Formal Methods in System Design, 24, 189–215, 2004

An Overview of the Runtime Verification Tool Java PathExplorer

KLAUS HAVELUND havelund@email.arc.nasa.gov
Kestrel Technology, NASA Ames Research Center, California, USA

GRIGORE ROŞU* grosu@cs.uiuc.edu
Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, USA

Abstract. We present an overview of the Java PathExplorer runtime verification tool, in short referred to as JPAX. JPAX can monitor the execution of a Java program and check that it conforms with a set of user provided properties formulated in temporal logic. JPAX can in addition analyze the program for concurrency errors such as deadlocks and data races. The concurrency analysis requires no user provided specification. The tool facilitates automated instrumentation of a program's bytecode, which when executed will emit an event stream, the execution trace, to an observer. The observer dispatches the incoming event stream to a set of observer processes, each performing a specialized analysis, such as the temporal logic verification, the deadlock analysis and the data race analysis. Temporal logic specifications can be formulated by the user in the Maude rewriting logic, where Maude is a high-speed rewriting system for equational logic, but here extended with executable temporal logic. The Maude rewriting engine is then activated as an event driven monitoring process. Alternatively, temporal specifications can be translated into automata or algorithms that can efficiently check the event stream. JPAX can be used during program testing to gain increased information about program executions, and can potentially furthermore be applied during operation to survey safety critical systems.

Keywords: runtime verification, trace analysis, temporal logic, rewriting logic, Maude, automata, dynamic programming, program instrumentation, deadlocks, data races, Java

1. Introduction

Correctness of software is becoming an increasingly important issue in many branches of our society. This is not the least true for NASA's space agencies, where spacecraft, rover and avionics technology must satisfy very high safety standards. Recent space mission failures have even further emphasized this. Traditional ad-hoc testing of software systems still seems to be the main approach to achieve higher confidence in software. By traditional testing we mean some manual, and at best systematic, way of generating test cases and some manual, ad-hoc way of evaluating the results of running the test cases. Since evaluating test case executions manually is time consuming, it becomes hard to run large collections of test cases in an automated fashion, for example overnight. Hence, there is a need for generating test oracles in an easy and automated manner, preferably from high level specifications. Furthermore, it is naive to believe that all errors in a software system can be detected

*Supported in part by joint NSF/NASA grant CCR-0234524.

before deployment. Hence, one can argue for an additional need to use the test oracles for monitoring the program during execution. This paper presents a system, called Java PathExplorer, or JPAX for short, that can monitor the execution of Java programs, and check that they conform with user provided high level temporal logic specifications. In addition, JPAX analyzes programs for concurrency errors, such as deadlocks and data races, also by analyzing single program executions. In this paper we shall, however, primarily focus on presenting the temporal logic checking aspect of JPaX.

The algorithms presented all take as input an execution trace, being a sequence of events relevant for the analysis. An execution trace is obtained by running an instrumented version of the program. Only the instrumentation needs to be modified in case programs in other languages than Java need to be monitored. The analysis algorithms can be re-used. A case study of 35,000 lines of C++ code for a rover controller has for example been carried out, as will be explained.

Concerning the first form of analysis, *temporal logic verification*, we consider two forms of logic: future time temporal logic and past time temporal logic. We first show how these can be implemented in Maude [7–9], a high-performance system supporting both rewriting logic and membership equational logic. The logics are implemented by providing their syntax in Maude's very convenient mixfix operator notation, and by giving an operational semantics of the temporal operators. The implementation is extremely efficient. The current version of Maude can do up to 3 million rewritings per second on 800 MHz processors, and its compiled version is intended to support 15 million rewritings per second. The Maude rewriting engine is used as an event driven monitoring process, performing the event analysis. The implementation of both these logics in Maude together with a module that handles propositional logic covers less than 130 lines. Therefore, defining new logics should be very feasible for advanced users. Second, we show how one from a specification written in temporal logic (be it future time or past time) can generate an observer automaton or algorithm that checks the validity of the specification on an execution trace. Such observer automata/algorithms can be more efficient than the rewriting-based implementations mentioned above. It especially removes the need for running Maude as part of the monitoring environment.

Concerning the second form of analysis, *concurrency analysis*, multi-threaded software is the source of a particular class of transient errors, namely deadlocks and data races. These errors can be very hard to find using standard testing techniques since multi-threaded programs are typically non-deterministic, and the deadlocks and data races are therefore only exposed in certain "unlucky" executions. Model checking can be used to detect such problems, and basically works by trying all possible executions of the program. Several systems have been developed recently, that can model check software, for example the Java PathFinder system (JPF) developed at NASA Ames Research Center [20, 41, 42], and similar systems [3, 11, 16, 28, 40]. This can, however, be very time and memory consuming (often program states are stored during execution and used to determine whether a state has been already examined before).

JPAX contains specialized trace analysis algorithms for deadlock and data race analysis, that from a single random execution trace try to conclude the presence or the absence of deadlocks and data races in other traces of the program. The deadlock algorithm is an improvement of the deadlock algorithm presented in [17] in that it minimizes the number of false positives [4]. The data race algorithms include the Eraser algorithm [38] for detecting low-level data races, and a new algorithm for detecting high-level data races [2]. These algorithms are based on the derivation of testable properties that are stronger than the original properties of deadlock freedom and data race freedom, but therefore also easier to test. That is, if the program contains a deadlock or a data race then the likelihood of these algorithms to find the problem is much higher than the likelihood of actually meeting the deadlock or data race during execution.

The idea of detailed trace analysis is not new. Beyond being the foundation of traditional testing, also more sophisticated trace analysis systems exist. Temporal logic has for example been pursued in the commercial Temporal Rover tool [12], and in the MaC tool [30]. Temporal Rover allows the user to specify future time and past time temporal formulae, but requires the user to manually instrument the code. The MaC tool supports past time temporal logic and is closer in spirit to what we describe in this paper due to its automated code instrumentation capability. However, its specification language is fixed. Neither of these tools provides support for concurrency analysis. A more recent approach using alternating automata for checking future time temporal logic is presented in [14]. A tool like Visual Threads [17, 38] contains hardwired deadlock and low-level data race analysis algorithms, but only works on Compaq platforms and only on C and C++. Furthermore, Visual Threads cannot be easily extended by a user, while JPAX can. We furthermore have improved the deadlock analysis algorithm to yield fewer false positives, an important objective if one wants such a tool to be adopted by programmers, and we have added a high-level data race detection algorithm. JProbe [39] is a commercial tool performing deadlock and low-level data race analysis on Java code. For an overview of recent work in runtime verification we refer the reader to the proceedings of RV'01 and RV'02, the 1st and 2nd workshops on runtime verification [22, 26].

This paper is a summary of several papers written on Java PathExplorer [2, 4, 18, 19, 21, 23–25, 27, 35, 36]. A rewriting implementation for future time Linear Temporal Logic (LTL) is presented in [25, 36]. A rewriting implementation of past time LTL is presented in [24]. Generation of observer automata for future time LTL is described in [36]. Generation of observer dynamic programming algorithms for past time LTL is presented in [27]. Concurrency analysis is studied in [2, 4, 18]. In [15] a framework is described for translating future time LTL to Büchi-like automata.

The paper is organized as follows. Section 2 gives an overview of the JPAX system architecture. Section 3 introduces the temporal logics that have been implemented in JPAX, namely future time and past time temporal logic. Each logic is defined by its syntax and its semantics. Section 4 presents the various algorithms for monitoring the logics. First (Section 4.1) the rewriting based approach is presented, showing how future time and past time logic can be encoded in the Maude rewriting system, and how Maude is then used as the monitoring engine. Second (Section 4.2) the observer automata/algorithm approach is presented, showing how efficient observer automata and algorithms can be generated from future time respectively past time temporal logic. Section 5 presents experiments that have been performed using temporal logic monitoring as well as concurrency analysis. Finally, Section 6 concludes the paper.

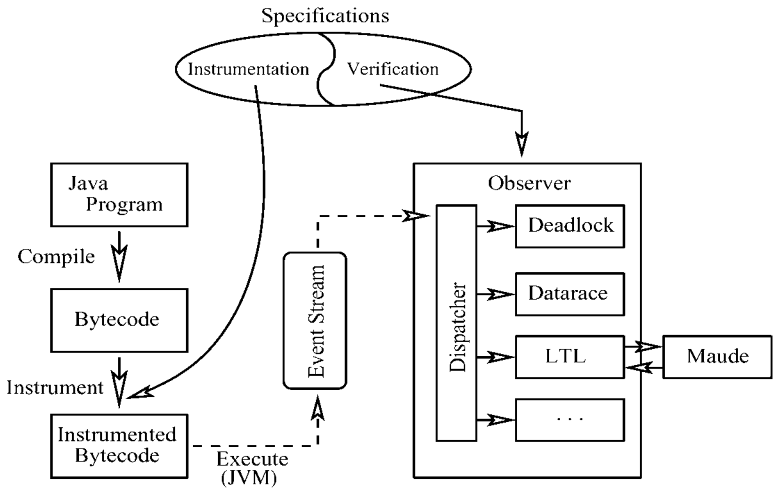

Figure 1. Overview of JPaX.

2. Architecture of JPAX

Java PathExplorer (JPaX) is a system for monitoring the execution of Java programs. The system extracts an execution trace from a running program and verifies that the trace satisfies certain properties. An execution trace is a sequence of events, of which there are several kinds as we shall discuss below. Two forms of monitoring are supported: *temporal verification* and *concurrency analysis*. JPAX itself is written in Java and consists of an *instrumentation module* and an *observer module* (see figure 1). The instrumentation module automatically instruments the bytecode class files of a compiled program by adding new instructions that when executed generate the execution trace. The events (forming the execution trace) are either written to a file or to a socket, in both cases in plain text format. The observer will read the events correspondingly. In case a socket is used, the observer can run in parallel with the observed program, even on a different computer, and perform the analysis in real time.

In temporal verification the user provides a specification in temporal logic of how the observed system is expected to behave. The models of this specification are all the execution traces that satisfy it. Finite execution traces generated by running the program are checked against the specification for model conformance, and error messages are issued in case of failure. The temporal logic is built over atomic predicate names, referred to as propositions, that have been connected to predicates over entities in the observed program, similar to [30]. The following example illustrates a very simple specification:[1]

```
// instrumentation:
     monitor C.x;
     proposition A is C.x > 0;
```

```
// verification:
     formula F1 is []A
```

It states that we want to monitor the variable `x` in class `C`, and in particular that we want to monitor the predicate `C.x > 0`, referring to it by the atomic proposition `A`. The formula, `[]A`, which states that `A` is always true, is the property that is monitored by the observer. Such formulae can only refer to atomic propositions, and not to general predicates over the program's variables. The program is instrumented to emit events signaling updates of the proposition `A`. Essentially, the observer will receive the initial value of each proposition, and then just the proposition each time its value toggles. Hence suppose for example during a program execution, that `C.x` initially is 0 and that it then changes to 1, then to 2, then to -1 and then to 3. The event stream observed will be the following sequence consisting of four events: $[(A, \mathit{false}), A, A, A]$, corresponding to the real events `C.x = 0` (initialization), `C.x = 1`, `C.x = -1`, and `C.x = 3`. Note that the assignment `C.x = 2` does not create a toggle since the predicate does not change value (it is true and stays true). The observer receives this toggle sequence and maintains in a local data structure the current value of `A`. The observer performs the continuous check of the formula `[]A`, which obviously in this case gets violated, initially, as well as at the third event.

In concurrency analysis, we are analyzing the trace for symptoms of concurrency problems, such as deadlocks and data races. Events in this case represent the taking or releasing of locks (needed for deadlock analysis as well as for data race analysis), and accesses to variables (needed for data race analysis).

2.1. *Instrumentation module*

The Java bytecode instrumentation is performed using the JTrek Java bytecode engineering tool [10]. JTrek makes it possible to easily process Java class files (bytecode files), facilitating the examination of their contents and insertion of new code. The inserted code can access the contents of various runtime data structures, such as for example the call-time stack, and will, when eventually executed, emit events carrying this extracted information to the observer. It works as follows. JTrek is instructed to perform a "trek" through a program's byte code instructions. For each class (within a user defined scope), and for each method in that class, JTrek sequentially walks though the byte codes. At each byte code, JTrek calls a method `void at(Instruction instr)`, which the user of JTrek can override, and to which the current instruction is passed as parameter. The instruction is represented by an object of the JTrek specific class `Instruction`, which has methods for accessing its opcode (identifying which instruction it is), its arguments, and other information related to the instruction. JTrek generally provides a variety of classes, each representing a Java concept that can be accessed from an instruction, such as for example `Statement` and `Method`, each with methods yielding various kind of information, that is either atomic (like integers or strings), or other objects that can be used for further navigation. For example, in the `Instruction` class, the method `Statement getStatement()` returns an object of the class `Statement`, representing the statement in which the instruction occurs. The `Statement` class in turn contains a method, `Method getMethod()`, returning the method in which the statement occurs. In this manner, as an example, the method in which an instruction `instr` occurs can be obtained by the expression: `instr.getStatement().getMethod()`.

In the `void at(Instruction instr)` overridden method defined in JPaX, a switch-statement branches out depending on what is the opcode of the instruction. In case an instruction is fit for instrumentation, JTrek allows to insert the call of a method either after or before the instruction. For each kind of bytecode that we want to instrument we have defined a class that contains essentially two methods: `void instrument(Instruction instr)`, which performs the required instrumentation by inserting a call to the second method, `void action(...)`, which when executed will emit the relevant information to the observer. The `action` method takes as parameter various information, such as for example the current contents of the call-time stack, typically representing what object is operated on, and in case of field updates, what value to be stored.

For temporal logic monitoring, the observer expects to receive a sequence of events, each of the form *toggle(propset)*, where *propset* is a set of propositions, who's values have toggled. In order to emit such events, the instrumentation package maintains a state that keeps track of the current value of each monitored proposition. At each instruction that updates a variable that is monitored, code is inserted that evaluates the value of the involved predicates (those that refer to the variable), and in case the value of a predicate changes, the corresponding proposition is added to the current event. When all relevant predicates have been evaluated, the combined event, which is now a set of propositions (who's values have changed), is sent to the observer.

For monitoring of concurrency errors, such as deadlocks and data races, we need to extract information about when locks are taken and released. For a `monitorenter` instruction for example, which signals that a thread takes a lock when entering a `synchronized` statement, we extract what object is locked and what thread does it, and similarly when `monitorexit` is executed. This gives rise to the following events: $lock(t, o)$ (thread t locks object o) and $unlock(t, o)$ (thread t unlocks object o). For data race analysis, information is additionally needed to convey when variables are accessed and by which threads.

2.2. *Observer module*

The observer module is responsible for performing the trace analysis. It receives the events and dispatches these to a set of observer rules, each rule performing a particular analysis that has been requested. Generally, this modular rule based design allows a user to easily define new runtime verification procedures without interfering with legacy code. Observer rules are written in Java, but can call programs written in other languages, such as for example Maude. Maude plays a special role in that high level requirement specifications can be written using equational logic, and the Maude rewriting engine is then used as a monitoring engine during program execution. More specifically, we implement various temporal logics in Maude, for example Linear Temporal Logic (LTL), by writing an operational semantics for each logic, as will be explained in the remainder of the paper. Maude can then be run in what is called *loop mode*, which turns Maude into an interactive system that can receive events, one by one, perform rewriting according to the operational semantics of the logic, and then wait for the next event. As will be explained in the paper, we have also implemented a monitor rule *synthesis* capability, that translates a collection of future and past time temporal formulae into a Java program, that monitors conformance of the trace with the formulae. Maude can here be used as a translator that generates the observer programs.

3. Temporal logic as a monitoring requirements language

Temporal logics are routinely used to express requirements to be proved or model checked on software or hardware concurrent systems. We also find them worth investigation as candidates for a monitoring requirements language. In this section we discuss future and past time temporal logic variants whose models are finite execution traces, as needed in monitoring, rather than infinite ones. Since a major factor in the design of JPAX and its underlying theory was efficiency, and since we were able to devise efficient algorithms for future and past time temporal logics regarded separately, in the rest of the paper we investigate them as two distinct logics.

Both future time and past time temporal logics extend propositional calculus, which can be described by the following syntax:

$$F ::= \mathsf{true} \mid \mathsf{false} \mid A \mid \neg F \mid F \vee F \mid F \wedge F \mid F \oplus F \mid F \rightarrow F \mid F \leftrightarrow F.$$

A is a set of propositions (atomic names) and $\oplus$ represents exclusive or (xor). Our explicit goal is to develop a testing framework using temporal logics. Since testing sessions are sooner or later stopped and a result of the analysis is expected, our execution traces will be *finite*. More precisely, we regard a trace as a finite sequence of abstract states. In practice, these states are generated by events emitted by the program that we want to observe. If s is a state and a is a proposition, then $a(s)$ is true if and only if a holds in the state s; what it means for a proposition to "hold" in a state is intentionally left undefined, but it can essentially mean anything: a variable is larger than another, a lock is acquired, an array is sorted, etc. Finite traces will be the models of the two temporal logics defined below, but it is worth mentioning that they are regarded differently within the two logics: in future time LTL a finite trace is a sequence of future events, while in past time LTL it is a sequence of past events. However, they both interpret the other propositional operators as expected, that is

$$\begin{array}{lll} t \models \mathsf{true} & & \text{is always true,} \\ t \models \mathsf{false} & & \text{is always false,} \\ t \models \neg F & \text{iff} & \text{it is not the case that } t \models F, \\ t \models F_1 \; op \; F_2 & \text{iff} & t \models F_1 \text{ or/and/xor/implies/iff } t \models F_2, \\ & & \text{when} op \text{is} \vee / \wedge / \oplus / \rightarrow / \leftrightarrow . \end{array}$$

3.1. *Future time linear temporal logic*

Formulae in classical Linear Temporal Logic (LTL) can be built using the following operators:

$$\begin{array}{lll} F ::= & \mathsf{true} \mid \mathsf{false} \mid A \mid \neg F \mid F \; op \; F & \text{Propositional operators} \\ & \circ F \mid \diamond F \mid \Box F \mid F \, \mathcal{U}_s \, F \mid F \, \mathcal{U}_w \, F & \text{Future time operators} \end{array}$$

The propositional binary operators, *op*, are the ones above, and $\circ F$ should be read "next F", $\diamond F$ "eventually F", $\square F$ "always F", $F_1\ \mathcal{U}_s\ F_2$ "F_1 strong until F_2", and $F_1\ \mathcal{U}_w\ F_2$ "F_1 weak until F_2." An LTL standard model is a function $t : \mathcal{N}^+ \rightarrow 2^{\mathcal{P}}$ for some set of atomic propositions $\mathcal{P}$, i.e., an infinite trace over the alphabet $2^{\mathcal{P}}$, which maps each time point (a positive natural number) into the set of propositions that hold at that point. The propositional operators have their obvious meaning. $\circ F$ holds for a trace if F holds in the suffix trace starting in the next (the second) time point. The formula $\square F$ holds if F holds in all future time points, while $\diamond F$ holds if F holds in some future time point. The formula $F_1\ \mathcal{U}_s\ F_2$ holds if F_2 holds in some future time point, and until then F_1 holds. The formula $F_1\ \mathcal{U}_w\ F_2$ holds if either F_1 holds in all future time points, or otherwise, if F_2 holds in some future time point and until then F_1 holds. As an example illustrating the semantics, the formula $\square\,(F_1 \rightarrow \diamond F_2)$ is true if for any time point it holds that if F_1 is true then eventually F_2 is true. Another property is $\square\,(X \rightarrow \circ(Y\ \mathcal{U}_s\ Z))$, which states that whenever X holds then from the next state Y holds until strong eventually Z holds. It is standard to define a core LTL using only atomic propositions, the propositional operators $\neg$ (not) and $\wedge$ (and), and the temporal operators $\circ\_$ and $\_\ \mathcal{U}_s\ \_$, and then define all other propositional and temporal operators as derived constructs, such as $\diamond F := \mathsf{true}\ \mathcal{U}_s\ F$ and $\square F := \neg\diamond\neg F$.

Since we want to use future time LTL in a runtime monitoring setting, we need to formalize what it means for a finite trace to satisfy an LTL formula. The debatable issue is, of course, what happens at the end of the trace. One possibility is to consider that all the atomic propositions fail or succeed; however, this does not seem to be a good assumption because it may be the case that a proposition held throughout the trace while a violation will be reported at the end of monitoring. Driven by experiments, we found that a more reasonable assumption is to regard a finite trace as an infinite stationary trace in which the last event is repeated infinitely. If $t = s_1 s_2 \ldots s_n$ is a finite trace then we let $t^{(i)}$ denote the trace $s_i s_{i+1} \ldots s_n$ for each $1 \le i \le n$. With the intuitions above we can now define the semantics of finite trace future time LTL as follows:

$$
\begin{array}{lll}
t \models a & \text{iff} & a(s_1) \text{ holds,} \\
t \models \circ F & \text{iff} & t' \models F, \text{ where } t' = t^{(2)} \text{ if } n > 1 \text{ and } t' = t \text{ if } n = 1, \\
t \models \diamond F & \text{iff} & t^{(i)} \models F \text{ for some } 1 \le i \le n, \\
t \models \square F & \text{iff} & t^{(i)} \models F \text{ for all } 1 \le i \le n, \\
t \models F_1\ \mathcal{U}_s\ F_2 & \text{iff} & t^{(j)} \models F_2 \text{ for some } 1 \le j \le n \text{ and} \\
 & & t^{(i)} \models F_1 \text{ for all } 1 \le i < j, \\
t \models F_1\ \mathcal{U}_w\ F_2 & \text{iff} & t \models \square F_1 \text{ or } t \models F_1\ \mathcal{U}_s\ F_2.
\end{array}
$$

It is easy to see that if t is a trace of size 1 then $t \models \circ F$ or $t \models \diamond F$ or $t \models \square F$ if and only if $t \models F$, and that $t \models F_1\ \mathcal{U}_s\ F_2$ if and only if $t \models F_2$, and also that $t \models F_1\ \mathcal{U}_w\ F_2$ if and only if $t \models F_1$ or $t \models F_2$. It is worth noticing that finite trace LTL can behave quite differently from standard, infinite trace LTL. For example, there are formulae which are not valid in infinite trace LTL but are valid in finite trace LTL, such as $\diamond\,(\square\, a \vee \square\neg a)$, and there are formulae which are satisfiable in infinite trace LTL but not in finite trace LTL, such as the negation of the above. The formula above is satisfied by any finite trace because the last event/state in the trace either satisfies a or it doesn't.

3.2. *Past time linear temporal logic*

We next introduce basic past time LTL operators together with some operators that we found particularly useful for runtime monitoring, and which (except for the last) were introduced in [30], as well as their finite trace semantics. Syntactically, we allow the following formulae:

$$
\begin{array}{lll}
F ::= & \mathsf{true} \mid \mathsf{false} \mid A \mid \neg F \mid F\ op\ F & \text{Propositional operators} \\
 & \odot F \mid \diamond F \mid \boxdot F \mid F\ \mathcal{S}_s\ F \mid F\ \mathcal{S}_w\ F & \text{Standard past time operators} \\
 & \uparrow F \mid \downarrow F \mid [F, F)_s \mid [F, F)_w & \text{Monitoring operators}
\end{array}
$$

The propositional binary operators, *op*, are like before, and $\odot F$ should be read "previously F", $\diamond F$ "eventually in the past F", $\boxdot F$ "always in the past F", $F_1\ \mathcal{S}_s\ F_2$ "F_1 strong since F_2", $F_1\ \mathcal{S}_w\ F_2$ "F_1 weak since F_2", $\uparrow F$ "start F", $\downarrow F$ "end F", and $[F_1, F_2)$ "interval F_1, F_2" with a strong and a weak version.

If $t = s_1 s_2 \ldots s_n$ is a trace then we let $t_{(i)}$ denote the trace $s_1 s_2 \ldots s_i$ for each $1 \leq i \leq n$. Then the semantics of these operators can be given as follows:

$$
\begin{array}{lll}
t \models a & \text{iff} & a(s_n) \text{ holds,} \\
t \models \odot F & \text{iff} & t' \models F, \text{ where} \\
 & & t' = t_{(n-1)} \text{ if } n > 1 \text{ and } t' = t \text{ if } n = 1, \\
t \models \diamond F & \text{iff} & t_{(i)} \models F \text{ for some } 1 \leq i \leq n, \\
t \models \boxdot F & \text{iff} & t_{(i)} \models F \text{ for all } 1 \leq i \leq n, \\
t \models F_1\ \mathcal{S}_s\ F_2 & \text{iff} & t_{(j)} \models F_2 \text{ for some } 1 \leq j \leq n \text{ and} \\
 & & t_{(i)} \models F_1 \text{ for all } j < i \leq n, \\
t \models F_1\ \mathcal{S}_w\ F_2 & \text{iff} & t \models F_1\ \mathcal{S}_s\ F_2 \text{ or } t \models \boxdot F_1, \\
t \models \uparrow F & \text{iff} & t \models F \text{ and it is not the case that } t \models \odot F, \\
t \models \downarrow F & \text{iff} & t \models \odot F \text{ and it is not the case that } t \models F, \\
t \models [F_1, F_2)_s & \text{iff} & t_{(j)} \models F_1 \text{ for some } 1 \leq j \leq n \text{ and} \\
 & & t_{(i)} \not\models F_2 \text{ for all } j \leq i \leq n, \\
t \models [F_1, F_2)_w & \text{iff} & t \models [F_1, F_2)_s \text{ or } t \models \boxdot \neg F_2.
\end{array}
$$

Notice the special semantics of the operator "previously" on a trace of one state: $s \models \odot F$ iff $s \models F$. This is consistent with the view that a trace consisting of exactly one state s is considered like a *stationary* infinite trace containing only the state s. We adopted this view because of intuitions related to monitoring. One can start monitoring a process potentially at any moment, so the first state in the trace might be different from the initial state of the monitored process. We think that the "best guess" one can have w.r.t. the past of the monitored program is that it was stationary, in a perfectly dual manner to future time finite trace LTL. Alternatively, one could consider that $\odot F$ is false on a trace of one state for any atomic proposition F, but we find this semantics inconvenient because some atomic propositions may be related, such as, for example, a proposition "gate-up" and a proposition "gate-down"; at any moment, one is true if and only if the other is false, so a semantics allowing both to initially have the same truth value would not be satisfactory.

The non-standard operators $\uparrow$, $\downarrow$, $[\_, \_)_s$, and $[\_, \_)_w$ were inspired by work in runtime verification in [30]. These can be defined using the standard operators, but we found them often more intuitive and compact than the usual past time operators in specifying runtime requirements. $\uparrow F$ is true if and only if F *starts* to be true in the current state, $\downarrow F$ is true if and only if F *ends* to be true in the current state, and $[F_1, F_2)_s$ is true if and only if F_2 was never true since the last time F_1 was observed to be true, including the state when F_1 was true; the interval operator, like the "since" operator, has both a strong and a weak version. For example, if START and DOWN are propositions representing predicates on the state of a web server to be monitored, say for the last 24 hours, then $[\text{START}, \text{DOWN})_s$ is a property stating that the server *was* rebooted recently and since then it was not down, while $[\text{START}, \text{DOWN})_w$ says that the server was not unexpectedly down recently, meaning that it was either not down at all recently or it was rebooted and since then it was not down.

As shown later in the paper, one can generate very efficient monitors from past time LTL formulae, based on dynamic programming. What makes it so suitable for dynamic programming is its recursive nature: the satisfaction relation for a formula can be calculated along the execution trace looking only one step backwards:

$$
\begin{array}{lll}
t \models \Diamond F & \text{iff} & t \models F \text{ or } \big(n > 1 \text{ and } t_{(n-1)} \models \Diamond F\big),\\
t \models \boxdot F & \text{iff} & t \models F \text{ and } \big(n > 1 \text{ implies } t_{(n-1)} \models \boxdot F\big),\\
t \models F_1\ \mathcal{S}_s\ F_2 & \text{iff} & t \models F_2 \text{ or}\\
 & & \big(n > 1 \text{ and } t \models F_1 \text{ and } t_{(n-1)} \models F_1\ \mathcal{S}_s\ F_2\big),\\
t \models F_1\ \mathcal{S}_w\ F_2 & \text{iff} & t \models F_2 \text{ or}\\
 & & \big(t \models F_1 \text{ and } \big(n > 1 \text{ implies } t_{(n-1)} \models F_1\ \mathcal{S}_w\ F_2\big)\big),\\
t \models [F_1, F_2)_s & \text{iff} & t \not\models F_2 \text{ and}\\
 & & \big(t \models F_1 \text{ or } \big(n > 1 \text{ and } t_{(n-1)} \models [F_1, F_2)_s\big)\big),\\
t \models [F_1, F_2)_w & \text{iff} & t \not\models F_2 \text{ and}\\
 & & \big(t \models F_1 \text{ or } \big(n > 1 \text{ implies } t_{(n-1)} \models [F_1, F_2)_w\big)\big).
\end{array}
$$

There is a tendency among logicians to minimize the number of operators in a given logic. For example, it is known that two operators are sufficient in propositional calculus, and two more ("next" and "until") are needed for future time temporal logics. There are also various ways to minimize our past time logic defined above. More precisely, as claimed in [27], any combination of one operator in the set $\{\odot, \uparrow, \downarrow\}$ and another in the set $\{\mathcal{S}_s, \mathcal{S}_w, [)_s, [)_w\}$ suffices to define all the other past time operators. Two of these 12 combinations are known in the literature. Unlike in theoretical research, in practical monitoring of programs we want to have as many temporal operators as possible available and *not* to automatically translate them into a reduced kernel set. The reason is twofold. On the one hand, the more operators are available, the more succinct and natural the task of writing requirement specifications. On the other hand, as seen later in the paper, additional memory is needed for each temporal operator in the specification, so we want to keep the formulae short.

4. Monitoring requirements expressed in temporal logics

Logic based monitoring consists of checking execution events against a user-provided requirement specification written in some logic, typically an assertion logic with states as models, or a temporal logic with traces as models. JPAX currently provides the two linear temporal logics discussed above as built-in logics. Multiple logics can be used in parallel, so each property can be expressed in its most suitable language. JPAX allows the user to define such new logics in a flexible manner, either by using the Maude executable algebraic specification language or by implementing specialized algorithms that synthesize efficient monitors from logical formulae.

4.1. Rewriting based monitoring

Maude [7–9] is a modularized membership equational [34] and rewriting logic [33] specification and verification system whose operational engine is mainly based on a very efficient implementation of rewriting. A Maude module consists of sort and operator declarations, as well as equations relating terms over the operators and universally quantified variables; modules can be composed. It is often the case that equational and/or rewriting logics act like universal logics, in the sense that other logics, or more precisely their syntax and operational semantics, can be expressed and efficiently executed by rewriting, so we regard Maude as a good choice to develop and prototype with various monitoring logics. The Maude implementations of the current temporal logics are quite compact, so we include them below. They are based on a simple, general architecture to define new logics which we only describe informally. Maude's notation will be introduced "on the fly" as needed in examples.

4.1.1. Formulae and data structures. We have defined a generic module, called `FORMULA`, which defines the infrastructure for all the user-defined logics. Its Maude code will not be given due to space limitations, but the authors are happy to provide it on request. The module `FORMULA` includes some designated basic sorts, such as `Formula` for syntactic formulae, `FormulaDS` for formula data structures needed when more information than the formula itself should be stored for the next transition as in the case of past time LTL, `Atom` for atomic propositions, or state variables, which in the state denote boolean values, `AtomState` for assignments of boolean values to atoms, and `AtomState*` for such assignments together with *final* assignments, i.e., those that are followed by the end of a trace, often requiring a special evaluation procedure as described in the subsections on future time and past time LTL. A state `As` is made terminal by applying to it the unary operator `_* : AtomState -> AtomState*`. `Formula` is a subsort of `FormulaDS`, because there are logics in which no extra information but a modified formula needs to be carried over for the next iteration (such as future time LTL). The propositions that hold in a certain program state are generated from the executing instrumented program.

Perhaps the most important operator in `FORMULA` is `_{_} : FormulaDS AtomState -> FormulaDS`, which updates the formula data structure when an (abstract) state change occurs during the execution of the program. Notice the use of *mix-fix* notation for operator declaration, in which underscores represent places of arguments, their order being the one in the arity of the operator. On atomic propositions, say `A`, the module `FORMULA` defines the "update" operator as follows: `A{As*}` is true or false, depending on whether `As*` assigns true or false to the atom `A`, where `As*` is a terminal or not atom state (i.e., an assignment from atoms to boolean values). In the case of propositional calculus, this update operation basically evaluates propositions in the new state. For other logics it can be more complicated, depending on the trace semantics of those particular logics.

4.1.2. Propositional calculus. Propositional calculus should be included in any monitoring logic worth its salt. Therefore, we begin with the following module which is heavily used in JPAX. It implements an efficient rewriting procedure due to Hsiang [29] to decide validity of propositions, reducing any boolean expression to an exclusive disjunction (formally written _++_) of conjunctions (_/_):

```
fmod PROP-CALC is ex FORMULA.
*** Constructors ***
  op _/\_ : Formula Formula -> Formula [assoc comm].
  op _++_ : Formula Formula -> Formula [assoc comm].

  vars X Y Z : Formula . var As* : AtomState*.

  eq true /\ X = X.
  eq false /\ X = false.
  eq false ++ X = X.
  eq X ++ X = false.
  eq X /\ X = X.
  eq X /\ (Y ++ Z) = (X /\ Y) ++ (X /\ Z).


*** Derived operators ***
  op _\/_ : Formula Formula -> Formula.
  op _->_ : Formula Formula -> Formula.
  op _<->_: Formula Formula -> Formula.
  op !_ : Formula -> Formula.
  eq X \/ Y = (X /\ Y) ++ X ++ Y.
  eq ! X = true ++ X.
  eq X -> Y = true ++ X ++ (X /\ Y).
  eq X <-> Y = true ++ X ++ Y.
*** Semantics
  eq (X /\ Y){As*} = X{As*} /\ Y{As*}.
  eq (X ++ Y){As*} = X{As*} ++ Y{As*}
endfm
```

In Maude, operators are introduced after the `op` and `ops` (when more than one operator is introduced) symbols. Operators can be given attributes in square brackets, such as associativity and commutativity. Universally quantified variables used in equations are introduced after the `var` and `vars` symbols. Finally, equations are introduced after the `eq` symbol. The specification shows the flexible mix-fix notation of Maude, using underscores to stay for arguments, which allows us to define the syntax of a logic in the most natural way.

4.1.3. Future time linear temporal logic. Our implementation of the future time LTL presented in Section 3.1 simply consists of 8 equations, executed by Maude as rewrite rules whenever a new event/state is received. For simplicity we only present the strong until operator here. The weak until operator, which occurs more rarely in monitoring requirements, can be obtained by replacing the righthand-side of the last equation by `X{As *} \/ Y{As *}`:

```
fmod FT-LTL is ex PROP-CALC.
*** Syntax ***
  op []_ : Formula -> Formula.
  op <>_ : Formula -> Formula.
  op o_  : Formula -> Formula.
  op _U_ : Formula Formula -> Formula.

*** Semantics ***
  vars X Y : Formula.  var As : AtomState.


  eq ([] X){As}  = ([] X) /\ X{As}.
  eq (<> X){As}  = (<> X) \/ X{As}.
  eq (o X){As}   = X.
  eq (X U Y){As} = Y{As} \/ (X{As} /\ (X U Y)).

  eq ([] X){As *}  = X{As *}.
  eq (<> X){As *}  = X{As *}.
  eq (o X){As *}   = X{As *}.
  eq (X U Y){As *} = Y{As *}.
endfm
```

The four LTL operators are added to those of the propositional calculus using the symbols: `[]_` (always), `<>_` (eventually), `o_` (next), and `_U_` (until). The operational semantics of these operators is based on a formula transformation idea, in which monitoring requirements (formulae) are transformed when a new event is received, hereby consuming the event. These rules are similar in spirit to the "expansion rules" used in tableau and automata construction for LTL (see, e.g., Chapter 5 in [31]). The 8 equations, divided in two groups, refine the operator `_{_}` : `FormulaDS AtomState -> FormulaDS` provided by the module `FORMULA` described in Section 4.1.1. Note that in the future time LTL case, the formulae themselves are used as data structures, and that this is permissible because `Formula` is a subsort of `FormulaDS`. The operator `_{_}` tells how a formula is transformed by the occurrence of a state change. The interested reader can consult [25] for a formal correctness proof and analysis of this simple to implement rewriting algorithm. The underlying intuition can be elaborated as follows. Assume a formula `X` that we want to monitor on an execution trace of which the first state is `As`. Then the equation `X{As}` = `X'`, where `X'` is a formula resulting from applying the `_{_}` operator to `X` and `As`, carries the following intuition: in order for `X` to hold on the rest of the trace, given that the first state in the trace is `As`, then `X'` must hold on the trace following `As`. The first set of 4 rules describes this semantics assuming that the state `As` is not the last state in the trace, while the last four rules apply when `As` is the last in the trace. The term `As *` represents a state that is the last in the trace, and reflects the before mentioned intuition that a *finite* trace can be regarded as an *infinite* trace where the last state of the finite trace is repeated infinitely. As an example, consider the formula:

```
[](green -> (!red) U yellow)
```

representing a monitoring requirement of a traffic light controller, and a trace where the first state `As` makes the proposition `green` true and the others false. In this case, the formula:

```
[](green -> (!red) U yellow)As
```

reduces by rewriting to:

```
[](green -> (!red) U yellow) /\ (!red) U yellow
```

This reflects the fact that after the state change, `(!red) U yellow` now has to be true on the remaining trace, in addition to the original always-formula. A proof of correctness of this algorithm is given in [25], together with a very simple improvement based on *memoization*, which can increase its efficiency in practice by more than an order of magnitude. The effect of memoization (which essentially caches normal forms of terms so they will never be reduced again) is that of building a monitoring automaton on the fly, as the formulae (which become states in that automaton) are generated during the monitored execution trace. Despite its overall exponential worst-case complexity (more precisely $O(2^{2m^2})$, where m is the size of the LTL formula to be monitored, as shown in a more general setting in [37]), our rewriting based algorithm tends to be quite acceptable in practical situations. We couldn't notice any significant difference in global concrete experiments with JPAX between this simple 8 rule algorithm and an automata-based one in [15] that implements in 1,400 lines of Java code a Büchi automata inspired algorithm adapted to finite trace LTL (this is due to the highly efficient implementation of Maude).

Such a finite trace semantics for LTL used for program monitoring has, however, some characteristics that may seem unnatural. At the end of the execution trace, when the observed program terminates, the observer needs to take a decision regarding the validity of the checked properties. Let us consider now the formula `[](p -> <>q)`. If each `p` was followed by at least one `q` during the monitored execution, then, at some extent one could say that the formula was satisfied; although one should be aware that this is not a definite answer because the formula could have been very well violated in the future if the program hadn't been stopped. If `p` was true and it was not followed by a `q`, then one could say that the formula was violated, but it may have been very well satisfied if the program had been left to continue its execution. Furthermore, every `p` could have been followed by a `q` during the execution, only to be violated for the last `p`, in which case we would likely expect the program to be correct if we terminated it by force. There are of course LTL properties that give the user absolute confidence during the monitoring. For example, a violation of a safety property reflects a clear misbehavior of the monitored program.

The lesson that we learned from experiments with LTL monitoring is twofold. First, we learned that, unlike in model checking or theorem proving, future time LTL formulae and especially their violation or satisfaction must be viewed with extra information, such as for example statistics of how well a formula has "performed" along the execution trace, as first suggested in [24] and then done in [13]. Second, we developed a belief that future time propositional LTL may not be the most appropriate formalism for logic based monitoring; other logics, such as real time LTL, interval logics, past time LTL, or most likely undiscovered ones, could be of greater interest than pure future time LTL. We next describe an implementation of past time LTL in Maude, a perhaps more natural logic for runtime monitoring.

4.1.4. Past time linear temporal logic. Safety requirements can usually be more easily expressed using past time LTL formulae than using future time ones. More precisely, they can be represented as formulae `[] F`, where `F` is a past time LTL formula [31, 32]. These properties are very suitable for logic based monitoring because they only refer to the past, and hence their value is always either true or false in any state along the trace, and never *to-be-determined* as in future time LTL. However, the implementation of past time LTL is, surprisingly, slightly more tedious than the above implementation of future time LTL. In order to keep the specification short, we only include the standard past time operators "previous" and "strong since" below, the others being either defined similarly or just rewritten in terms of the standard operators. Our rewriting implementation appears similar in spirit to the one used in [30] (according to a private communication), which also uses a version of past time logic:

```
fmod PT-LTL is ex PROP-CALC.
*** Syntax ***
  op ~_  : Formula -> Formula.
  op _S_ : Formula Formula -> Formula.

*** Semantic Data structure ***
  op mkDS  : Formula AtomState -> FormulaDS.
  op atom  : Atom Bool -> FormulaDS.
  op and   : FormulaDS FormulaDS Bool -> FormulaDS.
  op xor   : FormulaDS FormulaDS Bool -> FormulaDS.
  op prev  : FormulaDS Bool -> FormulaDS.
  op since : FormulaDS FormulaDS Bool -> FormulaDS.

  var A : Atom.
  var As : AtomState.
  var B : Bool.
  vars X Y : Formula.
  vars D D' Dx Dx' Dy Dy' : FormulaDS.

  eq [atom(A,B)] = B.
  eq [and(Dx,Dy,B)] = B.
  eq [xor(Dx,Dy,B)] = B.
  eq [prev(D,B)] = B.
  eq [since(Dx,Dy,B)] = B.


  eq mkDS(true, As) = true.
  eq mkDS(false, As) = false.
  eq mkDS(A, As) = atom(A, (A{As} == true)).
```

```
  ceq mkDS(X /\ Y, As) = and(Dx, Dy, [Dx] and [Dy])
     if Dx := mkDS(X, As) /\ Dy := mkDS(Y, As).
  ceq mkDS(X ++ Y, As) = xor(Dx, Dy, [Dx] xor [Dy])
     if Dx := mkDS(X, As) /\ Dy := mkDS(Y, As).
  ceq mkDS(~ X, As) = prev(Dx, [Dx])
     if Dx := mkDS(X, As).
  ceq mkDS(X S Y, As) = since(Dx, Dy, [Dy])
     if Dx := mkDS(X, As) /\ Dy := mkDS(Y, As).

*** Semantics ***
   eq atom(A,B){As} = atom(A, (A{As} == true)).
  ceq and(Dx,Dy,B){As} =
      and(Dx',Dy',[Dx'] and [Dy'])
     if Dx' := Dx{As} /\ Dy' := Dy{As}.
  ceq xor(Dx,Dy,B){As} =
      xor(Dx',Dy',[Dx'] xor [Dy'])
     if Dx' := Dx{As} /\ Dy' := Dy{As}.
   eq prev(D,B){As} = prev(D{As},[D]).
  ceq since(Dx,Dy,B){As} =
      since(Dx',Dy',[Dy'] or B and [Dx'])
     if Dx' := Dx{As} /\ Dy' := Dy{As}.
endfm
```

The module first introduces the syntax of the logic and then the formula data structure needed for past time LTL and its semantics. The data structure consists of terms of sort `FormulaDS` and is needed to represent a formula properly during monitoring. This is in contrast to future time LTL, where a formula represented itself, and a transformation caused by a state transition was performed by transforming the formula into a new formula that had to hold on the rest of the trace. In past time LTL this technique does not apply. Instead, for each formula a special tree-like data structure is introduced, which keeps track of the boolean values of all subformulae of the formula in the latest considered state. These values are used to correctly evaluate the value of the entire formula when the next state is received. The operation `mkDS` creates the data structure representing a formula. The constructors of type `FormulaDS` correspond to the different kinds of past time LTL operators: `atom` (for atomic propositions), `and`, `xor`, `prev`, and `since`. Hence, for example, the formula `~ A` (previously `A`) for some atomic proposition `A` is represented by `prev(atom(A, true), false)` in case `A` is true in the current state but was false in the previous state. Hence the second boolean argument represents the current value of the formula, and is returned by the `[_]` operation. The `mkDS` operation that creates the initial data structures from formulae is defined when the first, or the initial, event/state is received.

Once the data structure is initialized, the operation `_{_}` : `FormulaDS AtomState -> FormulaDS` is iteratively used to modify the formula data structure on each subsequent state. The equations for operators `and`, `xor`, `prev` and `since` are defined using conditional equations (`ceq`). Conditions are provided after the `if` keyword. They can introduce new variables via built-in matching operators (`_:=_`). For example, if a formula data structure has the form `since(Dx, Dy, B)`, where `Dx` and `Dy` are other formula data structures and `B` is the current boolean value of the associated "since" formula, then, when receiving a new state `As`, we first update the child data structures into

`Dx'` and `Dy'`, and then update the current value of the formula as expected: it is true if and only if the value associated with `Dy'` is true (i.e., if its second argument holds now) or else both the value associated with `Dx'` is true (i.e., its first argument holds now) and `B` is true (i.e., the formula's value at the previous step was true). The binary operator _==_ is also built-in and takes two terms of any sort, reduces them to their normal forms, and then returns true if they are equal and false otherwise.

4.2. *Efficient observer generation*

Even though the rewriting based monitoring algorithms presented in the previous subsection perform quite well in practice, there can be situations in which one wants to minimize the monitoring overhead as much as possible. Additionally, despite their simplicity and elegance, the procedures above require an efficient AC rewriting engine (for propositional calculus simplifications) which may not be available or may not be desirable on some monitoring platforms, such as, for example, within an embedded system. In this subsection we present two efficient monitoring algorithms, one for future time and the other for past time LTL.

4.2.1. Future time LTL. In this section we overview an algorithm built on the ideas in Section 4.1.3, taking as input a future time LTL formula and generating a special finite state machine (FSM), called *binary transition tree finite state machine (BTT-FSM)*, that can then be used as an efficient monitor. We here only present it at a high level and put emphasis on examples. A BTT-FSM for the traffic light control formula `[](green -> !red U yellow)` discussed in Section 4.1.3 can be seen in figure 2 (figure 3 shows a more formal representation).

One should think of transitions using BTTs as naturally as possible; for example, if the BTT-FSM in figure 2 is in state 1 and a non-terminal event is received, then: first evaluate

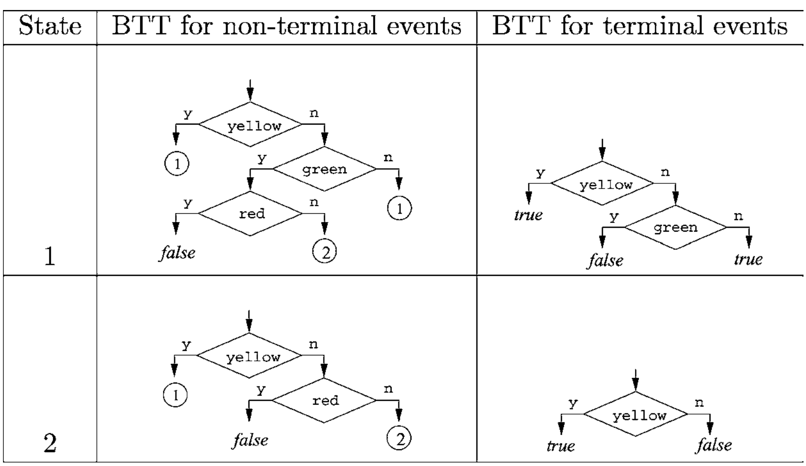

Figure 2. A BTT-FSM for the formula `[](green -> !red U yellow)`.

State	Non-terminal event	Terminal event
1	yellow ? 1 : green ? red ? false : 2 : 1	yellow ? true : green ? false : true
2	yellow ? 1 : red ? false : 2	yellow ? true : false

Figure 3. An optimal BTT-FSM for the formula `[](green -> !red U yellow)`.

the proposition `yellow`; if true then stay in state 1 else evaluate `green`; if false then stay in state 1 else evaluate `red`; if true then report "formula violated" else move to state 2. When receiving a terminal event, due to termination of monitoring, if the BTT-FSM is in state 1 then evaluate `yellow` and if true then return `true` else the opposite result of evaluating `green`. Only true/false messages are reported on terminal events, so the BTTs executed on terminal events are just Binary Decision Diagrams (BDDs) [6].

These FSMs can be either stored as data structures or generated as source code (case statements) which can further be compiled into actual monitors. BTT-FSMs only need to evaluate *at most* all the propositions in order to proceed to the next state when a new event is received, so the runtime overhead is actually linear with the number of distinct variables at worst. The size of our FSMs can become a problem when storage is a scarce resource, so we pay special attention to generating *optimal* FSMs. Interestingly, the number of propositions to be evaluated tends to decrease with the number of states, so the overall monitoring overhead is also reduced. The drawback of generating an optimal BTT-FSM statically, i.e., before monitoring, is the possibly double exponential time/space required at startup (compilation). Therefore, we recommend this algorithm only in situations where the LTL formulae to monitor are relatively small in size and the runtime overhead is necessary to be minimal.

Informally, our algorithm to generate minimal FSMs from LTL formulae uses the rewriting based algorithm presented in the previous section statically on all possible events, until the set of formulae to which the initial LTL formula can "evolve" stabilizes. More precisely, it builds a FSM whose states are formulae and whose transitions are "events", which are regarded as propositions on state boolean variables. Whenever a new potential state ψ, that is an LTL formula, is generated via a new event, that is a proposition on state variables p, from an existing state φ, ψ is semantically compared with all the previously generated states. If the ψ is not equivalent to any other existing state then it is added as a new state in the FSM. If found semantically equivalent to an already existing formula, say ψ', then it is not added to the state space, but the current transition from φ to ψ' is updated by taking its disjunction with p (if no transition from φ to ψ' exists then a transition p is added). The semantical comparison is done using a validity checker for finite trace LTL that we have developed especially for this purpose. All these techniques are described and analyzed in more detail in [36]. It is worth mentioning, though, that a minimal BTT-FSM can be generated *precisely* because we work with finite traces instead of infinite traces; the finite traces satisfying an LTL formula form a regular language, so they admit a minimal deterministic FSM.

Once the steps above terminate, the formulae φ, ψ, etc., encoding the states are not needed anymore, so we replace them by unique labels in order to reduce the amount of storage needed to encode the BTT-FSM. This algorithm can be relatively easily implemented in

Formula	State	Non-terminal event	Terminal event
$\Box\Diamond a$	1	1	a ? t : f
$\Diamond(\Box a \vee \Box\neg a)$	1	1	t
$\Box(a \rightarrow \Diamond b)$	1	a ? (b ? 1 : 2) : 1	a ? (b ? t : f) : t
	2	b ? 1 : 2	b ? t : f
$a\,\mathcal{U}\,(b\,\mathcal{U}\,c)$	1	c ? t : (a ? 1 : (b ? 2 : f))	c ? t : f
	2	c ? t : (b ? 2 : f)	c ? t : f
$a\,\mathcal{U}\,(b\,\mathcal{U}\,(c\,\mathcal{U}\,d))$	1	d ? t : a ? 1 : b ? 2 : c ? 3 : f	d ? t : f
	2	d ? t : b ? 2 : c ? 3 : f	d ? t : f
	3	d ? t : c ? 3 : f	d ? t : f
$((a\,\mathcal{U}\,b)\,\mathcal{U}\,c)\,\mathcal{U}\,d$	1	d ? t : c ? 1 : b ? 4 : a ? 5 : f	d ? t : f
	2	b ? c ? t : 7 : a ? c ? 6 : 2 : f	c ? b ? t : f : f
	3	b ? d ? t : c ? 1 : 4 : a ? d ? 6 : c ? 3 : 5 : f	d ? b ? t : f : f
	4	c ? d ? t : 1 : b ? d ? 7 : 4 : a ? d ? 2 : 5 : f	d ? c ? t : f : f
	5	b ? d ? c ? t : 7 : c ? 1 : 4 : a ? d ? c ? 6 : 2 : c ? 3 : 5 : f	d ? c ? b ? t : f : f : f
	6	b ? t : a ? 6 : f	b ? t : f
	7	c ? t : b ? 7 : a ? 2 : f	c ? t : f

Figure 4. Six BTT-FSMs generated in less than 15 seconds.

any programming language. We have, however, found Maude again a very elegant system, implementing this whole algorithm in about 200 lines of Maude code.

This BTT-FSM generation algorithm, despite its overall startup exponential time, can be very useful when formulae are relatively short. For the traffic light controller requirement formula discussed previously, `[](green -> (!red) U yellow)`, our algorithm generates in about 0.2 seconds the optimal BTT-FSM in figure 3 (also shown in figure 2 in flowchart notation).

For simplicity, the states true and false do not appear in figure 3. Notice that the proposition `red` does *not* need to be evaluated on terminal events and that `green` does not need to be evaluated in state 2. In this example, the colors are not supposed to exclude each other, that is, the traffic controller can potentially be both green and red.

The LTL formulae on which our algorithm has the worst performance are those containing many nested temporal operators (which are not frequently used in specifications anyway, because of the high risk of getting them wrong). For example, it takes our algorithm 1.3 seconds to generate the minimal 3-state (true and false states are not counted) BTT-FSM for the formula `a U (b U (c U d))` and 13.2 seconds to generate the 7-state minimal BTT-FSM for the formula `((a U b) U c) U d`. It never took our current implementation more than a few seconds to generate the BTT-FSM of any LTL formula of interest for our applications (i.e., non-artificial). Figure 4 shows the generated BTT-FSM of some artificial LTL formulae, taking together less than 15 seconds to be generated.

The generated BTT-FSMs are monitored most efficiently on RAM machines, due to the fact that case statements are usually implemented via jumps in memory. Monitoring

BTT-FSMs using rewriting does not seem appropriate because it would require linear time (as a function of the number of states) to extract the BTT associated to a state in a BTT-FSM. However, we believe that the algorithm presented in Section 4.1.3 is satisfactory in practice if one is willing to use a rewriting engine for monitoring.

4.2.2. Past time LTL. We next focus on generating efficient monitors from formulae in past time LTL. The generated monitoring algorithm tests whether the formula is satisfied by a finite trace of events given as input and runs in linear time and space in terms of both the formula and the size of trace. We only show how the generated monitoring algorithm looks for a concrete past time formula example, referring the interested reader to [27] (a detailed presentation on monitoring past time LTL formulae will appear soon elsewhere).

We think that the next example is practically sufficient for the reader to foresee our general algorithm in [27]. Let $\uparrow p \to [q, \downarrow (r \vee s))_s$ be the past time LTL formula that we want to generate code for. The formula states: "whenever p becomes true, then q has been true in the past, and since then we have not yet seen the end of r or s." The code translation depends on an enumeration of the subformulae of the formula that satisfies the *enumeration invariant*: any formula has an enumeration number smaller than the numbers of all its subformulae. Let $\varphi_0, \varphi_1, \ldots, \varphi_8$ be such an enumeration:

$$\begin{aligned}
\varphi_0 &= \uparrow p \to [q, \downarrow (r \vee s))_s,\\
\varphi_1 &= \uparrow p,\\
\varphi_2 &= p,\\
\varphi_3 &= [q, \downarrow (r \vee s))_s,\\
\varphi_4 &= q,\\
\varphi_5 &= \downarrow (r \vee s),\\
\varphi_6 &= r \vee s,\\
\varphi_7 &= r,\\
\varphi_8 &= s.
\end{aligned}$$

Note that the formulae have here been enumerated in a post-order fashion. One could have chosen a breadth-first order, or any other enumeration, as long as the enumeration invariant is true.

The input to the generated program will be a finite trace $t = e_1 e_2 \ldots e_n$ of n events. The generated program will maintain a state via a function $update\colon State \times Event \to State$, which updates the state with a given event. Our generated algorithms are dynamic programming algorithms based on the recursive nature of the semantics of past time LTL as shown in Section 3.2. In order to illustrate the dynamic programming aspect of the solution, one can imagine recursively defining a matrix $s[1..n, 0..8]$ of boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_{(i)} \models \varphi_j$. This would be the standard way of regarding the above satisfaction problem as a dynamic programming problem. An important observation is, however, that, like in many other dynamic programming algorithms, one doesn't have to store all the table $s[1..n, 0..8]$, which would be quite large in practice; in this case, one needs only $s[i, 0..8]$ and $s[i-1, 0..8]$, which we'll write $now[0..8]$ and $pre[0..8]$ from now on, respectively. It is now only a relatively simple exercise to write up the following algorithm for checking the above formula on a finite trace:

```
State state ← { };
bit pre[0..8]; bit now[0..8];
INPUT: trace t = e1 e2 ... en;
/* Initialization of state and pre */
state ← update(state, e1);
pre[8] ← s(state);
pre[7] ← r(state);
pre[6] ← pre[7] or pre[8];
pre[5] ← false;
pre[4] ← q(state);
pre[3] ← pre[4] and not pre[5];
pre[2] ← p(state);
pre[1] ← false;
pre[0] ← not pre[1] or pre[3];
/* Event interpretation loop */
for i = 2 to n do {
    state ← update(state, ei);
    now[8] ← s(state);
    now[7] ← r(state);
    now[6] ← now[7] or now[8];
    now[5] ← not now[6] and pre[6];
    now[4] ← q(state);
    now[3] ← (pre[3] or now[4]) and not now[5];
    now[2] ← p(state);
    now[1] ← now[2] and not pre[2];
    now[0] ← not now[1] or now[3];
    if now[0] = 0 then output(``property violated'');
    pre ← now;
};
```

In the following we explain the generated program.

Declarations. Initially a state is declared. This will be updated as the input event list is processed. Next, the two arrays *pre* and *now* are declared. The *pre* array will contain values of all subformulae in the previous state, while *now* will contain the value of all subformulae in the current state. The trace of events is then input. Such an event list can be read from a file generated from a program execution, or alternatively the events can be input on-the-fly one by one when generated, without storing them in a file first.

Initialization. The initialization phase consists of initializing the *state* variable and the *pre* array. The first event e_1 of the event list is used to initialize the *state* variable. The *pre* array is initialized by evaluating all subformulae bottom up, starting with highest formula numbers, and assigning these values to the corresponding elements of the *pre* array; hence, for any $i \in \{0 \dots 8\}$ $pre[i]$ is assigned the initial value of formula φ_i. The *pre* array is initialized in such a way as to maintain the view that the initial state is supposed stationary before monitoring is started. This in particular means that $\uparrow p$ is false, as well as is $\downarrow (r \vee s)$, since there is no change in state (indices 1 and 5). The interval operator has the obvious initial interpretation: the first argument must be true and the second false for the formula to be true (index 3). Propositions are true if they hold in the initial state (indices 2, 4, 7 and 8), and boolean operators are interpreted the standard way (indices 0, 6).

Event Loop. The main evaluation loop goes through the event trace, starting from the second event. For each such event, the state is updated, followed by assignments to the *now* array in a bottom-up fashion similar to the initialization of the *pre* array: the array elements are assigned values from higher index values to lower index values, corresponding to the values of the corresponding subformulae. Propositional boolean operators are interpreted the standard way (indices 0 and 6). The formula $\uparrow p$ is true if p is true now and not true in the previous state (index 1). Similarly with the formula $\downarrow (r \vee s)$ (index 5). The formula $[q, \downarrow (r \vee s))_s$ is true if either the formula was true in the previous state, or q is true in the current state, and in addition $\downarrow (r \vee s)$ is not true in the current state (index 3). At the end of the loop an error message is issued if $now[0]$, the value of the whole formula, has the value 0 in the current state. Finally, the entire *now* array is copied into *pre*.

Given a past time LTL formula, the analysis of this algorithm is straightforward. Its time complexity is $\Theta(n)$ where n is the length of the input trace, the constant being given by the size of the formula. The memory required is constant, since the length of the two arrays is the size of the formula. However, if one also includes the size of the formula, say m, into the analysis; then the time complexity is obviously $\Theta(n \cdot m)$ while the memory required is $2 \cdot (m + 1)$ bits. It is hard to find algorithms running faster than the above in practical situations, though some slight optimizations can be imagined as shown below.

Even though a smart compiler can in principle generate good machine code from the code above, it is still worth exploring ways to synthesize directly optimized code especially because there are some attributes that are specific to the runtime observer which a compiler cannot take into consideration. A first observation is that not all the bits in *pre* are needed, but only those which are used at the next iteration, namely 2, 3, and 6. Therefore, only a bit per temporal operator is needed, thereby reducing significantly the memory required by the generated algorithm. Then the body of the generated "for" loop becomes after (blind) substitution (we don't consider the initialization code here):

$state \leftarrow update(state, e_i)$
$now[3] \leftarrow r(state)\ or\ s(state)$
$now[2] \leftarrow (pre[2]\ or\ q(state))\ and\ not\ (not\ now[3]\ and\ pre[3])$
$now[1] \leftarrow p(state)$
if $((not\ (now[1]\ and\ not\ pre[1])\ or\ now[2]) = 0)$
    then output(``property violated'');

which can be further optimized by boolean simplifications:

$state \leftarrow update(state, e_j)$
$now[3] \leftarrow r(state)\ or\ s(state)$

$now[2] \leftarrow (pre[2]\ or\ q(state))\ and\ (now[3]\ or\ not\ pre[3])$
$now[1] \leftarrow p(state)$
if $(now[1]\ and\ not\ pre[1]\ and\ not\ now[2])$
  then output(``‘property violated’’``);

The most expensive part of the code above is clearly the function calls, namely $p(state)$, $q(state)$, $r(state)$, and $s(state)$. Depending upon the runtime requirements, the execution time of these functions may vary significantly. However, since one of the major concerns of monitoring is to affect the normal execution of the monitored program as little as possible, especially in online monitoring, one would of course want to evaluate the propositions on states only if really needed, or rather to evaluate only those that, probabilistically, add a minimum cost. Since we don't want to count on an optimizing compiler, we prefer to store the boolean formula as some kind of binary decision diagram. We have implemented a procedure in Maude, on top of a propositional calculus module, which generates all correct $(\_?\_ : \_)$-expressions for φ, admittedly a potentially exponential number in the number of distinct atomic propositions in φ, and then chooses the shortest in size. Applied on the code above, it yields:

$state \leftarrow update(state, e_j)$
$now[3] \leftarrow r(state)\ ?\ 1\ :\ s(state)$
$now[2] \leftarrow pre[3]\ ?\ pre[2]\ ?\ now[3]\ :$
    $q(state)\ :\ pre[2]\ ?\ 1\ :\ q(state)$
$now[1] \leftarrow p(state)$
if $(pre[1]\ ?\ 0\ :\ now[2]\ ?\ 0\ :\ now[1])$
  then output(``‘property violated’’``);

We would like to extend our procedure to take the evaluation costs of propositions, or rather: predicates, into consideration. These costs can either be provided by the user of the system or be calculated automatically by a static analysis of predicates' code, or even be estimated by executing the predicates on a sample of states. However, based on our examples so far, we conjecture that, given a boolean formula φ in which all the atomic propositions have the same cost, the probabilistically runtime optimal $(\_?\_ : \_)$-expression implementing φ is *exactly* the one which is smallest in size.

A further optimization would be to generate directly machine code instead of using a compiler. Then the arrays of bits *now* and *pre* can be stored in two registers, which would be all the memory needed. Since all the operations executed are bit operations, the generated code is expected to be very fast. One could even imagine hardware implementations of past time monitors, using the same ideas, in order to enforce safety requirements on physical devices.

5. Experiments

The JPaX system has been applied to several case studies, amongst them a planetary rover executive (K9), a spacecraft fault protection system (FP), and a spacecraft attitude control system (ACS). JPaX's temporal logic monitoring capability was used to analyze K9 as well as FP, while JPaX's concurrency analysis was used to analyze K9 as well as ACS.

The most interesting application of these with respect to temporal logic monitoring is the planetary rover K9, written in 35,000 lines of C++. K9 takes as input a plan, which states what actions to execute and when. The objective of K9 is to execute the plan correctly. The plan can for example be created by mission engineers on ground, and up-linked to the rover, or it can be generated automatically on board by a different software module. Testing K9 using temporal logic consists of generating a test-suite, consisting of a set of test-cases, each consisting of an input plan, and a set of temporal logic formulae that the execution of this plan must satisfy. In an initial experiment, a down-scaled Java version (approximately 7,300 lines of code) of K9 was analyzed [5]. The purpose was to compare different verification tools. The study involved several test groups, each applying a different tool to the same system. The techniques studied included temporal logic monitoring, deadlock and data race trace analysis, model checking, static analysis, and traditional testing. The reader is referred to [5] for a presentation of the results of this comparison. In the case of temporal logic monitoring, test-suites were written by hand. Temporal logic turned out to catch errors that were missed by just visually inspecting the output from running K9. However, manually writing a set of temporal formulae for each test-case turned out to be time consuming.

We therefore decided, based on the above experiment, to create a fully automated framework for testing the K9 rover. The framework, named X9 [1], automatically generates plans using model checking of a (non-deterministic) grammar of plans. Each generated plan is an "error trace" generated by the model checker, when it executes an `assert(false)` statement that occurs "at the end of" the grammar. For each plan a set of temporal properties is furthermore automatically generated that the execution of that plan must satisfy. The program is hand-instrumented in a few places, where actions start executing, and where they terminate, either successfully, or by failure. The instrumentation overhead on the running program is small, and not noticeable. This demonstrates how the observer part of JPaX can be used independently of the instrumentation part, and hence be applied to programs in other languages than Java, in this case C++.

When X9 is applied to K9, several hundred test-cases can be generated within a few seconds, each consisting of an input plan and a set of temporal properties. Each generated plan is subsequently executed, and the extracted execution trace is checked against the temporal formulae for that plan. Results are displayed on a web-site by displaying for each failed test-case links to the plan, the execution trace, and the temporal properties violated. Checking a resulting execution trace against its temporal formulae typically takes a couple of seconds, where most of this time is spent on starting the oracle engine (Maude). The checking of the trace itself takes only few milliseconds. To test the effectiveness of the approach, the K9 code was seeded with errors. X9 detected the errors automatically, highlighting the plans that caused the properties to be violated, and the violating execution traces. Beyond automatically generating and executing the test-cases, the method showed the importance of making it unnecessary to manually analyze thousands of lines of output.

K9 (the C++ version as well as the Java version) and ACS (written in 1850 lines of Java) were both analyzed using the concurrency algorithms in JPaX. JPaX found an unknown deadlock in the C++ version of K9 and two unknown data races in ACS. Furthermore, JPaX found all seeded deadlocks (cyclic lock acquisitions) and data race errors in the Java version of K9 and in ACS. The K9 concurrency analysis results are described in [5]. Further case studies with high-level data race analysis are described in [2].

6. Conclusions

We have presented work done in context of the JPAX runtime verification tool. JPAX provides an integrated environment for monitoring the execution of Java programs. A program is instrumented to generate an execution trace when run, which can then be examined by various specialized algorithms that we have described. Amongst these are algorithms for testing the trace against future time and past time temporal logic formulae. We have presented how such logics can be formulated in a rewriting logic, and how the Maude rewriting system can be used to monitor the validity of the execution trace against such formulae. It has also been shown how observer automata and algorithms can be generated from future time and past time temporal logic formulae, to achieve even more efficient observer algorithms. An interesting observation is, however, that the implementation of these logics in the Maude rewriting system resulted in very small code, and still compared very well with the automata/algorithm solutions in efficiency. Using Maude for defining new observer logics seems to be an excellent prototyping approach at least, and even seems fast enough for practical monitoring. Future work includes studying more powerful logics, including real-time information and data. Concerning concurrency analysis, we are currently expanding the kinds of errors that can be found by simple trace analysis. Runtime verification can be used during testing or after deployment, during operation of the software. An important and non-trivial research topic is how to correct the behavior of a program on-the-fly when properties are violated.

Note

1. The specification is in reality represented in two files, an instrumentation specification (the first two lines), with slightly less attractive syntax, and a verification specification (the last line), but for clarity we present it here as one file, with idealized syntax.

References

1. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser, "Experiments with test case generation and runtime analysis," in E. Börger, A. Gargantini, and E. Riccobene (Eds.), *Abstract State Machines 2003*, LNCS 2589, Taormina, Italy. Springer, 2003a, pp. 87–107. Invited paper.
2. C. Artho, K. Havelund, and A. Biere, "High-level data races," in *VVEIS'03, The First International Workshop on Verification and Validation of Enterprise Information Systems*, Angers, France, 2003b.
3. T. Ball, A. Podelski, and S. Rajamani, "Boolean and cartesian abstractions for model checking C programs," in *Proceedings of TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, Genova, Italy, 2001.
4. S. Bensalem and K. Havelund, "Deadlock analysis of multi-threaded Java programs." Submitted for publication, 2003.
5. G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser, "Experimental evaluation of verification and validation tools on martian rover software," in *Proceedings of Software Model Checking Workshop*. Carnegie Mellon Software Engineering Institute, 2003.

6. R.E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 677–691, 1986.
7. M. Clavel, F.J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada, "Maude: Specification and programming in rewriting logic," Maude System documentation at `http://maude.csl.sri.com/papers`, 1999a.
8. M. Clavel, F.J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada, "The Maude System," in *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, Vol. 1631 of *LNCS*. Springer-Verlag. Trento, Italy, pp. 240–243, 1999b, System description.
9. M. Clavel, F.J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada, "A Maude tutorial," Manuscript at `http://maude.csl.sri.com/papers`, 2000.
10. S. Cohen, "Jtrek." Compaq, `http://www.compaq.com/java/download/jtrek`, 2000.
11. J. Corbett, M.B. Dwyer, J. Hatcliff, C.S. Pasareanu, Robby, S. Laubach, and H. Zheng, "Bandera: Extracting finite-state models from Java source code," in *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, Limerich, Ireland, 2000.
12. D. Drusinsky, "The temporal rover and the ATG rover," in *SPIN Model Checking and Software Verification*, Vol. 1885 of *LNCS*. Springer, pp. 323–330, 2000.
13. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma, "Collecting statistics over runtime executions," in K. Havelund and G. Roşu (Eds.), *Proceedings of Runtime Verification (RV'02)*, Vol. 70, No. 4. of Electronic Notes in Theoretical Computer Science, Elsevier Science, 2002.
14. B. Finkbeiner and H. Sipma, "Checking finite traces using alternating automata," in K. Havelund and G. Roşu (Eds.), *Proceedings of Runtime Verification (RV'01)*, Vol. 55, No. 2, of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001.
15. D. Giannakopoulou and K. Havelund "Automata-based verification of temporal properties on running programs," in *Proceedings, International Conference on Automated Software Engineering (ASE'01)*. Institute of Electrical and Electronics Engineers, Coronado Island, California, 2001, pp. 412–416.
16. P. Godefroid, "Model checking for programming languages using veriSoft," in *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, 1997, pp. 174–186.
17. J. Harrow, "Runtime checking of multithreaded applications with visual threads," in *SPIN Model Checking and Software Verification*, Vol. 1885 of LNCS. Springer, 2000, pp. 331–342.
18. K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *SPIN Model Checking and Software Verification*, Vol. 1885 of LNCS. Springer, 2000, pp. 245–264.
19. K. Havelund, S. Johnson, and G. Roşu, "Specification and error pattern based program monitoring," in *Proceedings of the European Space Agency workshop on On-Board Autonomy*. Noordwijk, The Netherlands, 2001.
20. K. Havelund and T. Pressburger, "Model checking Java programs using Java pathfinder." *International Journal on Software Tools for Technology Transfer* Vol. 2, No. 4, pp. 366–381. Special issue of STTT containing selected submissions to the *4th SPIN Workshop*, Paris, France, 1998, 2000.
21. K. Havelund and G. Roşu, "Testing linear temporal logic formulae on finite execution traces." RIACS Technical report, 2000 `http://ase.arc.nasa.gov/pax`.
22. K. Havelund and G. Roşu (Eds.), *1st CAV Workshop on Runtime Verification (RV'01)*, Vol. 55, No. 2 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001a.
23. K. Havelund and G. Roşu, "Java pathexplorer—A runtime verification tool," in *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS'01)*. Montreal, Canada, 2001b.
24. K. Havelund and G. Roşu, "Monitoring Java programs with Java pathExplorer," in K. Havelund and G. Roşu (Eds.), *Proceedings of Runtime Verification (RV'01)*, Vol. 55 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001c.
25. K. Havelund and G. Roşu, "Monitoring programs using rewriting," in *Proceedings, International Conference on Automated Software Engineering (ASE'01)*. Institute of Electrical and Electronics Engineers, Coronado Island, California, 2001d, pp. 135–143.
26. K. Havelund and G. Roşu (Eds.), *2nd CAV Workshop on Runtime Verification (RV'02)*, Vol. 70, No. 4 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2002.

27. K. Havelund and G. Roşu, "Synthesizing monitors for safety properties," in *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, Vol. 2280 of Lecture Notes in Computer Science. Springer, 2002b, pp. 342–356. EASST best paper award at ETAPS'02.
28. G.J. Holzmann and M.H. Smith, "A practical method for verifying event-driven software," in *Proceedings of ICSE'99, International Conference on Software Engineering*. Los Angeles, California, USA, IEEE/ACM, 1999.
29. J. Hsiang, "Refutational theorem proving using term rewriting systems," Ph.D. thesis, University of Illinois at Champaign-Urbana, 1981.
30. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
31. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer, New York, 1992.
32. Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
33. J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, pp. 73–155, 1992.
34. J. Meseguer, "Membership algebra as a logical framework for equational specification," in *Proceedings, WADT'97*, Vol. 1376 of LNCS. Springer, 1998, pp. 18–61.
35. G. Roşu and K. Havelund, "Synthesizing dynamic programming algorithms from linear temporal logic formulae," RIACS Technical Report, 2001. `http://ase.arc.nasa.gov/pax`.
36. G. Roşu and K. Havelund, "Rewriting-based techniques for runtime verification." To appear in the *Journal of Automated Engineering*. Kluwer Academic Publishers, 2003.
37. G. Roşu, and M. Viswanathan, "Testing extended regular language membership incrementally by rewriting," in *Rewriting Techniques and Applications (RTA'03)*. Springer-Verlag, 2003.
38. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, Vol. 15, No. 4, pp. 391–411, 1997.
39. Sitraka, "JProbe." `http://java.quest.com/jprobe/threadalyzer.shtml`, 2001.
40. S.D. Stoller, "Model-checking multi-threaded distributed Java programs," in *SPIN Model Checking and Software Verification*, Vol. 1885 of *LNCS*. Springer, 2000, pp. 224–244.
41. W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, 2000.
42. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, Vol. 10, pp. 203–232, 2003.