Testing Temporal Logic on Infinite Java Traces

Damián Adalid, Alberto Salmerón, María del Mar Gallardo and Pedro Merino

Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga, Málaga, Spain

Abstract. This paper presents an approach for testing reactive and concurrent Java programs which combines model checking and runtime monitoring. We use a model checker for two purposes. On the one hand, it analyzes multiple program executions by generating test input parameters. On the other hand, it checks each program execution against a linear temporal logic (LTL) property. The paper presents two methods to abstract the Java states that allow efficient testing of LTL. One of this methods supports the detection of cycles to test LTL on potentially infinite Java execution traces. Runtime monitoring is used to generate the Java execution traces to be considered as input of the model checker. Our current implementation in the tool TJT uses Spin as the model checker and the Java Debug Interface (JDI) for runtime monitoring. TJT is presented as a plug-in for Eclipse and it has been successfully applied to complex public Java programs.

1 Introduction

LTL is usually employed to check complex behaviors over infinite traces, which are the traces produced by reactive and/or concurrent software. The analysis of a LTL formula along one or several potential infinite execution paths requires the use of algorithms based on automata, like Büchi automata, to recognize special cycles. If cycle detection is avoided, then the kind of formulas that can be checked are restricted to a subset, or their semantics must be adapted to finite executions, like in JavaPathExplorer [1]. In the context of testing and verification of programming languages, the ability to detect cycles depends on the way of storing the global states, which are much larger than in modeling languages. The proposals in [2][3] avoid storing the states at all, so they can perform a partial analysis of very large systems with little memory consumption. Unfortunately, this stateless approach does not allow the analysis of LTL for infinite traces. State-full approaches, like the implemented in Java PathFinder (JPF) [4] and CMC [5], keep a stack with the current execution trace to control backtracking, produce counter examples and detect cycles, in order to check LTL on infinite traces. However, at the time of writing this paper the public distribution of JPF does not detect all infinite loops and cannot check full LTL for any program.

In this paper we propose a method to convert a Java execution trace into a sequence of states that can be analyzed by the model checker Spin [6]. As far as Spin implements the analysis of LTL formulas by translation to Büchi automata, we can check the formulas on Java programs with potential cycles. The Spin stuttering mechanism to deal with finite execution traces allows us to deal with any kind of program without redefining the semantics of LTL. Our conversion of Java traces into Spin oriented traces is

Adalid D., Salmerón A., del Mar Gallardo M. and Merino P..
Testing Temporal Logic on Infinite Java Traces.
DOI: 10.5220/0004127700370047
In *Proceedings of the 10th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems and 1st International Workshop on Web Intelligence* (MSVVEIS-2012), pages 37-47
ISBN: 978-989-8565-14-3
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

based on two efficient abstraction methods of the full state of the program. The counter projection abstracts the Java state by preserving the variables in the LTL formula and adding a counter to distinguish the rest of the state. As long as we do not keep all the information, the counter projection is very efficient at the price of being useful only for a limited subset of LTL. The hash projection abstracts each Java state with the variables in the formula plus a hash of the whole state. The way of constructing the hash makes negligible the probability of conflict for two different states, so we can trust in the Spin algorithm to check LTL based on cycle detection.

Our approach has been implemented in TJT, a tool that combines runtime monitoring and model checking and allows Java application developers to check complex requirements represented with temporal logic in a transparent way.

The rest of the paper is organized as follows. Section 2 presents the architecture of TJT to combine model checking and runtime monitoring. The formalization of the abstraction approach is presented in Section 3. Experimental results of case studies are summarized in section 4. In Section 5 we compare our tool with related proposals. Finally, Section 6 presents some conclusions.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

2 Architecture and Features

TJT is divided in two modules, the test control module and the monitoring module, plus an Eclipse plug-in[1]. Figure 1 shows an overview of this architecture and the workflow of the tool. The programmer must supply two inputs in this workflow: the Java program being tested and a test specification. The latter provides the relevant information for carrying out the tests, including the LTL property that will be checked against execution traces of the program. The test control module, implemented using the model checker Spin, prepares the test and generates a series of test inputs for the Java program as instructed by the test specification. A series of Java Virtual Machines (JVMs) are then launched to test the execution of the program under each generated test input. The executions are controlled by the monitoring module, which detects and reports the events that are relevant to check the LTL formula. Spin processes the information reported by the monitoring module for each execution of the program, and checks if the LTL property stated in the test specification is verified or violated. We discuss some specific issues in the rest of this section, while Section 3 presents a formalization of this approach.

The core of the tool is the test control module, which is implemented with the Spin model checker. While Spin is generally used to check program specifications written in its own Promela language, TJT uses a special Promela specification that can communicate with the monitoring module through sockets. This Promela specification also contains the logic to generate test inputs according to the test specification and launch new program executions. Tests inputs are then generated until all possible combinations that can be inferred from the test specification have been considered. The module then launches the Java program being tested with the given test input under the supervision of the monitoring module. This module uses JDI to watch the events relevant to the

[1] The plug-in and examples are available from http://www.lcc.uma.es/∼salmeron/tjt/

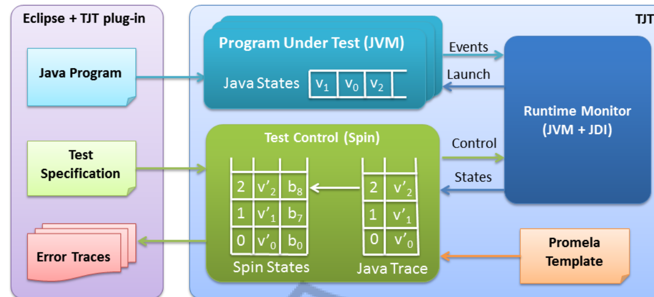

Fig. 1. TJT architecture and workflow.

specified property and sends the information back to the test control module, which reconstructs the current Java execution trace in Spin. Other events such as program termination, exceptions or deadlocks are also communicated through the socket.

Spin checks every execution path using a depth-first search algorithm that maintains a stack of program states ("Spin States" in Figure 1). The state of the Büchi never claim automata, which is used to track the satisfaction of an LTL property, is also stored as part of the global state. Each state in Spin is composed of variables v'_i, which are an abstraction of the Java states v_i, and variables b_i from the Büchi automata. The values of the v'_i variables are obtained from the information sent by the runtime monitoring module. Although the execution of a Java program results in a linear sequence of states, the addition of the Büchi automata may result in several branches that must be explored exhaustively. To support this backtracking, variable values received from the runtime monitoring module are first stored in a Java trace stack ("Java Trace" in Figure 1) and then retrieved from there as needed.

Our tool can check LTL properties on finite traces, deadlocks, uncaught exceptions and some types of cycles. The LTL property can reference class variables present in the Java program, thrown exceptions or breakpoints set in specific locations in the code. The latter are available using the default variables `exception` and `location`. The user may select the intention of the temporal formula, i.e. whether the tests passes when all, none or any of the traces check the property. As the Java executions are usually finite, we take advantage of the stuttering mechanism implemented in Spin [6], and we assume the semantics derived from considering the last state of the trace repeated forever. In that way, we have no limitations to use the LTL formulas supported by Spin. In addition, deadlocks can be detected by the monitoring module by checking the status of every thread before processing each event.

3 Formalization of the Abstraction of Java Traces

Let *Prog* be a Java program and *Var* a numerable set of variable names used by *Prog*. Variables names may be recursively constructed by appending the name of class members to object identifiers. For instance, if o is a reference to an object of class C, and f is an instance variable of C, $o.f$ is the name of variable recording the value of field f in the object instance o.

Values of Java variables may belong to a Java primitive data type ($int, char, \ldots$) or may be a reference if the variable is an object. Let $\mathcal{A}$ and $\mathcal{S}$ be the set of possible memory references and the set of all possible values of Java primitive data types. A state of a Java program is a function $\sigma : Var \to \mathcal{A} \cup \mathcal{S}$ that associates each variable with its value. Let us denote with *States* the set of possible states of a Java program *Prog*. Assume that if h is a variable referencing a thread, then $\sigma(h.cp) \in int$ represents the position of the program counter of h in σ.

Each possible execution of *Prog* may be represented as an infinite sequence of states $t = \sigma_0 \to \sigma_1 \to \sigma_2 \to \ldots \in States^{\omega}$, $States^{\omega}$ being the infinite set of all possible sequences of elements from *States*, called traces. If the sequence is finite, we assume that the last state is infinitely repeated.

Java traces represent possible executions of a given program. However, we do not intend that Spin analyzes complete Java traces. Instead, Spin will be given *projections* of traces, where states are not completely stored. Only, the part of the state that is involved in the evaluation of the formula is transferred to the model checker. We now describe how the projection of Java states is constructed and the correctness relation between the evaluation results regarding the original traces, and the projected ones on Spin.

Definition 1. *Given a subset of variables $V \subset Var$, we define the projection of a state σ onto V as the function $\rho_V(\sigma) : V \to \mathcal{A} \cup \mathcal{S}$ such that $\forall v \in V.\rho_V(\sigma)(v) = \sigma(v)$. Now, given a Java trace $t = \sigma_0 \to \sigma_1 \to \sigma_2 \to \ldots$, we define the projection of t onto $V \subset Var$ as $\rho_V(t) = \rho_V(\sigma_0) \to \rho_V(\sigma_1) \to \rho_V(\sigma_2) \to \ldots$*

We now recall the syntax and semantics of LTL. Let $\mathcal{P}rop$ a set of atomic propositions. The set of LTL temporal formulas may be inductively built using the elements of $\mathcal{P}rop$, the standard Boolean operators, and the *temporal operators*: *next* "$\bigcirc$", *always* "$\Box$", *eventually* "$\Diamond$", and *until* "U".

We assume that given a Java state σ, and an atomic proposition $p \in \mathcal{P}rop$, $\sigma \models p$ represents the result of evaluating p on σ, that is, $\sigma \models p$ holds iff σ satisfies p. In what follows, given a Java trace $t = \sigma_0 \to \sigma_1 \cdots$, we denote with $t_i = \sigma_i \to \cdots$ the suffix of t starting at state σ_i. Consider $p \in \mathcal{P}rop$, and f and g two LTL formulas. We inductively define $\models$ over traces and LTL formulas as follows.

(1) $t_i \models p$ *iff* $\sigma_i \models p$ (2) $t_i \models \neg p$ *iff* $\sigma_i \not\models p$
(3) $t_i \models p \vee q$ *iff* $t_i \models p$ *or* $t_i \models q$ (4) $t_i \models \bigcirc f iff t_{i+1} \models f$
(5) $t_i \models \Box f$ *iff* $\sigma_i \models f$ *and* $t_{i+1} \models \Box f$ (6) $t_i \models \Diamond f$ *iff* $\exists j \geq i.(t_j \models f)$
(7) $t_i \models f\ U\ g$ *iff* $\exists j \geq i.(t_j \models g$ *and* $\forall i \leq k < j.[t_k \models f])$

In the following, given a LTL formula, we denote the set of variables in f as $var(f)$.

Proposition 1. *Given a Java trace t and a temporal formula f, if $V = var(f) \subset Var$ then $t \models f \iff \rho_V(t) \models f$.*

In the rest of the paper, we use the same LTL semantics as Spin, without the *next* operator as usual. Note that in $t \models f$, t may be a prefix of a complete Java trace, i.e. the whole trace may not need to be generated in order to check the satisfaction of a property. As described in section 2, temporal formulas can be used in testing with different use cases. In contrast with model checking, testing works with a subset of program traces instead of every possible trace. Test cases may pass when a property is checked in all,

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

some or none of the given traces. Thus we extend $\models$ for sets of traces and the $\forall$ and $\exists$ quantifier operators.

Definition 2. *Given a temporal formula f and a set of traces T,*

$$T \models \forall f \Leftrightarrow \forall t \in T.t \models f, \quad T \models \nexists f \Leftrightarrow \nexists t \in T.t \models f, \quad T \models \exists f \Leftrightarrow \exists t \in T.t \models f \quad (1)$$

3.1 Dealing with Cycles

Due to the elimination of most program variables in the projected states, it is very likely that a projected trace $\rho_V(t)$ contains many consecutive repeated states. This represents a problem to the model checker since it can erroneously deduce that the original trace has a cycle due to the nested depth first search (DFS) algorithm used by Spin to check properties. Note that this does not contradict Proposition 1, since in this result we do not assume any particular algorithm to evaluate the property on the projected trace. In the following sections, we use relation $\models_s$ to distinguish between the LTL evaluation carried out by Spin through the DFS algorithm, and relation $\models$ defined above.

To correctly eliminate the consecutive repeated states in traces, we propose two techniques discussed now with their effects regarding the preservation results.

SCIENCE AND TECHNOLOGY PUBLICATIONS

State Counting. A simple solution would be to add a new counter variable c to the set of visible variables V. This counter is increased for every new state, thus removing the possibility that Spin erroneously deduce a cycle. Observe that this also precludes Spin from detecting real cycles present in the Java program. We extend the notion of trace projection given in Definition 1, by adding the state counter variable as follows:

Definition 3. *Given a subset of visible variables $V \subset Var$, we define the i-th* counter projection *of a state σ as function $\rho^i_V(\sigma) : V \cup \{c\} \rightarrow \mathcal{A} \cup \mathcal{S}$ defined as $\rho^i_V(\sigma)(v) = \rho_V(\sigma)(v)$, for all $v \in V$, and $\rho^i_V(\sigma)(c) = i$. Now, given a Java trace $t = \sigma_0 \rightarrow \sigma_1 \cdots$, the* counter projection *of t onto V is $\rho^c_V(t) = \rho^0_V(\sigma_0) \rightarrow \rho^1_V(\sigma_1) \cdots$.*

Figure 2 (left) shows the projection of a trace with the addition of the state counter.

State Hashing. In this section, we assume that Java states have a canonical representation, which makes it possible to safely check if two states are equal. We know that canonical representation of states in languages that make an intensive use of dynamic memory is not trivial. We are currently using an extension of the memory representation described in [7], but the actual representation is not relevant for the results obtained in this section. We only need to assume that given two logically equal Java states σ_1 and σ_2, there exists a *matching algorithm* able to check that they are equal.

We use a proper hash function $h : State \rightarrow int$ to represent each state in the projected trace. It is worth noting that since the whole Java states σ do not have to be stored at all we may assume that function h is very precise, producing a minimum number of collisions. That is, $h(\sigma_1) = h(\sigma_2) \Longrightarrow \sigma_1 = \sigma_2$, with a high degree of probability.

Now, we extend the notion of state projection given in Definition 1, by adding the codification of the non-visible part of state as follows:

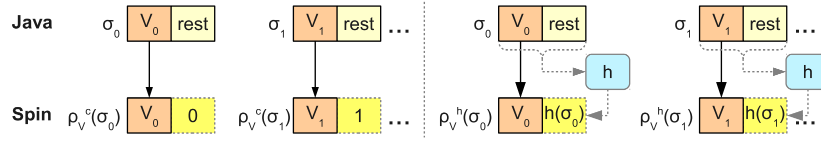

Fig. 2. Trace projection with state counting and state hashing.

Definition 4. *Given a subset of visible variables* $V \subset Var$*, we define the* hash projection $\rho^h_V(\sigma)$ *of a state* σ *onto* V *using the hash function* h *as pair* $\rho^h_V(\sigma) = \langle \rho_V(\sigma), h(\sigma) \rangle$.

Figure 2 (right) shows the projection of a trace with the addition of the state hash. Only projected states $\rho^h_V(\sigma)$ are transferred to Spin. If the model checker detects that two states $\rho^h_V(\sigma_1)$ and $\rho^h_V(\sigma_2)$ are equal, then we can infer that the original states σ_1 and σ_2 are equal with a high degree of probability.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Preservation of Results. We now discuss how the results are preserved regarding the satisfaction of temporal properties in Java and in the projected traces. Here we assume the the algorithm for checking the satisfaction of a property uses the double depth search algorithm as implemented by Spin.

Proposition 2. *Given a temporal formula* f *using only the eventually "$\Diamond$" and until "U" temporal operators, if* $V = var(f) \subset Var$ *then* $t \models f \iff \rho^c_V(t) \models_s f$.

Counter projection ρ^c_V guarantees that no cycle can be deduced by Spin in the projected trace. Thus, properties that do not require the detection of cycles (i. e. those that use only operators *eventually* "$\Diamond$" and *until* "U") can be properly checked over this projection. In contrast, since properties that use the *always* "$\Box$" temporal operator are checked by Spin by searching for cycles, they cannot be analyzed over $\rho^c_V(t)$.

Proposition 3. *Given a temporal formula* f*, if* $V = var(f) \subset Var$ *then*

(1) $\rho^h_V(t) \models_s f \implies t \models f$ *with the degree of probability allowed by h, and*
(2) $t \models f \implies \rho^h_V(t) \models_s f$

This proposition asserts that any temporal formula which is satisfied in the hash projection of a Java trace t, it is also satisfied in the original trace. The converse, while generally true for practical purposes, is limited by the quality of the hash function h. In addition to projecting the variables in f, as established in Proposition 1, the hash projection includes an variable computed by h that identifies the global state and is used to detect cycles in the trace. Our implementation uses the well-known MD5 hashing algorithm for this purpose.

3.2 Folding Consecutive Repeated States

In this section we propose an optimization approach to minimize the number of states of the projected trace that need to be generated and transferred to Spin. To do this, we

slightly modify transition relation $\rightarrow$ defined above by labeling transitions as follows. A Java trace is now given by a sequence of states

$$t = \sigma_0 \xrightarrow{M_0} \sigma_1 \xrightarrow{M_1} \sigma_2 \xrightarrow{M_2} \ldots \in States^{\omega} \tag{2}$$

where each label $M_i \subset Var$ is the set of variables which are modified by the Java sentence that produced the transition.

Recall that counter and hash projections of Java traces t ($\rho_V^c(t)$ and $\rho_V^h(t)$) discard all program variables except the ones in V, which are the only ones needed to check a temporal formula f, $(V = var(f))$, while the rest of the state is collapsed into a single variable. However, Spin does not need to know about states in which none of the variables in V change since they do not affect the evaluation of temporal formulas as described in Propositions 2 and 3. Thus, we propose removing these states from the final projection given to Spin. We call this removed states *folded states*.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Definition 5. *Given an Java trace t as defined in (2), we define the folded counter/hash projection $\phi_V^*(t)$ of t onto $V \subset Var$, where symbol $*$ stands for c or h, as $\phi_V^*(t) = \rho_V^*(\sigma_{i_0}) \rightarrow \rho_V^*(\sigma_{i_1}) \rightarrow \rho_V^*(\sigma_{i_2}) \rightarrow \ldots$ such that $i_0 = 0$, and $\forall k \geq 1$, if $i_{k-1} \leq j < i_k$ then $M_j \cap V = \emptyset$.*

That is, we only project to Spin those states where some visible variable has just modified. However, this definition of folding is not enough to allow a precise cycle detection. If an infinite cycle happens in the folded states, Spin will not be not aware of it, and thus the Java program may loop endlessly. To avoid this, we define the *limited* folding of a counter/hash projection, where *limited* means that the folding between two non-folded states has a limit. To balance the optimization of folding with the needs of cycle detection we introduce a timeout mechanism. After a specified number of folded Java states, we project every Java state onwards, until a non-folded state is reached and the timer is reset. After this limit is reached, the projection behaves like the counter/hash projections in Definitions 3 and 4. An implementation may choose to use a timer as limit instead of a state counter, which does not affect the results given below.

Definition 6. *Given an Java trace t as defined in (2) and a limit l, we define the limited folded counter/hash projection $\phi_V^{*,l}(t)$ of t onto $V \subset Var$, where $*$ stands for c or h, as $\phi_V^{*,l}(t) = \rho_V^*(\sigma_{i_0}) \rightarrow \rho_V^*(\sigma_{i_1}) \rightarrow \rho_V^*(\sigma_{i_2}) \rightarrow \ldots$ such that $i_0 = 0$, and $i_k - i_{k-1} > l$ or $\forall k \geq 1$, if $i_{k-1} \leq j < i_k$ then $M_j \cap V = \emptyset$.*

Figure 3 shows an example of a limited folded hash projection, with limit $l = 1$. This limit ensures that only one state can be folded consecutively. In the figure, a bold M_i label indicates that $M_i \cap V \neq \emptyset$, i.e. i-th transition modifies one or more variables of V. In this example, the limit forces the projection of states σ_4 and σ_5, which should have been folded, resulting in the projected states σ_{i_2} and σ_{i_3}. Also, the limit is reset after projecting states σ_2 and σ_6, which are states where variables in V have changed.

Proposition 4. *Given a temporal formula f using the eventually "$\Diamond$" and until "U" temporal operators, then if $V = var(f) \subset Var$ then $t \models f \iff \phi_V^{c,l} \models_s f$.*

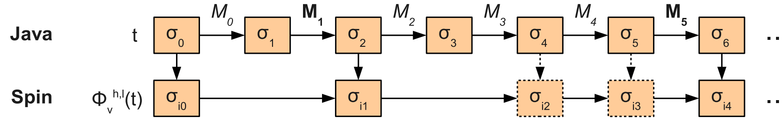

Fig. 3. Example of the limited folded hash projection of a Java trace.

This result is not affected by the folding in the projection, because i) folded states are not required to evaluate boolean temporal formula, and ii) folded states are not required for cycle detection as discussed in Section 3.1.

Proposition 5. *Given a temporal formula* f*, if* $V = var(f) \subset Var$ *then*

(1) $\phi_V^{h,l}(t) \models_s f \Longrightarrow t \models f$ *with the degree of probability allowed by h, and*
(2) $t \models f \Longrightarrow \phi_V^{h,l} \models_s f$

This result is not affected by the folding thanks to the limit, which covers the detection of cycles in the folded states. If there is a infinite cycle in a sequence of states whose transition labels contain no variable in the formula under test, the limited folding only removes a prefix of the sequence, projecting the cycle which will be detected by Spin.

SCIENCE AND TECHNOLOGY PUBLICATIONS

4 Experimental Results

TJT has been successfully used with real Java applications, some of which were also evaluated in [8]. These applications include two servers for FTP and NFS[2]. It is worth noting that, in order to test these servers, we had to implement dummy clients to simulate the behavior required by each test. The temporal formulas used in each test are shown on Table 1. The first application is a concurrent FTP server. One possible test would be to check if the server reacts correctly internally to a poorly implemented client that sends the `CDUP` command (go to parent directory) several times, trying to escape the root of the FTP server. In that case, the server must throw an `FTPDException` and continue attending other requests, as specified in `F1`. We can also test that a `STOR` operation can only be carried out if the client is authenticated, using `F2`. We also have tested a NFS server implemented in Java using a client that tries to mount a directory provided by the server. We can test some conditions related with an incorrect or unauthorized request, by checking if some internal error fields are updated (`F3`) or specific exceptions are thrown (`F4`, `F5`).

Counter Projection. First, we perform these tests using the folded counter projection. The properties used in these tests can be checked using this projection, as they do not require the detection of cycles over infinite traces or can be resolved via stuttering. The results are summarized in the left half of Table 2, averaged over a series of test executions. The third column shows number of projected Java states, while the fourth column indicates the number of state transitions in Spin. The last two columns show the size of a Spin state and the total time of analysis.

[2] From http://peter.sorotokin.com/ftpd and http://www.void.org/∼steven/jnfs/

Table 1. Formulas and atomic propostions for the examples.

(a) Formulas

F1 = all: <>(cdup AND <>(exL1 AND exT1))
F2 = none: <> (stor AND nAuth)
F3 = any: <>[]error
F4 = all: <>(error AND <>exL2)
F5 = none: nError U exL2
F6 = all: []<>one

(b) Atomic propositions

cdup = ftpd.FTPDConnection.status.equals(CDUP)
exL1 = location.equals(ftpd.FTPDConnection:470)
exT1 = exception.equals(FTPDException)
stor = ftpd.FTPDConnection.status.equals(STOR)
nAuth = ftpd.FTPDConnection.authentified.equals(false)
error = nfsServer.MountdHandler.err > 0
nError = nfsServer.MountdHandler.err == 0
exL2 = location.equals(nfsServer.MountdHandler:95)
one = numElements==1

Table 2. Test results using folded counter and hash projections.

		Counter projection				Hash projection			
Application	Formula	Java states	Transitions	State size	Time	Java states	Transitions	State size	Time
FTPD	$F1$	43.7	1027	48 bytes	9.7 s	43	205	76 bytes	16.0 s
	$F2$	59	647	48 bytes	9.5 s	59	265	76 bytes	17.6 s
Java NFS server	$F3$	15.3	94.3	40 bytes	5.1 s	17.6	99	68 bytes	10.8 s
	$F4$	5	140	40 bytes	22.9 s	5	140	68 bytes	27.1 s
	$F5$	4	14	40 bytes	4.6 s	4	14	68 bytes	5.2 s
Lists	$F6$	-	-	-	-	6	33	68 bytes	0.7 s

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Hash Projection. We now present the results of applying the folded hash projection to the same tests. Although the hash projection is not needed to perform any of these tests, it is interesting nonetheless to compare the performance of these two projections. The right half of Table 2 shows these results. The table shows that the hash projection is generally slower, due to the computation penalty associated with visiting the whole Java program state and computing its hash. As these results suggests the tests with more Java states are the ones where the test time is increased the most. To show the usefulness of the hash projection we analyzed a program that adds and removes elements to a list. We check that the list has always exactly one element (F6). This behavior requires the detection of cycles, which is only possible under the hash projection.

5 Comparison to Related Work

The most remarkable tools to analyse Java programs using some variant of full-state based model checking are Bandera [9], JavaPathExplorer and JPF. Bandera is a model extraction based tool that requires the Java program to be transformed into a model composed by pure Promela plus embedded C code. This model is optimized applying a data abstraction mechanism that provides and approximation of the execution traces. As Bandera uses Spin as the model checker, it can check LTL on infinite traces and preserve correction results according to the approximation of the traces. Compared with Bandera, TJT only checks a set of traces. However the use of runtime monitoring avoid model transformation, and the two abstraction methods guarantee the correctness of the results. JavaPathExplorer [1] uses the rewriting-logic based model checker Maude to check LTL on finite execution traces of Java programs. The authors provide a different

semantics for LTL formulas in order to avoid cycle detection. We share with them the idea of using of the model checker to process the stream of states produced by Java. However, our use of Spin allows us to check infinite execution traces.

JPF [10] is a complete model checker for Java programs that performs a complete coverage of a program. In contrast, TJT analyzes each trace in isolation without checking if several traces share already visited states. However due to the implementation approach, we still can take some advantages of reusing the well known model checker Spin instead of building a new one from scratch. Some realistic Java examples of reactive software are not well suited to be verified by JPF. For instance, we tried analyzing our elevator problem with JPF, but it ran out of memory after 58 minutes. The verification of LTL with JPF-LTL in JPF is still under development and has a limited visibility of the program elements to write the formula. According to the information in the JPF-LTL web site, the tool only considers entry to methods in the propositions, and it requires the user to explicitly declare whether the formula should be evaluated for infinite or finite traces. TJT allows a richer set of propositions to be used in the formulas and, due to the stuttering semantics used by Spin, the user does not need to declare whether the trace is finite.

Full-state on-the-fly explicit model checking, as implemented in Spin, requires two main data structures to manage global states. The *stack* is used to store states to support the whole state space exploration using with backtracking, and to find cycles. The *hash table* is used to store all unique states produced when executing instructions in order to prevent wasting computational resources when exploring the state space. The standard distribution of Spin and recent extensions implement many optimizations for these data structures. Our tool TJT inherits these optimizations in the context of Java programs, but we still do not take advantages of the *hash table*. As our hash projection method produces an almost univocal abstraction, in further works we plan to employ this reduced representation of the Java states in Spin's hash table in order to avoid revisiting previously explored states. So, TJT could be considered as a first step towards the whole verification of Java with Spin avoiding the transformation of Java into Promela necessary in tools like Bandera. It is worth noting that our method to produce the stream of states from the real Java execution can be also employed for other programming languages, so the same approach could be useful to extend Spin in order to verify other programming languages apart from the current support for C programs.

6 Conclusions and Future Work

We have presented TJT, a tool for checking temporal logic properties on Java programs. This tool is useful to test functional properties expressed as LTL on both sequential and concurrent programs with finite and infinite execution traces. The formalization of the abstraction approach guarantees the preservation of the results, and the experimental results confirm the applicability to realistic software. Our tool chain includes well known software packages, like the model checker Spin, the JDI API and the well known development environment Eclipse.

Future work will focus on two directions. The first one is to increase the amount of information stored in the Spin's hash table of visited states and the control of scheduling

in Java programs in order to move towards having Spin as model checking tool for Java. The second one is applying our approach to test Android applications, to extend its domain of application. Although these applications are written in Java, the Android JVM does not support some JDI features on which we rely.

Acknowledgements

Work partially supported by projects P07-TIC-03131 and WiTLE2. We thank Franco Raimondi and Michelle Lombardi for their help regarding JPF-LTL.

References

1. Havelund, K., Roşu, G.: An overview of the runtime verification tool java pathexplorer. Form. Methods Syst. Des. 24 (2004) 189–215
2. Godefroid, P.: Model checking for programming languages using verisoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '97, New York, NY, USA, ACM (1997) 174–186
3. Stoller, S. D.: Model-checking multi-threaded distributed Java programs. International Journal on Software Tools for Technology Transfer 4 (2002) 71–91
4. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engg. 10 (2003) 203–232
5. Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., Dill, D. L.: Cmc: A pragmatic approach to model checking real code. In: OSDI. (2002)
6. Holzmann, G. J.: The SPIN Model Checker : Primer and Reference Manual. Addison-Wesley Professional (2003)
7. Gallardo, M. D. M., Merino, P., Sanán, D.: Model checking dynamic memory allocation in operating systems. J. Autom. Reason. 42 (2009) 229–264
8. Fu, C., Milanova, A., Ryder, B. G., Wonnacott, D. G.: Robustness testing of java server applications. IEEE Trans. Softw. Eng. 31 (2005) 292–311
9. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Software Engineering, 2000. Proceedings of the 2000 International Conference on. (2000) 439–448
10. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. STTT 2 (2000) 366–381

SCIENCE AND TECHNOLOGY PUBLICATIONS