Using SPIN for automated debugging of infinite executions of Java programs

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

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

ARTICLE INFO
Article history:
Received 31 January 2013
Received in revised form 8 October 2013
Accepted 26 October 2013
Available online xxx

ABSTRACT
This paper presents an approach for the automated debugging of reactive and concurrent Java programs, combining model checking and runtime monitoring. Runtime monitoring is used to transform the Java execution traces into the input for the model checker, the purpose of which is twofold. First, it checks these execution traces against properties written in linear temporal logic (LTL), which represent desirable or undesirable behaviors. Second, it produces several execution traces for a single Java program by generating test inputs and exploring different scheduling in multithreaded programs. As state explosion is the main drawback to model checking, we propose two abstraction approaches to reduce the memory requirements when storing Java states. We also present the formal framework to clarify which kinds of LTL safety and liveness formulas can be correctly analysed with each abstraction for both finite and infinite program executions. A major advantage of our approach comes from the model checker, which stores the trace of each failed execution, allowing the programmer to replay these executions to locate the bugs. Our current implementation, the tool TJT, uses Spin as the model checker and the Java Debug Interface (JDI) for runtime monitoring. TJT is presented as an Eclipse plug-in and it has been successfully applied to debug complex public Java programs.

© 2013 The Authors. Published by Elsevier Inc. All rights reserved.

1. Introduction
The complexity of current software development is pushing programmers towards more automated analysis techniques, instead of the traditional interactive or postmortem debuggers. For instance, unit testing allows the execution of test cases against a program, checking parts of the code such as single methods or classes in isolation (Runeson, 2006; JUnit testing and framework, xxxx; Beust and Suleiman, 2007; CppUnit framework, xxxx). Runtime monitoring tools usually carry out some controlled executions of instrumented code on real or emulated target platforms (Run-time Monitoring and Checking, (MaC), xxxx; Kraft et al., 2010). Model checking can produce and inspect all possible execution traces of a program, checking the presence or absence of failures along each trace (Clarke et al., 1999; Havelund and Pressburger, 2000; Beyer et al., 2007). In the case of a failure, this technique records a trace to replay the failed execution. To overcome some of the shortcomings of these automatic methods when used in isolation and to extend their domain of application, there have been several proposals that combine a few of them (Peled, 2003; Artho et al., 2003; Havelund and Roşu, 2004). This paper discusses an approach to automated software debugging by the combination of model checking and runtime monitoring. We focus on its application to analyze the (infinite) executions of a given reactive and/or concurrent Java program.

Model checking allows the software developer to describe correctness properties with specification languages such as Temporal Logic (Manna and Pnueli, 1992). The properties could represent safety requirements, like □p (“p is always true”) and p U q (“q will be true, and p will be true in all previous states”), or liveness properties expressed with formulas such as ◯p (“eventually p will be true”), ◯ □ p (“eventually p will be true forever”), and □(p → ◯ q) (“p will always be followed by q”), p and p being any kind of proposition or even temporal formulas. The most common use of LTL is to express complex liveness behaviors of infinite traces, which are the traces produced by reactive and/or concurrent software (Manna and Pnueli, 1992).

In order to check whether or not a program satisfies an LTL formula, model checking algorithms were designed to produce the whole execution graph in a concurrent program and to efficiently detect execution traces violating a formula, presenting these traces as counter-examples. Counter-examples provide the sequence of instructions to the error, and they are the main source of information used to debug the program.
When we do not wish to produce all traces or check liveness properties, other less-expensive approaches, like the use of runtime monitors can be used to check only the subset of LTL representing safety properties. Other monitor based approaches adapt the semantics of full LTL to finite executions, as done by Java Pathfinder (Havelund and Roșu, 2004). The original design of Java Pathfinder only considered finite executions, and to the best of our knowledge, the extension for infinite traces is still not available. Tools such as Verisoft (Godefroid, 1997) and CMC (Stoller, 2002) avoid storing the states of the program during monitoring, so they can perform a partial analysis of very large systems with little memory consumption. Unfortunately, this stateless approach does not permit the analysis of LTL for infinite traces.

The analysis of an LTL formula along one or several potential infinite execution paths cannot be carried out with standard monitors, but requires storing the states of the program and the use of algorithms based on automata to recognize special cycles, like Büchi automata (Clarke et al., 1999). Stateful approaches, like the one implemented in Java Pathfinder (Visser et al., 2003) (JPF) keep a stack with the current execution trace to control backtracking, to produce counter-examples and to check cycles, so they could check LTL on infinite traces. However, at the time of writing this paper, the extension for checking LTL formulas can only detect a few program events. In the following sections we expand on the current status of LTL verification with JPF in a comparison with our proposal.

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 (Holzmann, 2003). We use runtime monitoring to generate just the Spin oriented execution paths from real software, thereby allowing the formulas to be evaluated by Spin. Our work focuses on two major issues of software model checking, analysis of infinite executions and efficient abstraction of execution paths.

As Spin implements the analysis of LTL formulas by translation to Büchi automata, thanks to our method to feed Java executions to Spin as input, we can check the formulas on Java programs with infinite cycles. Furthermore, the Spin stuttering mechanism for dealing with finite execution traces allows us to deal with any kind of program without redefining the original LTL semantics.

In order to address the second issue, the abstraction of execution paths, our conversion of Java traces into Spin oriented traces is based on two efficient abstraction methods of the full state of the program. The counter projection abstracts the Java state by preserving the variables which appear in the LTL formula and adding a counter to distinguish the rest of the state. As we do not keep all the information, the counter projection is very efficient at the cost of being useful only for finite executions. 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 the probability of conflict for two different states negligible, so we can rely on the Spin algorithm to check LTL based on cycle detection. The paper provides a formal study of the correctness of both abstraction methods.

We have implemented the proposed approach in TJF, a tool that combines runtime monitoring and model checking and allows Java application developers to debug programs by checking complex requirements represented with temporal logic in a transparent way: the actual Java program is analyzed on the final target platform without additional modifications by the user, while the test execution is managed in the usual integrated development environment. Specifically, we combine the Spin model checker (Holzmann, 2003) and the runtime debugging API Java Debug Interface (JDI) (Java Platform Debugger Architecture, xxxx). Checking each execution means evaluating a temporal formula representing a failure, over the observable states in the program. Such observable states are provided for Spin by a runtime monitoring module built on top of the JDI support in the Java virtual machine. Both modules are integrated as a new Eclipse (Eclipse development environment, xxxx) plug-in for automatic debugging. TJF stores the failed executions, so that the programmer can later replay them in Eclipse to locate and fix the bugs.

In summary, our method for combining Spin with runtime monitoring offers several advantages to Java developers:

- Linear time temporal logic (LTL) is a compact and rich formalism to represent both failures and desirable behaviors regarding a temporally ordered sequence of events and/or conditions to be checked along one execution. Checking the LTL formula naturally considers the history of the execution, providing clear advantages compared with the isolated evaluation of invariants, assertions or just the values returned by methods.
- Model checking algorithms record the history of the failed execution, which we then use to implement a controlled replay to locate and fix the bugs.
- The support for model checking permits the analysis of potentially infinite executions, which may have two origins. On the one hand, they are produced by reactive software like servers or daemons (e.g. FTP servers and web proxies) which are always in execution responding to interactions with an environment (for instance, client processes). On the other hand, bugs may introduce infinite loops that should not happen. In both situations, model checking can be used to locate the cycles and to decide whether they should be considered as failures.
- The use of runtime techniques removes the extra work required to produce model checking oriented models and makes it possible to start the debugging work directly on the programmer’s code.

We do not intend to perform “full” model checking of Java programs, like Java Pathfinder. Full model checking requires a specific virtual machine to control the Java execution in order to carry out the exploration of all possible execution paths, which is time- and memory-consuming. Our approach consists of using only some features of model checking to have a light automated debugging method that helps the programmer to locate errors. Potential errors are described with temporal logic and we use the capability of model checking to check the temporal logic formula in “one execution path”, even if the execution path corresponds to the infinite behavior of a reactive program. This execution path is naturally produced by the execution of the program in the real environment, with the standard Java virtual machine.

This is a cost-effective application of model checking to program traces that is nevertheless useful for finding faults in concurrent programs and debugging their causes. The use of LTL formulas and a reduced set of variables of interest produces traces as counterexamples, which are very valuable when locating bugs (Clarke et al., 2005). This is specially relevant in concurrent programs where it is usually more important and difficult to find interleaving of actions that produce faults, than failure-inducing test inputs (Balardi et al., 1998). While the analysis of the traces for finding the root cause is still a manual process, the formulas and the selected variables of interest significantly reduce the size of the traces to analyze (Alipour, 2012).

This paper is an extension of previous work of ours presented in (Adalid et al., 2012). In particular, the description of our approach, its implementation, the experimental results and comparison with related work have been significantly expanded.

The rest of the paper is organized as follows. Section 2 introduces the use of model checking for debugging Java programs using a real example. Section 3 presents the architecture of TJF for combining model checking and Java runtime monitoring. The formalization of the abstraction approach and the preservation results are presented in Section 4. Experimental results of the case studies are summarized in Section 5. In Section 6 we compare our tool with related
proposals. Finally, Section 7 presents some conclusions and points of interest for future work.

2. Model checking for debugging

In this section we outline how model checking can be applied for the debugging of real Java programs, as the motivation for the development of our tool. We introduce a real example (an open source FTP server) and show several tests where the use of LTL formulas would be useful. Then we introduce the semantics of the LTL formulas that we consider in our approach, i.e. the usual one for infinite traces. Finally, we discuss how Spin performs the analysis of LTL formulas, translated into Büchi automata.

2.1. Motivating example

To illustrate our proposal we use an FTP server (Sorotkin, xxxx). This server understands the usual commands and can handle several concurrent user connections. We show several tests that a programmer may want to perform on the code of the server, using LTL formulas where the variables and events of the program can be referenced. The formulas are presented in a formal notation, but using helper functions, such as “loc(“)” for checking the program counter location, that are available in our tool. It is worth noting that the following three formulas represent liveness properties to be evaluated on potentially infinite executions, and they cannot be handled by other runtime checkers cited in Sections 1 and 6.

The code in Fig. 1 corresponds to the main loop in the server. The programmer may want to check if the program variables in the loop are correctly cleared between client connections. For instance, to check that the incoming variable is set to null after each iteration we could use the following formula:

\[ \square \ (\text{loc(“FTPServer”}) = 285) \rightarrow \neg \text{(incoming = null)} \]  

This formula states that, after reaching line 285 of the FTPServer.java file in Fig. 1, the incoming variable should be null at some point in the future.

Fig. 2 shows the method that handles CWD (change directory) commands. If a client performs an erroneous request the operation should fail, but the server should recover from the exception and return the appropriate error code (line 471). The programmer may want to check if this code is reachable when a client misbehaves, using the following formula:

\[ \square \ (\text{req}(\text{FTPConnection}. \text{status}, \text{“CDUP”}) \rightarrow \\
\quad \neg \text{(loc(“FTPConnection”} = 470) \)  

Testing these and other properties requires the use of controlled mock clients as part of the test fixture. In this case, the client tries to send several CWD commands (change to parent directory), which are executed as CWD commands in the server and should lead to the behavior described above.

The final condition that we want to check deals with thread scheduling and fairness. In addition to synchronization problems, multithreaded programs are prone to fairness issues: some of the threads may take all the CPU time, leaving others starving. The programmer may want to check whether this is a possible outcome under the default scheduling employed by the JVM or under other schedulings that may be forced in the execution. For instance, we can check the fairness between two clients that compete with each other to interact with the FTP server in a loop.

\[ (\text{req}1 \rightarrow \neg \text{resp}1) \land (\text{req}2 \rightarrow \neg \text{resp}2) \]  

For clarity, in the previous formula we used boolean propositions, such as req1, instead of referencing program variables like in the first two formulas. These propositions refer to auxiliary boolean variables in the FTP clients, i.e. the clientFTP and clientFTP2 classes.

2.2. LTL for Java traces

In this section, we give a formal characterization of LTL formulas for Java, like the three examples used above.

Let \( \text{Prog} \) be a Java program and \( \text{Var} \) an enumerable set of variable names used by \( \text{Prog} \). The 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 the 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, ...) or may be a reference if the variable is an object. Let \( A \) and \( S \) 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 \( \sigma : \text{Var} \rightarrow A \cup S \) that associates each variable with its value. Let us denote with \( \text{States} \) the set of possible states of a Java program \( \text{Prog} \). Assume that if \( h \) is a variable referencing a thread, then \( \sigma(h.cp) \in \text{int} \) represents the position of the program counter of \( h \) in \( \sigma \).
Each possible execution of Prog may be represented as an infinite sequence of states
\[ t = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \in \text{States} \]
(4)

States being the set of all possible infinite sequences of elements from States, called traces.

We now recall the syntax and semantics of LTL. Let \( \mathcal{P}_\varnothing \) be a set of atomic propositions. The set of LTL temporal formulas may be inductively built using the elements of \( \mathcal{P}_\varnothing \), the standard Boolean operators, and the temporal operators: next “\( \varnothing \)”, always “\( \Box \)”, eventually “\( \Diamond \)”, and until “\( \ast \)”. We assume that given a Java state \( \sigma \) and an atomic proposition \( p \in \mathcal{P}_\varnothing \), \( \sigma \models p \) represents the result of evaluating \( p \) on \( \sigma \), that is, \( \sigma \models p \) holds iff \( \sigma \) satisfies \( p \). In what follows, given a (possibly infinite) Java trace \( t = \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \), we denote with \( t_i = \sigma_i \rightarrow \ldots \) the suffix of \( t \) starting at state \( \sigma_i \). Consider \( p \in \mathcal{P}_\varnothing \) and \( f \) and \( g \) two LTL formulas. We inductively define \( \models \) over traces and LTL formulas as follows.

1. \( 1 \models p \) iff \( \sigma_1 \models p \).
2. \( 0 \models p \) iff \( \sigma_0 \models p \).
3. \( q \models (p \lor q) \) iff \( \sigma_i \models p \) or \( \sigma_i \models q \).
4. \( q \models (p \land q) \) iff \( \sigma_i \models p \) and \( \sigma_i \models q \).
5. \( q \models \Box f \) iff \( \sigma_i \models f \) and \( \sigma_{i+1} \models \Box f \).
6. \( q \models \Diamond f \) iff \( \exists j \geq i. (\sigma_j \models f) \).
7. \( q \models f \cup g \) iff \( \exists j \geq i. (\sigma_j \models g \land vi \leq k < j. [\sigma_k \models f] \).

Note that the operator \( \models \) “\( \rightarrow \)” is usually omitted in these rules and transformed into a combination of negation and disjunction.

In what follows, 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. it may not be necessary to generate the whole trace in order to check the satisfaction of a property.

### 2.3. Checking LTL with SPIN

Spin is a well known model checker for analyzing models of software and other complex systems, defined with the Promela language. Promela contains constructions for describing concurrent and non-deterministic behavior which, combined with the right tool, makes it easier to discover unexpected events or interactions which could be difficult to find in the traditional debugging tools available for programming languages.

A Promela model produces a set of possible executions called execution traces or paths. The role of Spin is to look for traces that satisfy or violate a given set of properties. Properties include deadlocks, assertions, code reachability or non-progress loops. However, the most interesting set of properties are complex requirements described with linear temporal logic (LTL) (Manna and Pnueli, 1992).

Spin implements the algorithms by Vardi and Wolper (1986) to check LTL properties, which are based in the translation of the negated LTL formula into a Büchi automaton. A Büchi automaton is defined as a standard automaton that recognizes states in a program to be analysed, but with the addition of final states (also called “accepting states”) that restrict the number of executions allowed by the automaton. In particular, we say that one execution of the program violates the original LTL formula if the corresponding Büchi automaton visits, at least, one of the accepting states infinitely often. This method is well suited to check LTL liveness properties in infinite program executions, and has been adapted in Spin to be used for finite executions as well.

**Fig. 4** contains a simplified graphical representation of the Büchi automaton generated from Formula (3), from the examples above. This automaton is executed synchronously with the Java trace, inspecting the Java states to decide which transition (or transitions) must be taken, and stopping when no transition is possible. A trace is accepted if it contains a finite sequence of states (a cycle).
The model checking algorithm requires the full representation of each state to be included in both data structures (stack and hash table). This might pose a problem for large models, where the number of states to be stored can be higher than $10^{20}$. In order to deal with such large models, Spin has been extended with several optimization techniques, some of which can be used in TJT. Hash-compact (Wolper et al., 1993) reduces the use of memory by compressing the representation of the states without losing information. Bit-state hashing (Holzmann, 1995) represents states as single bits in a hash table, which may lead to a partial analysis of the model in some cases. Currently, work is being carried out in order to obtain parallel versions of Spin that preserve most of these optimizations (Holzmann and Bosnacki, 2007). Finally, there are other strategies that deal with scalability, such as the automatic transformation of the models to implement abstraction methods (Gallardo et al., 2004) or the abstract matching proposed in (de la Cámara et al., 2010).

3. TJT approach for debugging Java executions

This section gives an overview of our approach for debugging Java programs using model checking and runtime monitoring. The main idea is to make Spin handle the states produced by Java instead of the states produced by a regular Promela model. In the standard use of Spin, states are produced by the execution of Promela specifications. Such states include all the local and global variables in the Promela specification and other information, such as the program counters of the processes or the contents of the communications channels. The entire space state generated from the Promela code is managed with the stack and hash table in order to check properties such as deadlocks and LTL formulas. In our particular use of Spin, states are produced by the execution of Java programs. However, in order to reuse Spin features transparently, we still use a special Promela specification that is able to transform sequences of Java states into sequences of Promela-like states. Therefore, we can check complex correctness requirements, like LTL properties, on the Java execution.

Fig. 6 shows an overview of the architecture and the workflow of our tool TJT, which is divided into three modules: the model checking module, the runtime monitoring module and the Eclipse plug-in. The programmer must supply two inputs in this workflow (to the left of the figure): the main entrypoint of the Java program being analyzed and an XML file with the test specification. This main entrypoint may be the real one from the program, or a specific main method associated with a particular test scenario. The test specification includes the correctness requirement: a complex behavior described in a formalism supported by Spin, such as an LTL formula. The user must also declare the objective of the formula, i.e. whether it represents a behavior that should be checked for all traces (desired behavior), none (undesired behavior) or if...
it is enough for any trace to satisfy it (whether that is desired or undesired behavior is up to the programmer). This specification also contains additional information for carrying out the tests, like the program parameters, and their ranges, for generating test inputs.

The model checking module, implemented with Spin and a special Promela template, creates a series of Java Virtual Machines (JVMs) to execute the Java program with all the values considered for the configuration variables. The executions are actually launched and monitored by the runtime monitoring module, which detects the events that are relevant for checking the LTL formula. Each event provokes the creation of a Java state that is sent to the model checking module. Spin processes the information reported by the monitoring module for each execution of the program, and checks the LTL formula. When Spin detects that a Java execution does not match an LTL formula and objective (desired/undesired), it sends information to the Eclipse plug-in in order to show the steps that have led to the failed execution. In the following section we discuss model checking in detail, focusing on efficient methods for abstracting the Java states.

Each Java execution is carried out in the target platform under the control of our runtime monitoring module, which has been implemented in Java using JDI (Java Platform Debugger Architecture, xxx). The monitor and the program being tested run in different JVMs. JDI offers an event based framework, where the application can be notified of certain events in a remote JVM, such as breakpoints, exceptions, changes in object fields or thread states. The monitoring module watches the events relevant to the specified property and sends the information to the model checking module.

At present, our tool can check LTL properties on finite and infinite traces, asserts, and deadlocks. The LTL property can reference class variables present in the Java program, thrown exceptions or breakpoints set at specific locations in the code. When Java executions are finite, we take advantage of the stuttering mechanism implemented in Spin (Holzmann, 2003), and we assume the semantics derived from considering the last state of the trace repeated forever. So, there are no limitations to using the LTL formulas supported by Spin. In addition, deadlocks can be detected by the monitoring module by checking the status of each thread before processing each event.

TJT analyzes each program trace independently. Different traces can be generated by providing information about the input parameters of the program, which will generate different test inputs. These test inputs are currently passed on to the main method of the program as command line arguments. A main method developed specifically for a test may use these arguments to set different parameters in the program, or to execute slightly different test scenarios. In addition, the program may be run more than once with
the same test input, in order to produce different schedulings for threaded programs. In connection with this, we are experimenting with the automatic insertion of calls to methods that alter thread scheduling, e.g. \texttt{yield} and \texttt{sleep}, to cover a greater range of program schedulings.

We have also developed an Eclipse plug-in to make executing tests and reviewing their results more user-friendly. The plug-in includes a form-based editor for creating test specification files, instead of writing error prone XML code. This includes selecting fields to be monitored, setting breakpoints, writing the LTL property to be checked and declaring the test input parameters. Once the specification has been finished, it can be executed within Eclipse and its progress tracked in the TJT console. After the test has finished, a dedicated view shows the erroneous traces that were found, i.e. the execution paths that led to a property violation. Clicking on a trace line takes the user to the corresponding Java line of code. A screenshot of the plug-in is shown in Fig. 7. The tool TJT and several examples can be downloaded from (Adalid et al., xxxx).

3.1. Implementation with Spin and JDI

This section explains our approach, which uses Spin as the core of the model checking module of our debugging tool for Java. In addition to the capability to check properties with Büchi automata, Spin also allows embedding C code in the Promela models, using \texttt{c_code} blocks. These blocks are executed atomically by Spin and may interact with global state variables or call external library functions. The \texttt{c_expr} allows the evaluation of a C expression free from side effects, e.g. to use it as a loop condition. Furthermore, C variables can be treated as if they were part of the global state. Using \texttt{c_track}, existing C variables can be tracked and included in the global state, even as unmatched variables, i.e. they are stored in the stack but not in the hash. Unmatched variables are restored when backtracking, but they are not taken into account when deciding whether two states are equal (see (Holzmann, 2003) for details). Note that states in the stack contain all the information whereas the hash table contains only part of the information. We take advantage of these C oriented features to communicate Spin with the JDI-based monitor, to represent the Java states in Spin, and to implement our abstraction methods for Java states, explained in Section 3.1.3.

As explained above, while Spin is generally used to check program specifications written in its own Promela language, TJT uses a special Promela specification, part of which is shown in Fig. 8, to drive all the automatic debugging work. Such Promela code contains the logic to generate the values for the configuration variables that produce different executions, to communicate with the runtime monitoring module and to check whether a Java execution fails. The code is automatically generated using an initial Promela template and the information provided by the user in the correctness specification file (see Fig. 6). When an LTL formula is present in this file, it is translated into a Büchi automaton, and then included in the resulting Promela specification as a never claim definition. If the formula represents a behavior that must be satisfied in all traces, it is negated first, in order to find counterexamples.

The execution of this Promela code by Spin is summarized in Algorithm 1. This algorithm shows how Spin produces and inspects several Java traces, depending on the potential values for the configuration variables in the correctness specification file. For each combination of input values, Spin launches a new execution and then enters a loop to collect the sequence of Java states for that execution, checking the LTL formula and reporting failed executions to the Eclipse plug-in. This is described in more detail in the following sections.

Algorithm 1. TJT main loop: Spin executing the Promela code in Fig. 8.

The loop represented in Algorithm 1 actually corresponds to the execution (in Spin) of lines 17 to 22 of the code in Fig. 8. The first two functions, \texttt{initialize()} and \texttt{createSocket()}, are executed as if they were a single instruction, using the \texttt{c_code} mechanism. They create all the data and communication structures needed to connect the model checking and the runtime monitoring modules. Note that the communication is done with a socket, so if necessary, e.g. to increase performance, they can run in different computers. To ensure interoperability of these two modules in different nodes, we also use standard XDR-based encoding for the data transferred in this socket.

Each Java execution corresponds to a possible combination of values for configuration variables defined by the user in the
correctness requirement (see Fig. 6). The generation of one combination of values is done by \texttt{generateConfig()}. The function \texttt{execute()} launches the Java program being tested with the given configuration values as test input under the supervision of the monitoring module. When the current Java execution finishes, Spin backtracks to \texttt{generateConfig()} to select another set of values for the configuration variables. Then, \texttt{execute()} is again called to run the program under the new test input. This backtracking-based process continues until no more combinations are possible. The result is the exploration of all the Java executions defined by the programmer in the test specification file.

3.1.2. Getting new Java states and checking failures

The current Java execution trace is reconstructed in Spin thanks to the \texttt{getNextState()} function partially shown in Fig. 9. The next state is either read from the socket with the runtime monitoring module (lines 9 to 16) or retrieved from a list of already visited states in the case of backtracking, as will be explained below. For each new state, we check events such as program termination (line 17) or assertion violations (line 21), which are also communicated through the socket. The current list of failure-related events includes a dozen cases.

The most interesting analysis is checking LTL properties. Spin checks each execution path using a double depth-first search algorithm that maintains a stack of program states (“Spin States” in Fig. 6). The state of the Büchi automaton, which is used to track the satisfaction of an LTL property, is also stored as part of the global state. Each state \( s_i \) handled by Spin is composed of three components \( \langle j, \rho(\sigma_j), b_i \rangle \), where \( b_i \) is the state of the Büchi automata which is executed synchronously with the system, \( \sigma_j \) is the current Java state provided by the runtime monitoring module, and \( \rho \) is a projection function used to simplify the Java states before being analyzed by Spin. The fact that the indexes of \( s_i \) and \( \sigma_j \) are not necessarily equal will be explained below.

Although the execution of a Java program results in a linear sequence of states, the addition of the Büchi automaton representing the LTL formula may result in several branches that must be explored exhaustively. To support this, variable values received from the runtime monitoring module are first stored in a Java trace stack (“Java Trace” in Fig. 6), and then retrieved from there, as needed (as explained above). Therefore, if Spin backtracks during the search, the Büchi automaton will produce new states but the Java states will be a replay of the previously visited states. Note that we have acknowledged this in Fig. 6 by not necessarily showing the same subindex for the whole state \( s_i \) and the corresponding Java part \( \rho(\sigma_j) \).

3.1.3. Representing Java states inside Spin

The main drawback that usually has to be taken into consideration when applying model checking to programming languages is state space explosion: states may be too large and too many to be stored in the memory. Apart from taking advantage of some of the Spin optimization methods described in Section 2, our tool TIT deals with these problems with several novel techniques. The first one consists of selecting the Java states to be sent to the model checking module: we only send those states produced after relevant events in the Java execution. These events include exceptions, deadlocks, update of designated variables, method entry and exit, interactions with monitors, breakpoints, and program termination.

The second optimization consists of abstracting (projecting) the Java state when it is converted to a Spin state. The simplest abstraction method generates a Spin state with only some (a small number) of the variables of the current Java state. These variables are stored in C variables, which are tracked by Spin and part of the global state. In this case, the runtime monitoring module only sends the \( \rho(\sigma_j) \) part
of the original $\sigma_j$ Java state. These selected variables are those that are relevant for checking the user requirements, like the LTL formula. We include one additional variable, the index $j$, in the Spin state, which is useful to retrieve the appropriate Java state $\rho(\sigma_j)$ when backtracking, as shown in Fig. 9.

An alternative, and more complex, abstraction method consists of building an optimized Spin state with all the information in the Java state. Note that this information may be huge, and includes all the variables in objects, thread state and static variables (states $\rho(\sigma_j)$ and $\sigma_j$ Java would be the same). This state is optimized in two steps. First, we collect strings representing the Java hash value for all objects, threads and static variables. Then, we apply the hashing algorithm MD5 to a canonical concatenation of these strings. The result is extremely efficient in both processing time and size of the final state. This abstraction method, called state hashing, is suitable for checking cycles in Spin, and it can be used to detect cycles in the Java program and to check LTL liveness formulas in infinite executions of Java programs.

Both abstraction methods are implemented, making use of the Promela features for embedded C code as explained above. The next section is dedicated to describing and proving the correctness of these abstraction methods supported by TJI.

4. Abstraction of Java traces

In this section we formalize the Java state abstractions mentioned in previous section and which enable the analysis of infinite Java execution traces.

4.1. Definitions

A Java trace $t$, as defined in Section 2.2, represents a possible Java execution of a given program. However, sometimes we do not intend for Spin to analyze complete Java traces. Instead, Spin will be given projections of traces, some Java states are discarded, and the states that are transferred are simplified. 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. In order to simplify the presentation below, we assume that the set of possible data values of program variables, $A \cup S$ contains the integer numbers.

**Definition 1.** [Projection] Given a subset of variables $V \subseteq \text{Var}$, we define the projection of a state $\sigma$ onto $V$ as the function $\rho_V(\sigma) : V \rightarrow A \cup S$ such that $\forall v \in V, \rho_V(\sigma)[v] = \sigma[v]$. Now, given a Java trace $t = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots$, we define the projection of $t$ onto $V \subseteq \text{Var}$ as $\rho_V(t) = \rho_V(\sigma_0) \rightarrow \rho_V(\sigma_1) \rightarrow \rho_V(\sigma_2) \rightarrow \ldots$ \hspace{1cm} (5)

Fig. 10 shows the projection $\rho_V$ of a trace. Observe that $V$ divides each state $\sigma_j$ into two parts: the part concerning the variables of $V$ in state $i (\forall i)$, and the rest (rest$_i$). The projection simply takes the first part from each state and ignores the rest. The effect of this projection is similar to that of the “cone of influence” technique (Clarke et al., 1999). However, while this technique simplifies the code to include only variables which are on the set $V$ (or influence them) before executing it, we execute the program as is and then simplify (i.e. project) the generated trace. We do not automatically include variables not in $V$, though.

As a general result of this definition of projection, if all the variables required for evaluating an LTL formula are present in the projection, the evaluation of the formula is not affected. Let $f$ be an LTL formula and let us denote the set of variables in $f$ as $\text{var}(f)$.

**Proposition 1.** Given a Java trace $t$, a temporal formula $f$ and a subset of program variables $V \subseteq \text{Var}$, if $\text{var}(f) \subseteq V$ then

$t \models f \iff \rho_V(t) \models f$, \hspace{1cm} (6)

As described in Section 3, temporal formulas can be used in debugging with different use cases. In contrast to 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, 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 $T$,

$T \models \forall t \in T.t \models f \iff \rho_V(T) \models f$, \hspace{1cm} (7)

$T \models \neg f \iff \nexists t \in T.t \models f$, \hspace{1cm} (8)

$T \models f \iff \exists t \in T.t \models f$, \hspace{1cm} (9)

4.2. 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 for the model checker since it can erroneously deduce that the original trace has a cycle due to the double depth search (DDS) 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 $\models_{\text{f}}$ to distinguish between the LTL evaluation carried out by Spin through the DDS algorithm, and the satisfaction relation $\models$ defined above.

To correctly eliminate consecutive repeated states in traces, we propose two different techniques that we discuss in the following subsections, along with the corresponding preservation results.

4.2.1. State counting

A simple solution is to add a new counter variable $\text{count}$ to the set of visible variables $V$. This counter is increased for every new state, thus removing the possibility that Spin erroneously finds a non-existing cycle. Observe that this also precludes Spin from detecting
real cycles present in the Java program. This case will be discussed in the following subsection.

We extend the notion of trace projection given in Definition 1, by adding the state counter variable as follows:

**Definition 3.** [Counter projection of states and traces] Given a subset of visible variables \( V \subseteq \text{Var} \) and a fresh variable count \( \# \text{Var} \), we define the \( i \)th counter projection of a state \( \sigma : \mathcal{A} \rightarrow \mathcal{A} \cup S \) as \( \rho^i_{\#} (\sigma) : \mathcal{A} \cup \{ \text{count} \} \rightarrow \mathcal{A} \cup \# \) defined as \( \rho^i_{\#} (\sigma)(v) = \rho_{\#} (\sigma)(v) \) for all \( v \in \mathcal{V} \), and \( \rho^i_{\#}(\sigma)(\text{count}) = i \). Variable count is called state counter of \( \rho^i_{\#} (\sigma) \). Sometimes, we will represent function \( \rho^i_{\#} (\sigma) \) as the pair \( \langle \rho_{\#} (\sigma), i \rangle \).

Now, given a Java trace \( t = \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \) we define the counter projection of \( t \) onto \( V \), \( \rho^i_{\#} (t) \), by projecting each state \( \sigma_i \) with the \( i \)-th counter projection, that is, \( \rho^i_{\#} (t) = \rho^i_{\#}(\sigma_0) \rightarrow \rho^i_{\#}(\sigma_1) \rightarrow \ldots \).

It is worth noting that there is a slight difference in the notation of the counter projection of a trace (\( \rho^i_{\#} (t) \)) and the counter projection of a state (\( \rho^i_{\#} (\sigma) \)). Notice that the former includes a \( c \) superscript, while the latter includes the value of the counter itself as a superscript. Fig. 11 shows the projection of a trace with the addition of the state counter.

### 4.2.2. State hashing

In this section, we assume that Java states have a canonical representation, which makes it possible to safely check whether 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 evaluating an extension of the memory representation described in (Gallardo et al., 2009). But a detailed explanation of this expansion would exceed the goals of this paper. Besides, in this section, the actual representation is not relevant for the results obtained. We only need to assume that given two logically equal Java states \( \sigma_1 \) and \( \sigma_2 \), there exists a matching algorithm able to check that they are equal.

We use a proper hash function \( h : \text{State} \rightarrow \text{int} \) to represent each state in the projected trace. It is worth noting that as not all of the Java states \( \sigma \) have to be stored (we only project the visible part \( \rho_{\#} (\sigma) \)), we may assume that function \( h \) is very precise, producing a minimum number of collisions. That is, \( h(\sigma_1) = h(\sigma_2) \Rightarrow \sigma_1 = \sigma_2 \), with a high degree of probability.

To put this in perspective, we present a very brief study of the MD5 hash function, which we used in our implementation. This function transforms the given input (in this case a string representation of the Java state) into a 128-bit digest. Thus, there are \( 2^{128} \) (or about 3.3 \times 10^{38}) possible values of this function. We are interested in the likelihood of a birthday attack (Schneier, 1995), i.e. the probability of a collision between any two states belonging to the same trace or, conversely, in the number of different states that could be generated before a collision is found with a given probability. For instance, if we assume that a probability \( 10^{-12} \) is enough for our analysis, this number is approximately \( 2.6 \times 10^{13} \). Given a state size of 64 bytes (a reasonable assumption, see Table 2 in Section 5), about \( 1.5 \times 10^{9} \) gigabytes of memory would be required to store this number of states. This is well beyond what current computers carry, and therefore computationally unfeasible. Thus, we conclude that such a hash function is adequate for our uses.

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

**Definition 4.** [Hash projection] Given a subset of visible variables \( V \subseteq \text{Var} \) and a fresh variable hash \( \# \text{Var} \), we define the hash projection \( \rho^h (\sigma) \) of a state \( \sigma \) onto \( V \) using the hash function \( h \) as \( \rho^h (\sigma) : \mathcal{V} \cup \{ \text{hash} \} \rightarrow \mathcal{S} \cup \mathcal{A} \) as follows. \( \rho^h (\sigma)(v) = \rho_{\#} (\sigma)(v) \), for all \( v \in \mathcal{V} \), and \( \rho^h(\sigma)(\text{hash}) = h(\sigma) \). We usually represent function \( \rho^h (\sigma) \) as pair \( \langle \rho_{\#} (\sigma), h(\sigma) \rangle \).

Now, given a Java trace \( t = \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \) we define the hash projection of \( t \) onto \( V \), \( \rho^h (t) \), by projecting each state \( \sigma_i \) with the hash projection, that is, \( \rho^h (t) = \rho^h (\sigma_0) \rightarrow \rho^h (\sigma_1) \rightarrow \ldots \).

Fig. 12 shows the projection of a trace with the addition of the state hash. Only projected states \( \rho^h (\sigma) \) are transferred to Spin. If the model checker detects that two states \( \rho^h (\sigma_1) \) and \( \rho^h (\sigma_2) \) are equal, then we can infer that the original states \( \sigma_1 \) and \( \sigma_2 \) are equal with a high degree of probability.

#### 4.2.3. 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 that the algorithm for checking the satisfaction of a property uses the double depth search algorithm as implemented by Spin. We focus on the preservation of results using the counter and hash projections as described in Definitions 3 and 4, respectively, which were introduced to deal with cycles as required by the model checking algorithm implemented by Spin.

**Proposition 2.** Given a temporal formula fusioning only the eventually "\( \diamond \)" and until "\( U \)" temporal operators, if \( V = \text{var}(f) \subseteq \text{Var} \) then

\[
t \models f \Leftrightarrow \rho^h (t) \models f.
\]

Counter projection \( \rho^h \) does not permit Spin to detect cycles 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 \( \rho^h (t) \).

**Proposition 3.** Given a temporal formula \( f \) and a set of variables \( V \), if \( V = \text{var}(f) \subseteq \text{Var} \) then

\[
\rho^h (t) \models f \Rightarrow t \models f
\]

with the degree of probability allowed by \( h \), and

\[
t \models f \Rightarrow \rho^h (t) \models f.
\]

This theorem asserts that any temporal formula which is satisfied in the original Java trace \( t \), is also satisfied in the hash projection of the 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 a variable computed by \( h \) that identifies the global state and is used to detect cycles in the trace.

### 4.3. 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

\[
t = \sigma_0 \rightarrow M_1 \sigma_1 \rightarrow M_2 \sigma_2 \rightarrow M_3 \ldots
\]

where each label \( M_i \subseteq \text{Var} \) is the set of variables which are modified by the Java statement that produced the transition.

Recall that counter and hash projections of Java traces \( t (\rho^C_v(t) \text{ and } \rho^H_v(t)) \) discard all program variables except the ones in \( V \), which are the only ones needed to check a temporal formula \( f, (V = \text{var}(f)) \), 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 as 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**.

**Definition 5.** [Folded projection] Given a Java trace \( t = \sigma_0 \rightarrow M_1 \sigma_1 \rightarrow M_2 \sigma_2 \rightarrow \ldots \) as defined in \((13)\), we define the folded counter/hash projection of \( t \) onto \( V \subseteq \text{Var} \) as

\[
\phi^C_v(t) = \rho^C_v(\sigma_j) \rightarrow \rho^C_v(\sigma_{j+1}) \rightarrow \ldots
\]

and

\[
\phi^H_v(t) = \rho^H_v(\sigma_j) \rightarrow \rho^H_v(\sigma_{j+1}) \rightarrow \ldots
\]

such that:

1. index \( i_0 = 0 \),
2. for all \( k \geq 0, i_k < i_{k+1} \),
3. for all \( k \geq 1, M_i \cap V \neq \emptyset \),
4. if there exists \( j \geq 0 \) such that \( \forall k \geq 0, i_k \neq j \), then \( M_j \cap V = \emptyset \).

That is, we only project to Spin those states where some visible variable has just been modified.

However, this definition of folding is not enough to allow a precise cycle detection, which was the main reason for introducing the hash projection. If an infinite cycle is located in the folded states, Spin will not be informed of any new Java state, and thus Spin will not be aware that the Java program is going to loop endlessly in those states.

To avoid this, we define the **limited** folding of a hash projection, where the word **limited** means that the folding between two nonfolded states is never greater than a given limit. After a specified number of folded Java states, we project the next Java state, even that state did not change any of the variables \( v \in V \).

An implementation may choose to use a timer as a limit instead of a state counter, which may be more practical and would not affect the results given below. This projection may be further refined in the implementation with an adaptive limit, e.g. a limit which decreases progressively.

**Definition 6.** [Limited folded hash projection] Given a Java trace \( t = \sigma_0 \rightarrow M_1 \sigma_1 \rightarrow M_2 \ldots \) as defined in \((13)\) and a limit \( l > 0 \), we define the limited folded hash projection of \( t \) onto \( V \subseteq \text{Var} \) as

\[
\phi^H_v(t) = \rho^H_v(\sigma_j) \rightarrow \rho^H_v(\sigma_{j+1}) \rightarrow \rho^H_v(\sigma_{j+2}) \rightarrow \ldots
\]

such that:

1. index \( i_0 = 0 \),
2. for all \( k \geq 0, i_k < i_{k+1} \), and
3. for all \( k \geq 1, M_i \cap V \neq \emptyset \), or, the distance between \( j_k \) and \( j_{k-1} \) is limit \( l \), that is \( j_k - j_{k-1} = l \),
4. if there exists \( j \geq 0 \) such that \( \forall k \geq 0, i_k \neq j \), then \( M_j \cap V = \emptyset \).

Although we could define a limited folded counter projection in a similar fashion, there would be no benefit in doing so, since the counter prevents any kind of cycle from being detected.

**Fig. 13.** Example of the folded and limited folded hash projections of a Java trace

---


Please cite this article in press as: Adalid, D., et al., Using SPIN for automated debugging of infinite executions of Java programs. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.056
5. Experimental results

TJT has been successfully used with real Java applications. In this section, we show some example properties evaluated on public open source Java projects, some of which were also evaluated in [Fu et al., 2005]. These applications include three servers: FTP, NFS and HTTP. It is worth noting that, in order to test these servers, we had to implement mock clients to simulate the behavior required by each test. In addition, we studied the elevator problem, a typical example of control software. The temporal formulas used in each test have been gathered together in Table 1. Note that all formulas, with the exception of F4 and F6, represent liveness properties, and, in the case of programs with infinite execution, they can only be analyzed with runtime checkers that implement mechanisms like the cycle detection considered in this paper (see Vardi and Wolper (1986)).

FTP server (http://peter.sorotkin.com/ftp). The first application is the FTP server described in Section 2.1. We tested the three formulas from Section 2.1, shown in Table 1 as formulas F1, F2 and F3, and two additional ones. Of all these formulas, F2 was prepared to uncover a manually introduced bug, while the rest were analyzed over the normal server. F4 deals with security in the server, checking that no user is able to perform a STOR operation without being authenticated first. Finally, formula F5 is a twist on F1, but using the temporal operator “until”. Using this formula we check that, at some point, the variable incoming, which holds the socket just opened for attending the client, should be non-null until a specific thread is created for attending that client.

Elevator problem. The next application is a typical example of concurrency: a shared resource (an elevator) and several clients trying to use it at the same time. This example has been implemented in Java using locks and conditions. Figs. 14 and 15 show part of the elevator and client code, respectively. We can use a temporal formula to check that the elevator does not wait for clients if it is not free (F6). For the sake of illustration, we tested this formula to debug an incorrect implementation, where the wait condition of the server is wrong (line 3 in Fig. 14). We also used this incorrect implementation to test the opposite condition (F7). In both cases, the generated traces led us to this manually introduced error.

NFS server (http://www.void.org/steven/jnfs/). We have also developed an NFS server implemented in Java using a client that tries to mount a directory provided by the server. We can test some conditions related to an incorrect or unauthorized request, by checking whether some internal error fields are updated (F8) or specific exceptions are thrown (F9, F10).

Jibble Web Server (http://www.jibble.org/jibblewebserver.php). Finally, we also studied a Java web server that manages HTTP requests. One possible condition to check would be that the server throws the right exception if it is launched from an incorrect root directory (F11). In addition, we can test whether the server fails to start on any port from a given range, which can be specified as a test input parameter. All executions should thus avoid the location of the exception that would be thrown in this case (F12).

5.1. Counter projection

We performed some of the aforementioned tests using the folded counter projection. It is worth noting that not all of the proposed tests can be carried out using this projection. Formulas that would require cycle detection cannot be checked, as per Proposition 2. However, formulas where the cycle that Spin’s stuttering mechanism creates using the last state is enough for detecting every accepting cycle in the never claim automata generated from the formula. Most of the programs we are debugging are infinite, i.e. they are servers with an infinite reactive loop, and this cannot be checked with finite resources and this projection. We modified these programs to produce finite versions that could be checked with both projections, for the purpose of comparison.

The results are summarized on the left hand side of Table 2, averaged over a series of test executions. The third column shows the

---

Table 1

<table>
<thead>
<tr>
<th>Application</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTP server</td>
<td>F1: all: □(loc(&quot;FTPserver:285&quot;) → □(incoming=null))</td>
</tr>
<tr>
<td></td>
<td>F2: all: □(streq(FTPConnection.status, &quot;C40P&quot;) → □(streq(FTPConnection.requested, &quot;STOR&quot;) ∧ ~FTPConnection.authenticated))</td>
</tr>
<tr>
<td>Elevator</td>
<td>F3: all: □((incoming=0) U (loc(&quot;FTPserver:285&quot;)))</td>
</tr>
<tr>
<td>NFS server</td>
<td>F4: all: □(loc(&quot;Elevator:86&quot;) ∧ -Elevator.isFree)</td>
</tr>
<tr>
<td>Web server</td>
<td>F5: none: □(loc(&quot;ClientConnection:41&quot;))</td>
</tr>
<tr>
<td>Lists</td>
<td>F6: all: □(loc(&quot;WebServer:65&quot;) ∧ 0 ≤ loc(&quot;WebServer:66&quot;)))</td>
</tr>
</tbody>
</table>

---

Fig. 14. Elevator: elevator waiting for clients
number of projected Java states, while the fourth column indicates the number of state transitions in Spin. The next two columns show the size of a Spin state and the total time of analysis. The last column of the table shows an approximation of the size of the Java states, before any projection. This number only takes into account the size of objects allocated in the heap. Since the size of the heap changes dynamically, we report the maximum value we observed during the execution of each program.

The size of states in Spin after the projection is influenced by several factors. First, Spin has an overhead of 16 B for a Promela specification with a single process, and a Büchi automata adds another 8 B. Then, a step integer variable is added to track the Java state that is retrieved in each state (see Section 3.1.3). The variables used for generating test inputs, if any, are added to the global state as well. Furthermore, the counter projection requires an additional variable as described in Section 4.2.1. It is worth noting that the variables being monitored are not part of the global state, but are kept in a separate data structure, in order to support backtracking with minimal impact in the size of Spin states.

TJT also has a deadlock detection algorithm. Although the purpose of formula $F_7$ was to detect the incorrect wait condition, it uncovered a deadlock in each execution of the program detected by the monitoring module. Although deadlocks may be detected while using any property, omitting the temporal formula is recommended when specifically searching for them in order to prevent the trace being terminated early due to a specified property.

5.2. Hash projection

We also evaluated the hash projection, using all the properties described above. Thanks to its cycle detection capabilities, we could use it for more tests than the counter projection. The right hand side of Table 2 shows these results. The table shows that, compared with the counter projection, 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 suggest the tests with more Java states are the ones where the test time increases the most. Also, the difference between the size of Spin states between the counter and the hash projections is constant: the hash projection adds a variable to store the hash of each state, but removes the counter variable. Although the size of the hash proper is 16 B, in our implementation it is stored as a 32 B character array, which explains the total difference of 28 B.

Furthermore, we included some additional tests that required true cycle detection, which is only possible under the hash projection. First, we tested Formula 3 from Section 2.1 (F3) on the FTP server. In addition, we also tested a simple Java program that deals with lists in an infinite loop. The program adds elements to the list and then removes them, and we checked that the list ended up with exactly one element an infinite number of times (F13).

To end our experiments, we performed a small comparison between TJT and the LTL extension for JPF, using the hash projection. Although both tools are based on model checking and can test LTL properties, their scopes are different. JPF performs an exhaustive search over the complete space state of the program, while TJT analyzes a range of execution traces. We compared both tools with some examples available with JPF-LTL, summarizing the results in Table 3. In this table, the first two columns show the name of the example and the formula being analyzed. “Transitions” are the number of state transitions traversed, and “Time” the total time required to check the formula against the program.

### Table 2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FTP server</td>
<td>F1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>47</td>
<td>83</td>
<td>76 B</td>
<td>4.2 s</td>
<td>4.27 MB</td>
</tr>
<tr>
<td>FTP server</td>
<td>F2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>150</td>
<td>187</td>
<td>76 B</td>
<td>14.9 s</td>
<td></td>
</tr>
<tr>
<td>FTP server</td>
<td>F3</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>2419</td>
<td>6211</td>
<td>76 B</td>
<td>34.7 s</td>
<td></td>
</tr>
<tr>
<td>FTP server</td>
<td>F4</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>59</td>
<td>265</td>
<td>76 B</td>
<td>17.6 s</td>
<td></td>
</tr>
<tr>
<td>FTP server</td>
<td>F4*</td>
<td>130</td>
<td>131</td>
<td>48 B</td>
<td>0.8 s</td>
<td>171</td>
<td>172</td>
<td>76 B</td>
<td>0.9 s</td>
</tr>
<tr>
<td>FTP server</td>
<td>F5</td>
<td>19</td>
<td>20</td>
<td>48 B</td>
<td>0.8 s</td>
<td>33</td>
<td>34</td>
<td>84 B</td>
<td>3.9 s</td>
</tr>
<tr>
<td>FTP server</td>
<td>F6</td>
<td>31</td>
<td>219</td>
<td>40 B</td>
<td>28.7 s</td>
<td>31</td>
<td>204</td>
<td>68 B</td>
<td>34.7 s</td>
</tr>
<tr>
<td>FTP server</td>
<td>F6*</td>
<td>3</td>
<td>54</td>
<td>40 B</td>
<td>23.0 s</td>
<td>3</td>
<td>54</td>
<td>68 B</td>
<td>24.1 s</td>
</tr>
<tr>
<td>FTP server</td>
<td>F7</td>
<td>3</td>
<td>51</td>
<td>40 B</td>
<td>29.0 s</td>
<td>3</td>
<td>51</td>
<td>68 B</td>
<td>33.7 s</td>
</tr>
<tr>
<td>NFS server</td>
<td>F8</td>
<td>153</td>
<td>943</td>
<td>40 B</td>
<td>17.6 s</td>
<td>5</td>
<td>140</td>
<td>68 B</td>
<td>10.8 s</td>
</tr>
<tr>
<td>NFS server</td>
<td>F9</td>
<td>5</td>
<td>140</td>
<td>40 B</td>
<td>22.9 s</td>
<td>6</td>
<td>140</td>
<td>68 B</td>
<td>27.1 s</td>
</tr>
<tr>
<td>NFS server</td>
<td>F10</td>
<td>4</td>
<td>14</td>
<td>40 B</td>
<td>4.6 s</td>
<td>4</td>
<td>14</td>
<td>68 B</td>
<td>5.2 s</td>
</tr>
<tr>
<td>Web server</td>
<td>F11</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>211</td>
<td>231</td>
<td>92 B</td>
<td>202 s</td>
<td>6.71 MB</td>
</tr>
<tr>
<td>Web server</td>
<td>F11*</td>
<td>165</td>
<td>175</td>
<td>64 B</td>
<td>8.6 s</td>
<td>258</td>
<td>270</td>
<td>92 B</td>
<td>13 s</td>
</tr>
<tr>
<td>Web server</td>
<td>F12</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>135</td>
<td>146</td>
<td>96 B</td>
<td>214 s</td>
<td></td>
</tr>
<tr>
<td>Web server</td>
<td>F12*</td>
<td>168</td>
<td>179</td>
<td>68 B</td>
<td>9.1 s</td>
<td>273</td>
<td>284</td>
<td>96 B</td>
<td>13.5 s</td>
</tr>
<tr>
<td>Lists</td>
<td>F13</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>6</td>
<td>33</td>
<td>68 B</td>
<td>0.7 s</td>
<td>0.14 MB</td>
</tr>
</tbody>
</table>

1 Formula tested over a program where errors were manually introduced.
2 Formula tested over a finite version of the original infinite program.
It is worth noting the disparity in time and space required for the analysis of the second formula with TJT, compared to the other two. This program deals with random number generators, and the property requires cycle detection. Although checking whether or not a single trace violates the property is relatively quick, the first few traces generated and analyzed by TJT did not violate the property. Thus, when a violating trace was generated, the cost of the analysis had accumulated the analysis of the previous traces.

6. Comparison with related work

The most notable tools for analyzing Java programs using some variant of full-model state model checking are Bandera (Corbett et al., 2000) and Java PathFinder (Visser et al., 2003). 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 by applying a data abstraction mechanism that provides an 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 to avoid model transformation, and the two abstraction methods guarantee the correctness of the results.

Java PathFinder (Havelund and Pressburger, 2000) (JPF) is a complete model checker for Java programs that performs a complete coverage of a program, while our testing tool does a partial analysis of the program. In addition, thanks to a matching mechanism, JPF does not revisit the same execution path twice, while TJT analyzes each trace in isolation without checking whether several traces share already visited states. However due to our integration approach, we can still gain some advantages from reusing the well known model checker Spin, instead of building a new one from scratch. Some realistic Java examples of reactive software are not suitable for verification 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 for writing the formula. At the time of writing, JPF-LTL only considers entry to methods in the propositions, and it requires the user to explicitly declare whether the formula should be evaluated for finite or infinite 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 or not.

The specification of LTL properties to analyze programming languages at runtime has been proposed by other authors, which we discuss in the rest of this section. Probably, the most complete overview of the approaches can be found in a paper by Bauer et. al (Bauer et. al., 2011). Bauer et. al consider the runtime verification of LTL and tLTL (timed LTL) with a three-valued semantics (with truth values true, false, inconclusive) suitable to check whether or not a partial observation of a running system meets a property. They generate deterministic monitors to decide the satisfaction (or not satisfaction) of a property as early as possible. They use these three-values as a way to adapt the semantics of LTL to the evaluation of finite traces. The authors write that “the set of monitorable properties does not only encompass the safety and cosafety properties but is strictly larger”. However, the general case of liveness properties for infinite traces is not considered. Compared with our work, they develop the foundations to create monitors to support the new semantics of LTL for infinite traces, while our work relies on the already existing algorithms and tools to check Büchi automata for infinite traces.

Java PathExplorer, developed by Havelund and Roşu (Havelund and Rosu, 2004), uses the rewriting-logic based model checker Madoka to check LTL on finite execution traces of Java programs. The authors provide different semantics for LTL formulas in order to avoid cycle detection. Java PathExplorer also supports the generation of a variant of Büchi automata for finite traces developed by Giannakopoulou and Havelund (Giannakopoulou and Havelund, 2001). We share with Java PathExplorer the idea of using the model checker to process the stream of states produced by Java. However, our use of Spin allows us to check infinite execution traces.

The tool Temporal Rover (Druisinsky, 2000) can check temporal logic assertions against reactive systems (with non-terminating loops) at runtime. The author considers that both finite and infinite traces are possible. However, only finite traces are evaluated, and a default fail value is returned for formulas like $\varphi p$ when $p$ has not been satisfied at the end of the trace and there is no evidence that the program has terminated. TJT can provide a conclusive verdict when inspecting the infinite trace.

Boddien (Boddien, 2004) uses Aspect to implement a method to evaluate LTL, inserting pieces of Java code to be executed at points where the behavior specified by the formula is relevant and must be evaluated. This method is useful to check only safety properties. d’Amorim and Havelund (d’Amorim and Havelund, 2005) have developed the tool HAWK for the runtime verification of Java programs, which allows the definition of temporal properties with the logic EAGLE. In addition, the user must supply a method that must be called when the program terminates in order to produce a finite trace. FILM (Finite LTL runtime Monitor) (Zhang et al., 2009) also gives a specific semantics to LTL to check both safety and liveness in finite traces. However, in the case of liveness, manual inspection is required when the tool reports a potential liveness violation. All these tools for runtime monitoring of LTL are focused on finite traces. The main difference with TJT is the support of cycle detection due to the way in which the states are abstracted and stored, and the use of Büchi automata.

Note that we have not included further experimental comparison of TJT with some of these runtime monitoring tools due to the lack of comparable public examples, or of the tools themselves.

7. Conclusions and future work

We have presented the foundations of TJT, a tool for checking temporal logic properties on Java programs. This tool is useful for testing functional properties on both sequential and concurrent programs. In particular, we explained how the use of Büchi automata combined with storing the states from runtime monitoring can be used to check liveness properties in non-terminating executions of reactive programs.

Our toolchain includes the model checker Spin and JDI, which are integrated in the well known development environment...
Eclipse. The use of JDI instead of instrumented code makes it possible to detect deadlocks and provides wider access to events in the execution of the program, while being completely transparent.

Our current work follows several paths. One is to apply static influence analysis to automatically select the variables relevant to the given property, as we proposed in [de la Cámara et al., 2006]. The second one is to implement methods to produce more scheduling in multithreaded programs for the same initial state. Finally, we plan to take advantage of multicore architectures to speed up the analysis, due to the already decoupled interaction between Spin and JDI modules.

Acknowledgements

Work partially supported by projects P11-TIC-07659 (Regional Government of Andalusia), TIN2012-35669 (Spanish Ministry of Economy and Competitiveness) and UMA-806/47.3154-1 (AT4Wireless). We thank Franco Raimondi and Michelle Lombardi for their help regarding JPF-LTL.

References

CppUnit framework. (http://sourceforge.net/apps/mediawiki/cppunit/).
Eclipse development environment. (http://www.eclipse.org/).
Havelund, K., Pressburger, T., 2010. Model Checking Java Programs using JPF PathFinder. STTT 2, 366–381.
Java Platform Debugger Architecture. (http://java.sun.com/javase/technologies/core/tools/jpda/).
JUnit testing framework. (http://www.junit.org/).
Run-time Monitoring and Checking (MaC). (http://rtc.cis.upenn.edu/mac/).
Sorotokin, P. FTP server v64.2. (http://ftp.sorotokin.com/ftp/).
Damían Adalid is a software engineer at the University of Málaga’s Software Engineering Group. His research interest include the development of tools for software correctness analysis, by applying testing and model checking technologies. Adalid has a MS in telecommunications and networking from the University of Málaga.
Alberto Salmerón is a PhD candidate at the University of Malaga’s Software Engineering Group. His research interests include model-driven engineering and formal methods applied to communications software. Salmerón has an MS in computer science from the University of Málaga.
Dr. María del Mar Gallardo is an associate professor at the University of Malaga (Spain). Her research interests involves the development and use of formal methods and testing and model checking technologies to improve the quality of software. She has participated in several Spanish research projects related with this theme.
Dr. Pedro Merino is an associate professor at University of Malaga. His research interests are foundations, tools, and applications of formal and rigorous methods for critical systems, particularly communications software and development techniques for Internet services on mobile networks. Pedro Merino was the chair of the ERCIM Working Group on Formal Methods for Industrial Critical Systems, and member of the Executive Committee of ERCIM, one of the larger European research consortium in ICT.