5, 10] and *Bandera* [4]. Both tools operate on the Java bytecode. On the contrary, we analyse Java programs themselves. Therefore, in the paper we do not provide any comparison of our results with the tools mentioned above.

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 `Java`^{TM}`2TADD` 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. Finally, we present a case study that confirms that our approach provides a valuable aid for Java programs verification.

2 Translation from JAVA to TADD

We translate a concurrent multi-threaded Java program to a network of Timed Automata with Discrete Data (TADDs) [8, 11]. 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 TADD is an abstraction of the state of the Java program, and each transition represents the execution of code transforming this abstract state. The subset of Java that can be translated to 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 and methods, static and non–static methods and synchronisation of methods and blocks. Also, standard thread creation constructs and the following special methods: `Object.wait()`, `Object.notify()`, `Thread.sleep(int)` and `Random.nextInt(int)` are recognised.

A theoretical method of constructing a network of TADDs that model a Java program is shown in [12]. 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: one TADD for each instance of a started thread, and one TADD for each used semaphore. 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.

To implement the translation shown in [12], we first translate a Java code to an internal assembly language (see the next section). Than, the resulting assembly code is translated to TADD; this is presented in Section 4.

3 Internal Assembly Language

This section describes an assembly language that is used internally by our `Java`^{TM}`2TADD` 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 of any type. 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 is the same as that of *field*.

Literals are constant values and can be 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 $\langle\text{value}\rangle ::= \langle\text{literal}\rangle|\langle\text{dereference}\rangle$.

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 to a respective run-time object "this" for the non–static method. A constructor is simply a method, but it is named after the class it constructs.

3.3 Instantiation and polymorphism

In programming, *instantiation* is the creation of a real instance of a run-time object or particular realisation of an abstraction or template such as a class of objects. So to instantiate means to create such an instance by, for example, defining one particular variation of object within a class, giving it a name, and locating it in some physical place. How do we instantiate Java objects in our assembly language. 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`.

Polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. 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 polymorphism rules are defined by the translator, and applied during the parsing process, semantic check and code generation.

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;
– `@@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 basic interpreter 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`.

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(modelchecking.abp.v1.Sender,
            modelchecking.abp.v1.LossyChannel) channel
  ```

  An object `Sender` is created using a constructor that takes an argument of the type `modelchecking.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.
  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 points 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. Note that a non–static method is defined by the run-time type of "this" and not by the variable type. 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 the object of the calling non–static method. The additional parameter 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. This 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>*)`. It is used as a label holder, and can optionally hold operation result and annotations that define the operation.
– `return` – returns from a method.

– `sync begin|end <dereference>` – either the begin or the end of a synchronised code. The dereference references the lock. 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 standard Java unary operators.
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 null for no label.
– A signature, like in Java, is a key containing the method's name and argument types, and does not contain "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 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 Internal Assembly Language to TADDs

Once the assembly code is generated, two stages remain to convert the code to TADDs: interpreting and generating transitions.

4.1 Interpreter

The role of the interpreter is to initialise variables, load needed classes, create objects and threads.
Threads. A main thread is created with the main method as the start one. 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 the beginnings of TADDs, one TADD per thread; Only assignments, allocations, calls and the expressions that can be statically determined are allowed in the main thread.
Library and annotations. The interpreter uses a TADD–specific library that defines the needed Java classes like `Object` or `Thread`. In the library, some methods are marked with special annotations beginning with double '@': The annotations `@@IGNORE`, `@@MARKER` and `@@NEW_THREAD` are used as defined by the assembly language. Additional TADD–specific annotations defined in the library are as follows:

- `@@NOTIFY` – perform `notify()` on an object, used with marker methods, usually only with `Object.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 `Object.wait()`.

Annotations `@@IGNORE`, `@@MARKER` and `@@NEW_THREAD` are used only by the interpreter. Annotations `@@NOTIFY`, `@@RANDOM`, `@@SLEEP` and `@@WAIT` are recognised at the stage of generating transitions, where respective special sets of transitions are generated out of them.

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–type of an object, but by the run-time type of an object. After the interpreter is run and run-time types are known, some calls can be resolved thus. If it is not possible because of a null variable or a variable not initialised at the interpreter stage, an error is reported.
Code flattening. Once the calls are resolved, it is possible to 'flatten' the code. To do so, the code is recursively inlined beginning at a thread's start method, so that there should be no any calls left in the 'flattened' code. This is because no

stack is emulated in the generated TADDs, 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, also unused because of the optimisations.

This step is performed because inlining methods allows for substantial simplifications even if some code was previously optimised.

Ambiguity check. The TADD translator allows for assignments of reference variables by the TADD threads, if the translator can determine that the assignments do not cause a reference to be ambiguous. The translator checks only local reference variables for ambiguity, thus, assignments to non–local reference variables by TADD threads are forbidden.

Ambiguity means here that a variable can have more than a single value. Such an ambiguity of a reference variable would make it impossible to statically dereference the variable. Only dereferences that can be determined statically are allowed, as TADDs do not support dereferences and some 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 multiple assembly language operations that are treated as a whole and as such transformed to respective TADD transitions:

– conditional expression followed by 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 it 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 Case study

In this case study, we apply a SAT-based reachability verification method, which is implemented in the tool `VerICS`, together with our `Java`^{TM}`2TADD` translator to validate representative concurrency examples written in Java: *alternating bit protocol (ABP)*, *race condition problem (RCP)* and *dining philosophers problem (DPP)*. The codes of programs are in [1].

The *reachability problem* consists in establishing whether a state in a set of states satisfying certain (usually undesired) property is reachable from the initial state of a transition system associated to a network of TADDs. In order to apply Bounded Model Checking to test reachability, we unfold the transition relation of a given network of TADDs up to some depth k, and encode this unfolding as a propositional formula. Then, the 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.

We start with describing the ABP protocol together with an error introduced intentionally. Then, we present an example of a Java program in which a race condition occurs. Finally, we discuss the DPP problem together with a well-known solution that could lead to a deadlock. For all the above problems experimental results are shown and commented. All of them have been performed on a computer equipped with the processor Intel Pentium D (3 GHz), 2 GB main memory and the operating system Linux 2.6.24. Moreover, we have set the time-out limit to 15min for RSAT- solver to get the answer.

5.1 Alternating Bit Protocol

The *Alternating Bit Protocol* is a simple communication protocol that is often used as a test case for some tools for the analysis or verification of concurrent systems.

Protocol Description. The protocol sends messages from sender S to receiver R through an unreliable bidirectional communication channel. Each message contains a control bit only. Further, it is assumed that the channel from S to R is initialised and that there are no messages in transit. Since the communication channel transmits bits both from S to R and from R to S, then there are four possible situations that can occur: bit is properly transmitted from S to R; or bit is properly transmitted from R to S; or bit is lost during transmission from S to R; or bit is lost during transmission from R to S.

The protocol works in the following way. When S sends a control bit to R, it does it continuously until it receives an acknowledgement. After that, S flips the control bit and starts all over again. As soon as R receives the control bit that matches its internal control bit, it sends an acknowledgement bit to S and flips his internal control bit. R sends an acknowledgement bit to S continuously until it receives a new control bit (see [1] for a Java source code of ABP).

An introduced error. We have introduced in the Java code of ABP the following error. In the method *get()* of the *LossyChannel* class we have exchanged the condition of the while loop to the following one: `while (!channelEmpty){...}`. This small change causes the protocol to stop working.

The reason for the deadlock is the following: the protocol can reach a state where both threads wait for a notification from another thread, but none of them can invoke the method *notify()*. We can formally show that such a situation indeed exists by checking reachability of a state of the TADD model for ABP that contains ”wait” locations, i.e., locations that belong to automata that are the translation of the methods *wait()* appearing in the synchronised methods *get()* and *put()*.

Experimental results. We have done the reachability test mentioned above and we have found a witness of length 50. It shows that both S and R can move to a waiting state and there is no way for them to move to a ready state. The results are presented in Table 1.

k	variables	clauses	BMC sec	BMC MB	RSAT sec	RSAT MB	SAT
0	366	559	0.0	1.5	0.0	1.3	NO
6	10052	27862	0.2	2.7	0.0	3.4	NO
12	20506	57346	0.5	4.0	0.1	5.9	NO
18	31381	87943	0.8	5.3	0.3	8.3	NO
24	43347	121540	1.2	6.7	0.7	11.0	NO
30	55470	155500	1.5	8.1	1.2	13.7	NO
36	68151	190954	1.9	9.7	3.0	16.5	NO
42	81390	227902	2.2	11.2	3.8	19.4	NO
48	96248	269350	2.7	12.9	9.3	23.2	NO
50	101015	282621	2.8	13.5	82.4	42.9	YES
		In total:	33.2	13.5	122.9	42.9	

Table 1: Checking reachability of a deadlock state of ABP

5.2 Race condition

Problem Description. In most practical multi-threaded applications, 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 (RC)* (see [1] for a Java source code of RC).

An Example of a Race Condition. In our example program 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 exits an execution of the program which ends 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. This is however not the case in our code.

We are able to detect the above race condition by translation of the given Java code 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 experimental results for $n = 1$ are presented in Table 2.

k	variables	clauses	BMC sec	BMC MB	RSAT sec	RSAT MB	SAT
0	507	763	0.0	1.5	0.0	1.3	NO
8	17815	52111	0.5	3.7	0.3	5.3	NO
12	26439	77695	0.8	4.8	0.7	7.4	NO
16	35063	103279	1.1	5.9	3.1	10.0	NO
20	43687	128863	1.3	6.9	9.5	14.1	NO
24	52311	154447	1.6	8.0	32.3	23.1	NO
28	60935	180031	1.9	9.1	43.5	27.6	NO
32	69559	205615	2.1	10.2	170.6	55.4	NO
35	76027	224803	2.4	10.9	784.2	127.3	NO
36	78183	231199	2.4	11.2	97.0	57.2	YES
		In total:	44.0	11.2	2165.5	127.3	

Table 2: Detection of a race condition

5.3 Dining Philosophers problem

Protocol Description. The description of the dining philosophers problem (DPP) we provide below is based on that in [6]. Consider n ($n \geq 2$) philosophers living in a College who spend their lives just thinking and eating. 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, gets up from his chair, exits dining-room and continues thinking; see [1] for a Java source code of DPP.

Discussion. We consider a possible solution of the above problem and implement it as a Java program (see [1]). It is well-known that this solution could lead to a deadlock. This 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 lock 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.

We are able to detect the above deadlock by translation of the given Java code to a network of TADDs and then checking reachability of a state in which all the philosophers hold their left forks. The experimental results are presented in Table 3.

k	variables	clauses	BMC sec	BMC MB	RSAT sec	RSAT MB	SAT
0	564	883	0.0	1.7	0.0	1.3	NO
6	21210	62371	1.1	4.2	0.3	6.1	NO
9	31428	92800	1.6	5.5	2.1	8.9	NO
12	41646	123229	2.2	6.8	16.2	16.3	NO
14	48458	143515	2.6	7.7	130.3	35.7	NO
16	55270	163801	3.0	8.6	900.1	101.9	?
18	62082	184087	3.4	9.4	902.2	140.6	?
19	65488	194230	3.6	9.9	900.6	167.4	?
20	68894	204373	3.7	10.3	9.1	20.7	YES
		In total:	38.4	10.3	4203.6	167.4	

Table 3: Detection of deadlock in the implementation of DPP

6 Summary

In the paper we have presented our `Java`^{TM}`2TADD` translator which together with the verification core of `VerICS` (the SAT-based reachability module) allows to validate well-known concurrency examples written in Java: *alternating bit protocol (ABP)*, *race condition problem (RCP)* and *dining philosophers problem (DPP)*. 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.

References

1. http://www.ajd.czest.pl/∼modelchecking.
2. VerICS. http://verics.ipipan.waw.pl/verics/.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 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 *Proc. of ICSE '00*, pp. 439–448, 2000. ACM Press.
5. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on STTT*, V2(4):366–381, March 2000.
6. C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall, 1985.
7. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
8. A. Janowska and P. Janowski. Slicing of timed automata with discrete data. *Fundamenta Informaticae*, 72(1-3):181–195, 2006.
9. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
10. C. Pasareanu and W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proceedings of SPIN'04*, volume 2989 of *LNCS*, pp. 164–181. Springer-Verlag, 2004.
11. A. Zbrzezny and A. Pólrola. Sat-based reachability checking for timed automata with discrete data. *Fundamenta Informaticae*, 79(3–4):579–593, 2007.
12. A. Zbrzezny and B. Woźna. Towards verification of Java programs in VerICS. *Fundamenta Informaticae*, 85(1-4):533–548, 2008.