J. of STTT, Volume 6(2), pp 158-173. 2004

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Efficient Monitoring of Safety Properties

Klaus Havelund[1], **Grigore Roşu**[2⋆]

[1] Kestrel Technology NASA Ames Research Center, USA. e-mail: havelund@email.arc.nasa.gov
[2] Department of Computer Science, University of Illinois at Urbana-Champaign, USA. e-mail: grosu@cs.uiuc.edu

Received: date / Revised version: date

Abstract. The problem of testing whether a finite execution trace of events generated by an executing program violates a linear temporal logic (LTL) formula occurs naturally in runtime analysis of software. Two efficient algorithms for this problem are presented in this paper, both for checking safety formulae of the form "always P", where P is a past time LTL formula. The first algorithm is implemented by rewriting and the second synthesizes efficient code from formulae. Further optimizations of the second algorithm are suggested, reducing space and time consumption. Special operators suitable for writing succinct specifications are discussed and shown equivalent to the standard past time operators. This work is part of NASA's PathExplorer project, the objective of which is to construct a flexible framework for efficient monitoring and analysis of program executions.

1 Introduction

The work presented in this paper is part of a project at NASA Ames Research Center, called PathExplorer [19, 18, 14, 17, 34], that aims at developing a practical testing environment for NASA software developers. The basic idea of the project is to analyze the execution trace of a running program to detect errors. The errors that we are considering at this stage are multi-threading errors such as deadlocks and data races, and non-conformance with linear temporal logic specifications, which is the main focus of this paper.

Linear Temporal Logic (LTL) [33, 27, 28] is a logic for specifying properties of reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being a typical example. LTL has been mainly used to specify properties of concurrent and interactive down-scaled models of real systems, so that fully formal correctness proofs could subsequently be carried out, for example using theorem provers or model checkers (see for example [21, 20, 15]). However, formal proof techniques are usually not scalable to real sized systems without a substantial effort to abstract the system more or less manually to a model which can be analyzed. Model checking of programs has received an increased attention from the formal methods community within the last couple of years, and several systems have emerged that can directly model check source code, such as Java and C [16, 37, 9, 22, 8, 3, 32]. Stateless model checkers [12, 36] try to avoid the abstraction process by not storing states. Although these systems provide high confidence, they scale less well because most of their internal algorithms are exponential or worse.

Testing scales well, and is by far the most used technique in practice to validate software systems. The merge of testing and temporal logic specification is an attempt to achieve the benefits of both approaches, while avoiding some of the pitfalls of ad hoc testing and the complexity of full-blown theorem proving and model checking. Of course, there is a price to pay in order to obtain a scalable technique: the loss of coverage. The suggested framework can only be used to examine single execution traces, and can therefore not be used to prove a system correct. Our work is based on the belief that software engineers are willing to trade coverage for scalability, so our goal is to provide tools that are completely automatic, implement very efficient algorithms and find *many* errors in programs. A longer term goal is to explore the use of conformance with a formal specification to achieve fault tolerance. The idea is that the failure may trigger a recovery action in the monitored program.

The idea of using LTL in program testing is not new. It has already been pursued in commercial tools such as

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

J. of STTT, Volume 6(2), pp 158-173. 2004

Temporal Rover (TR) [10], which stimulated our work us in a major way. In TR, future and past time LTL properties are stated as formal comments within the program at chosen program points, like assertions. These formal comments are then, by a pre-processor, translated into code, which is inserted at the position of the comments, and executed whenever reached during program execution[1]. The MaC tool [26] is another example of a runtime monitoring tool that has inspired this work. Here Java bytecode is automatically instrumented to generate events of interest during the execution. Of special interest is the temporal logic used in MaC, which can be classified as a past time interval logic convenient for expressing monitoring properties in a succinct way. A theoretical contribution in this paper is Theorem 1, which shows that the MaC temporal logic, together with 10 others, is equivalent to the standard past time temporal logic. The path exploration tool described in [13] uses a future time temporal logic formula to guide the execution of a program for debugging purposes. Hence, the role of a temporal logic formula is turned around from monitoring a trace to generation of a trace.

Past time LTL has been shown to have the same expressiveness as future time LTL [11]. However, past time LTL is exponentially more succinct than future time LTL [29]. For example, a property like *"every response should be preceded by a request"* can be easily stated in past time logic (reflecting directly the previous sentence), but the corresponding future time representation becomes *"it's not the case that (there is no request until (there is a response and no request))"*. Hence, past time LTL is more convenient for specifying certain properties, and is the focus of this paper.

We present two efficient monitoring algorithms for checking safety formulae of the form "always P", where P is a past time LTL formula, one based on formula rewriting and one based on synthesizing efficient monitoring code from a formula. The rewriting-based algorithm illustrates how rewriting can be used to easily and elegantly define new logics for monitoring. This may be of interest when experimenting with logics, or if logics are domain specific and change with the application, or if one simply wants a small and elegant implementation. The synthesis-based algorithm, on the other hand, generates a very effective monitor for the particular past time logic, and focuses on efficiency. It is also better suited for generating code that can be inserted in the monitored program, in contrast to the rewriting approach, where a rewriting engine must be called by an external call.

The first algorithm is implemented by rewriting using Maude [5–7], an executable specification language whose main operational engine is based on term rewriting. Since flexibility with respect to defining/modifying monitoring logics is a very important factor at this stage in the development of PathExplorer, we have actually developed a general framework using Maude which allows one to easily and effectively define new logics for runtime analysis and to monitor execution traces against formulae in these logics. The rewriting algorithm presented in this paper instantiate that framework to our logic of interest, past time LTL. The second algorithm presented in this paper is designed to be as efficient and specialized as possible, thus adding the minimum possible amount of runtime overhead. It essentially synthesizes a special purpose, efficient monitoring code from formulae, which is further compiled into an executable monitor. Further optimizations of the second algorithm are suggested, making each monitoring step typically run in time lower than the size of the monitored formula. Both algorithms are based on the fact that the semantics of past time LTL can be defined recursively in such as way that one only needs to look one step, or event, backwards in order to compute the new truth value of a formula and of its subformulae, thus allowing one to process and then discard the events as they are received from the instrumented program. Several special operators suitable for writing succinct monitoring safety specifications are introduced and shown semantically equivalent to the standard past time operators.

Section 2 gives a short description of the PathExplorer architecture, putting the presented work in context. Section 3 recalls past time LTL and introduces several monitoring operators together with their semantics, then discusses several past time logics and finally shows their equivalences. Section 4 first presents our rewriting-based framework for defining and executing new monitoring logics, and then shows how past time LTL fits into this framework. Section 5 finally explains our monitor-synthesis algorithm, together with optimizations and two ways to implement it. Section 6 concludes the paper.

2 The PathExplorer Architecture

PathExplorer, PaX, is a flexible environment for monitoring and analyzing program executions. A program (or a set of programs) to be monitored, is supposed to be instrumented to emit execution events to an observer, which then examines the events and checks whether they satisfy certain user-given constraints. We first give an overview of the *observer* that monitors the event stream. Then we discuss how a program is instrumented for monitoring of temporal logic properties. The instrumentation presented is specialized to Java, but the principles carry over to any programming language.

2.1 The Observer

The constraints to be monitored can be of different kinds and defined in different languages. Each kind of constraint is represented by a module. Such a constraint

[1] The implementation details of TR are not public.

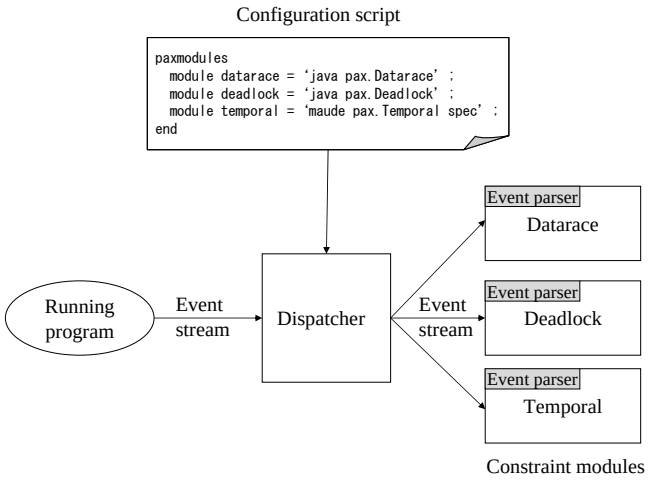

Fig. 1. Overview of the PaX observer.

module in principle implements a particular logic or program analysis algorithm. Currently there are modules for checking deadlock potentials, data race potentials, and for checking temporal logic formulae in different logics. Amongst the latter, several modules have been implemented for checking future time temporal logic, and the work presented in this paper is the basis for a module for checking past time logic formulae. In general, the user can program new constraint modules and in this manner extend PaX in an easy way.

The system is defined in a component-based way, based on a dataflow view, where components are put together using a "pipeline" operator, see Figure 1. The dataflow between any two components is a stream of events in simple text format, without any a-priori assumptions about the format of the events; the receiving component just ignores events it cannot recognize. This simplifies composition and allows for components to be written in different languages and in particular to define observers of arbitrary systems, programmed in a variety of programming languages. This latter fact is important at NASA since several systems are written in a mixture of C, C++ and Java.

The central component of the PaX system is a so-called *dispatcher*. The dispatcher receives events from the executing program or system and then retransmits the event stream to each of the constraint modules. Each module is running in its own process with one input pipe, only dealing with events that are relevant to the module. For this purpose each module is equipped with an event parser. The dispatcher takes as input a configuration script, which specifies a list of commands - a command for each module that starts the module in a process. The dispatcher may read its input event stream from a file, or alternatively from a socket, to which the instrumented running program must write the event stream. In the latter case, monitoring can happen on-the-fly as the event stream is produced, and potentially on a different computer than the observed system.

2.2 Code Instrumentation

The program or system to be observed must be instrumented to emit execution events to the dispatcher (writing them to a file or to a socket as discussed above). We have currently implemented an automated instrumentation package for Java bytecode using the Java bytecode engineering tool JTrek [25]. The instrumentation package together with PathExplorer is called Java PathExplorer (JPaX). Given information about what kind of events to be emitted, the instrumentation package instruments the bytecode by inserting extra code for emitting events. For deadlock analysis, for example, events are generated that inform about lock acquisitions and releases. For temporal logic monitoring, one specifies the variables to be observed, and what predicates over these variables one wants to refer to in the temporal properties to be monitored. Imagine for example that the observer monitors the formula: "*always* p", involving the predicate p, and that p is intended to be defined as $p \equiv x > y$, where x and y are static variables defined in a class C. In this case all updates to these variables must be instrumented, such that an update to any of them causes the predicate to be evaluated, and a toggle p to be emitted to the observer in case it has changed. The instrumentation script is written in Java (using reflection), but in essence can be represented as follows:

```
monitor C.x, C.y;
proposition p is C.x > C.y;
```

The code will then be instrumented to emit changes in the predicate p. More specifically, first the initial value of the predicate is transmitted to the observer. Subsequently, whenever one of the two variables is updated, the predicate is evaluated, and in case its value has changed since last evaluation, the predicate name p is transmitted to the observer as a toggle. The observer keeps track of the value of the predicate, based on its initial value, and the subsequent predicate toggles. Figure 2 shows an execution trace where x and y initially are 0, and then subsequently updated. The corresponding values of p are shown. Also shown are the events that are sent to the observer. That is, the initial value of p and the subsequent p toggles.

3 Finite Trace Past Time LTL

In this section we remind some basic notions of finite trace linear past time temporal logic [27,28], establish some conventions and introduce some operators that we found particularly useful for runtime monitoring. We emphasize that the semantics of past time LTL can be elegantly defined recursively, thus allowing us to implement

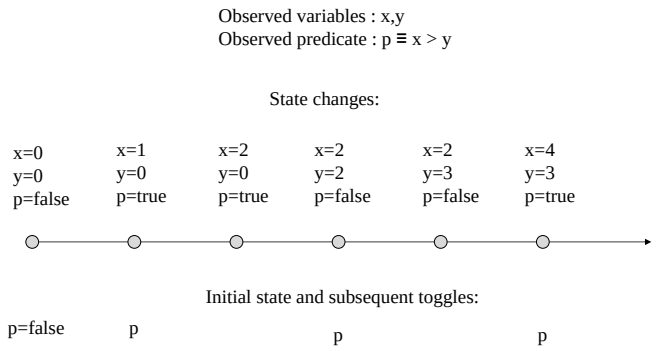

Fig. 2. Events corresponding to observing predicate $p \equiv x > y$.

monitoring algorithms that only need to look one step backwards. We also show that past time LTL can be entirely defined using just the special operators, that were introduced essentially because of practical needs, thus strengthening our belief that past time LTL is an appropriate candidate logic for expressing monitoring safety requirements.

3.1 Syntax

We allow the following constructors for formulae, where A is a finite set of "atomic propositions":

$$\begin{aligned} F ::= {} & \mathit{true} \mid \mathit{false} \mid A \mid \neg F \mid F \ \mathit{op}\ F \\ & \qquad \text{(propositional operators)} \\ & \circ F \mid \diamond F \mid \boxdot F \mid F \ \mathcal{S}_s \ F \mid F \ \mathcal{S}_w \ F \\ & \qquad \text{(standard past time operators)} \\ & \uparrow F \mid \downarrow F \mid [F, F)_s \mid [F, F)_w \\ & \qquad \text{(monitoring operators)} \end{aligned}$$

The propositional binary operators, *op*, are the standard ones, that is, disjunction, conjunction, implication, equivalence, and exclusive disjunction.

The standard past time and the monitoring operators are often called "temporal operators", because they refer to other (past) moments in time. The operator $\circ F$ should be read "previously F"; its intuition is that F held at the immediately previous moment in time. $\diamond F$ should be read "eventually in the past F", with the intuition that there is some past moment in time when F was true. $\boxdot F$ should be read "always in the past F", with the obvious meaning. The operator $F_1 \ \mathcal{S}_s \ F_2$, which should be read "$F_1$ strong since F_2", reflects the intuition that F_2 held at some moment in the past and, since then, F_1 held all the time. $F_1 \ \mathcal{S}_w \ F_2$ is a weak version of "since", read "F_1 weak since F_2", saying that either F_1 was true all the time or otherwise $F_1 \ \mathcal{S}_s \ F_2$.

The monitoring operators $\uparrow$, $\downarrow$, $[\_, \_)_s$, and $[\_, \_)_w$ were inspired by work in runtime verification in [26]. We found these operators often more intuitive and compact than the usual past time operators in specifying runtime requirements, despite the fact that they have the same expressive power as the standard ones, as we discovered later. The operator $\uparrow F$ should be read "start F"; it says that the formula F just started to be true, that is, it was false previously but it is true now. Dually, the operator $\downarrow F$ which is read "end F", carries the intuition that F ends to be true, that is, it was previously true but it is false now. The operators $[F_1, F_2)_s$ and $[F_1, F_2)_w$ are read "strong/weak interval F_1, F_2" and they carry the intuition that F_1 was true at some point in the past but F_2 has not been seen to be true since then, including that moment. For example, if START and DOWN are predicates on the state of a web server to be monitored, 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 down recently, meaning that it was either not down at all recently or it was rebooted and since then it was not down.

3.2 Formal Semantics

We next present formally the intuitive semantics described above. We regard a trace as a finite sequence of abstract states. In practice, these states are generated by events emitted by the program or system that we want to observe. Such events could indicate when variables' values are changed or when locks are acquired or released by threads or processes, or even when a physical action takes place, such as opening or closing a valve, a gate, or a door. If s is a state and a is an atomic proposition then $a(s)$ is true if and only if a holds in the state s. Notice that we are loose with respect to what "holds" means, because, depending on the context, it can mean anything. However, in the case of JPAX the atomic predicates are just any Java boolean expressions and their satisfaction is decided by evaluating them in the current state of the Java program. If $t = s_1 s_2 \ldots s_n$ $(n \geq 1)$ is a trace then we let t_i denote the trace $s_1 s_2 \ldots s_i$ for each $1 \leq i \leq n$. The formal semantics of the operators defined in the previous subsection is given in Figure 3.

Notice the special semantics of the operator "previously " on a trace of one state: $s \models \circ F$ 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. Alternatively, one could consider that $\circ F$ 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".

$t \models true$		is always true,
$t \models false$		is always false,
$t \models a$	iff	$a(s_n)$ holds,
$t \models \neg F$	iff	$t \not\models F$,
$t \models F_1\ op\ F_2$	iff	$t \models F_1$ and/or/etc. $t \models F_2$, when op is $\wedge/\vee$/etc.,
$t \models \circ F$	iff	$t' \models F$, where $t' = t_{n-1}$ if $n > 1$ and $t' = t$ if $n = 1$,
$t \models \diamond F$	iff	$t_i \models F$ for some $1 \leq i \leq n$,
$t \models \Box F$	iff	$t_i \models F$ for all $1 \leq i \leq n$,
$t \models F_1\ \mathcal{S}_s\ F_2$	iff	$t_j \models F_2$ for some $1 \leq j \leq n$ and $t_i \models F_1$ for all $j < i \leq n$,
$t \models F_1\ \mathcal{S}_w\ F_2$	iff	$t \models F_1\ \mathcal{S}_s\ F_2$ or $t \models \Box F_1$,
$t \models \uparrow F$	iff	$t \models F$ and $t_{n-1} \not\models F$,
$t \models \downarrow F$	iff	$t_{n-1} \models F$ and $t \not\models F$,
$t \models [F_1, F_2)_s$	iff	$t_j \models F_1$ for some $1 \leq j \leq n$ and $t_i \not\models F_2$ for all $j \leq i \leq n$,
$t \models [F_1, F_2)_w$	iff	$t \models [F_1, F_2)_s$ or $t \models \Box\neg F_2$.

Fig. 3. Semantics of finite trace past time LTL.

3.3 Recursive Semantics

An observation of crucial importance in the design of the subsequent algorithms is that the semantics above can be defined recursively, in such a way that the satisfaction relation for a formula and a trace can be calculated along the execution trace looking only one step backwards, as shown in Figure 4.

For example, according to the formal, nonrecursive, semantics, a trace $t = s_1 s_2 ... s_n$ satisfies the formula $[F_1, F_2)_w$ if and only if either F_2 was false all the time in the past or otherwise F_1 was true at some point and since then F_2 was always false, including that moment. Therefore, in the case of a trace of size 1, i.e., when $n = 1$, it follows immediately that $t \models [F_1, F_2)_w$ if and only if $t \not\models F_2$. Otherwise, if the trace has more than one event then first of all $t \not\models F_2$, and then either $t \models F_1$ or else the prefix trace satisfies the interval formula, that is, $t_{n-1} \models [F_1, F_2)_w$. Similar reasoning applies to the other recurrences.

3.4 Equivalent Logics

We call the past time temporal logic presented above *ptLTL*. 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 *ptLTL*. Let $ptLTL\restriction_{Ops}$ be the restriction of *ptLTL* to the propositional operators plus the operations in *Ops*. Then

Theorem 1. *The following 12 logics are all equivalent to ptLTL:*

1. $ptLTL\restriction_{\{\circ, \mathcal{S}_s\}}$,
2. $ptLTL\restriction_{\{\circ, \mathcal{S}_w\}}$,
3. $ptLTL\restriction_{\{\circ, [)_s\}}$,
4. $ptLTL\restriction_{\{\circ, [)_w\}}$,
5. $ptLTL\restriction_{\{\uparrow, \mathcal{S}_s\}}$,
6. $ptLTL\restriction_{\{\uparrow, \mathcal{S}_w\}}$,
7. $ptLTL\restriction_{\{\uparrow, [)_s\}}$,
8. $ptLTL\restriction_{\{\uparrow, [)_w\}}$,
9. $ptLTL\restriction_{\{\downarrow, \mathcal{S}_s\}}$,
10. $ptLTL\restriction_{\{\downarrow, \mathcal{S}_w\}}$,
11. $ptLTL\restriction_{\{\downarrow, [)_s\}}$,
12. $ptLTL\restriction_{\{\downarrow, [)_w\}}$.

The first two are known in the literature [27].

Proof. We first show the following properties:

1.	$\diamond F = true\ \mathcal{S}_s\ F$
2.	$\Box F = \neg\diamond\ \neg F$
3.	$F_1\ \mathcal{S}_w\ F_2 = (\Box F_1) \vee (F_1\ \mathcal{S}_s\ F_2)$
4.	$\Box F = F\ \mathcal{S}_w\ false$
5.	$\diamond F = \neg\Box\neg F$
6.	$F_1\ \mathcal{S}_s\ F_2 = (\diamond F_2) \wedge (F_1\ \mathcal{S}_w\ F_2)$
7.	$\uparrow F = F \wedge \neg\circ F$
8.	$\downarrow F = \neg F \wedge \circ F$
9.	$[F_1, F_2)_s = \neg F_2 \wedge ((\circ\neg F_2)\ \mathcal{S}_s\ F_1)$
10.	$[F_1, F_2)_w = \neg F_2 \wedge ((\circ\neg F_2)\ \mathcal{S}_w\ F_1)$
11.	$\downarrow F = \uparrow \neg F$
12.	$\uparrow F = \downarrow \neg F$
13.	$[F_1, F_2)_w = (\Box\neg F_2) \vee [F_1, F_2)_s$
14.	$[F_1, F_2)_s = (\diamond F_1) \wedge [F_1, F_2)_w$
15.	$\circ F = (F \rightarrow \neg \uparrow F) \wedge (\neg F \rightarrow\downarrow F)$
16.	$F_1\ \mathcal{S}_s\ F_2 = F_2 \vee [\circ F_2, \neg F_1)_s$

These properties are intuitive and relatively easy to prove. For example, property 15., the definition of $\circ F$ in terms of $\uparrow F$ and $\downarrow F$, says that in order to find out the value of a formula F in the previous state it suffices to look at the value of the formula in the current state and then, if it is true then look if the formula just started to be true or else look if the formula just ended to be true. We next only prove property 10., the proofs of the others are similar and straightforward.

In order to prove 10., one needs to show that for any trace t, it is the case that $t \models [F_1, F_2)_w$ if and only if $t \models_w \neg F_2 \wedge ((\circ\neg F_2)\ \mathcal{S}_w\ F_1)$. We show this by induction on the size of the trace t. If the size of t is 1, that is, if

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

Fig. 4. Recursive semantics of finite trace past time LTL.

$t = s_1$, then

$$
\begin{array}{l}
t \models [F_1, F_2)_w \text{ iff}\\
\quad \text{iff } \ t \not\models F_2\\
\quad \text{iff } \ t \models \neg F_2\\
\quad \text{iff } \ \text{(by "absorption" in boolean reasoning)}\\
\qquad t \models \neg F_2 \text{ and } (t \models F_1 \text{ or } t \models \neg F_2)\\
\quad \text{iff } \ t \models \neg F_2 \text{ and } (t \models F_1 \text{ or } t \models \odot \neg F_2)\\
\quad \text{iff } \ t \models \neg F_2 \wedge ((\odot \neg F_2) \ \mathcal{S}_w \ F_1).
\end{array}
$$

If the size of the trace t is $n > 1$ then

$$
\begin{array}{l}
t \models [F_1, F_2)_w \text{ iff}\\
\quad \text{iff } \ \text{(by the recursive semantics)}\\
\qquad t \not\models F_2 \text{ and } (t \models F_1 \text{ or}\\
\qquad\quad t_{n-1} \models [F_1, F_2)_w)\\
\quad \text{iff } \ \text{(by the induction hypothesis)}\\
\qquad t \not\models F_2 \text{ and } (t \models F_1 \text{ or}\\
\qquad\quad t_{n-1} \models \neg F_2 \wedge ((\odot \neg F_2) \ \mathcal{S}_w \ F_1))\\
\quad \text{iff } \ t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t_{n-1} \models \neg F_2 \text{ and}\\
\qquad\quad t_{n-1} \models (\odot \neg F_2) \ \mathcal{S}_w \ F_1)\\
\quad \text{iff } \ t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t \models \odot \neg F_2 \text{ and}\\
\qquad\quad t_{n-1} \models (\odot \neg F_2) \ \mathcal{S}_w \ F_1)\\
\quad \text{iff } \ \text{(by the recursive semantics)}\\
\qquad t \not\models F_2 \text{ and } t \models (\odot \neg F_2) \ \mathcal{S}_w \ F_1\\
\quad \text{iff } \ t \models \neg F_2 \wedge ((\odot \neg F_2) \ \mathcal{S}_w \ F_1).
\end{array}
$$

Therefore, $[F_1, F_2)_w = \neg F_2 \wedge ((\odot \neg F_2) \ \mathcal{S}_w \ F_1)$.

The equivalences of the 12 logics with *ptLTL* follow now immediately. For example, in order to show the 8th logic, $ptLTL\restriction_{\{\uparrow, [\,)_s\}}$, equivalent to *ptLTL*, one needs to show how the operators $\uparrow$ and $[\_, \_)_s$ can define all the other past time temporal operators. This is straightforward because, 11. shows how $\downarrow$ can be defined in terms of $\uparrow$, 15. shows how $\circ\!\!-F$ can be defined using just $\uparrow$ and $\downarrow$, 16. defines $\mathcal{S}_s$, 1. defines $\diamond$, 2. $\boxdot$, 3. $\mathcal{S}_w$, and 13. defines the weak interval. The interested reader can check the other 11 equivalences of logics. □

Unlike in theoretical research, in practical monitoring of programs we want to have as many temporal operators available as possible 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, so we want to keep the formulae as concise as possible.

4 Monitoring Safety by Rewriting

The architecture of JPaX is such that events extracted from a running program are sent to an observer which decides whether requirements are violated or not. An important concern that we had and are still having at this relatively incipient stage of JPaX, is whether the chosen monitoring logics are expressive enough to specify powerful, practical and interesting requirements. Since flexibility with respect to defining/modifying monitoring logics is a very important factor at this stage, we have developed a rewriting-based framework which allows one to easily and effectively define new logics for runtime analysis and to monitor execution traces against formulae in these logics. We use the rewriting system Maude as a basis for this framework. In the following we first present Maude, and how formulae and important data structures are represented in Maude. Then we describe how basic propositional calculus is defined in Maude, and then how *ptLTL* is defined. Finally, it is described how this Maude definition of *ptLTL* is used for monitoring requirements stated by the user on execution traces.

4.1 Maude

We have implemented our logic-defining framework by rewriting in Maude [5–7]. Maude is a modularized membership equational [31] and rewriting logic [30] 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 in a hierarchical manner, building new theories from old theories. A particular attractive aspect of Maude is its *mix-fix* notation for syntax, which, together with precedence attributes of operators gives us an elegant way to compactly define syntax of logics. For example, the operation declarations[2]:

```
op _/\_ : Expression Expression
              -> Expression [prec 33] .
op _\/_ : Expression Expression
              -> Expression [prec 40] .
op [_,_} : Expression Expression
              -> Expression .
```

[2] These declarations are artificial and intended to explain some of Maude's features; they will not be needed later in the paper.

```
op if_then_else_ : Bool Expression Expression
                -> Expression .
```

define a simple syntax over the sort `Expression`, where conjunction and disjunction are infix operators (the underscores stand for arguments, whose sorts are listed after the colon), while the interval and the conditional are mix-fix: operator and arguments can be mixed. Conjunction binds tighter than disjunction because it has a lower precedence (the lower the precedence the tighter the binding), so one is relieved from having to add useless parentheses to one's formulae.

It is often the case that equational and/or rewriting logics act like foundational 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 logics supported by JPaX are quite compact. They are based on a simple, general architecture to define new logics which we only describe informally in the next subsection. Maude's notation will be introduced "on the fly" as needed.

4.2 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 is rather technical and so will not be given here. 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), `AtomState` for assignments of boolean values to atoms, also called "states", and `AtomState*` for such assignments together with *final* assignments, i.e., those that are followed by the end of a trace, sometimes requiring a special evaluation procedure. A state `As` is made terminal by applying it a 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 which is also provided by JPaX). There are two constants of sort `Formula` provided, namely `true` and `false`, with the obvious meaning. The propositions that hold in a certain program state are generated by the executing instrumented program.

One of the most important operators 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 an atom state (i.e., an assignment from atoms to boolean values), which is either a terminal state (the last in a trace) or not. 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 their trace semantics.

4.3 Propositional Calculus

Propositional calculus should be included in any monitoring logic. Therefore, we begin with the following module which is heavily used in JPaX. It implements an efficient rewriting procedure due to Hsiang [23] to decide validity of propositions, reducing any boolean expression to an exclusive disjunction (formally written `_++_`) of conjunctions (`_/\_`):

```
fmod PROP-CALC is extending 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 .

*** Operational 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 of the simple propositional calculus above shows the flexibility of the mix-fix notation of Maude, which allows us to define the syntax of a logic in the most natural way.

J. of STTT, Volume 6(2), pp 158-173. 2004

The equations above are interpreted as rewriting rules by Maude, so they will be applied from left to right only. However, due to the associativity and commutativity attributes, rewrites as well as matchings are applied *modulo* associativity and commutativity (AC), making therefore the procedure implied by the rewrite rules for propositional calculus above highly non-trivial. As proved by Hsiang [23], the AC rewriting system above has the property that any proposition is reduced to true or false if it is semantically true or false, or otherwise to a canonical form modulo AC; thus two formulae are equivalent if and only if their canonical forms are equal modulo AC. We found this procedure quite convenient so far, being able to efficiently reduce formulae of hundreds of symbols that occurred in practical examples. However, one should of course not expect this procedure to work efficiently on any proposition, because the propositional validity problem is NP-complete.

4.4 Past Time Linear Temporal Logic

Past time LTL can now be implemented on top of the provided logic-defining framework. Our rewriting based implementation below follows the recursive semantics of past time LTL defined in Subsection 3.3, and, it appears similar to the Java implementation used in [26]. We next explain the `PT-LTL` module in detail.

We start by defining the syntax of past time LTL. Since it extends the module `PROP-CALC` of propositional calculus, we only have to define syntax for the temporal operators:

```
fmod PT-LTL is extending PROP-CALC .
  op (*)_ : Formula -> Formula .
                    *** previously
  op <*>_ : Formula -> Formula .
                    *** eventually in the past
  op [*]_ : Formula -> Formula .
                    *** always in the past
  op _Ss_ : Formula Formula -> Formula .
                    *** strong since
  op _Sw_ : Formula Formula -> Formula .
                    *** weak since
  op start : Formula -> Formula .
                    *** start
  op end   : Formula -> Formula .
                    *** end
  op [_,_}s : Formula Formula -> Formula .
                    *** strong interval
  op [_,_}w : Formula Formula -> Formula .
                    *** weak interval
```

We have used a curly bracket to close the intervals because, for some technical parsing related reasons, Maude does not allow unbalanced parentheses in its terms. The syntax above can now be used by users to write monitoring requirements as formulae. These formulae are loaded by JPaX at initialization and then sent to Maude for parsing and processing. When the first event from the instrumented program is received by JPaX, it sends this event to Maude in order to initialize its monitoring data structures associated to its formulae (remember that the recursive definition of past time LTL in Subsection 3.3 treats the first event of the trace differently). This is done by launching the reduction `mkDS(F, As)` in Maude, where `F` is the formula to monitor and `As` is the atom state abstracting the first event generated by the monitored program; `mkDS` is an abbreviation for "make data structure" and is defined below.

Before we define the operation `mkDS`, we first discuss the formula data structures storing not only the formulae but also their current satisfaction status. It is worth noticing that the strong and weak temporal operators have exactly the same recursive semantics starting with the second event. That suggests that we do not need nodes of different type (strong and weak) in the formula data structure once the monitoring process is initialized: the difference between strong and weak versions of an operator are rather represented by the initial values passed as arguments to a single common version of the operator. The following operation declarations therefore define the constructors for these data structures:

```
op atom                  : Atom Bool -> FormulaDS .
op and  :   FormulaDS FormulaDS Bool -> FormulaDS .
op xor  :   FormulaDS FormulaDS Bool -> FormulaDS .
op previously       : FormulaDS Bool -> FormulaDS .
op eventuallyPast   : FormulaDS Bool -> FormulaDS .
op alwaysPast       : FormulaDS Bool -> FormulaDS .
op since  : FormulaDS FormulaDS Bool -> FormulaDS .
op start            : FormulaDS Bool -> FormulaDS .
op end              : FormulaDS Bool -> FormulaDS .
op interval: FormulaDS FormulaDS Bool -> FormulaDS.
```

The first operation defines a cell storing an atomic proposition together with its observed boolean value, while the next two store conjunction and exclusive disjunction nodes. According to the propositional calculus procedure defined in module `PROP-CALC` in Subsection 4.3, these are the only propositional operators that can occur in reduced formulae. The remaining operators are the seven past time temporal operators introduced so far.

An operator that extracts the boolean value associated to a temporal formula is needed in the sequel, so we define it next. The syntax of this operator is `[_] : FormulaDS -> Bool` and it is defined in the module `FORMULA`, together with its obvious equations `[true] = true` and `[false] = false`. Its definition on temporal and propositional and temporal operators follows:

```
var A : Atom .  var B : Bool .
vars D Dx Dy : FormulaDS .
eq [and(Dx,Dy,B)] = B .
eq [xor(Dx,Dy,B)] = B .
eq [atom(A,B)] = B .
eq [previously(D,B)] = B .
eq [eventuallyPast(D,B)] = B .
eq [alwaysPast(D,B)] = B .
eq [since(Dx,Dy,B)] = B .
```

```
  eq [interval(Dx,Dy,B)] = B .
  eq [start(Dx,B)] = B .
  eq [end(Dx,B)] = B .
```

The operation `mkDS` can be defined now. It basically follows the recursive semantics in Subsection 3.3, when the length of the trace is 1:

```
vars X Y : Formula .
op mkDS : Formula AtomState -> FormulaDS .
eq mkDS(true, As) = true .
eq mkDS(false, As) = false .
eq mkDS(A, As) = atom(A, (A{As} == true)) .
eq mkDS(X /\ Y, As) =
   and(mkDS(X,As), mkDS(Y,As),
       [mkDS(X,As)] and [mkDS(X,As)]) .
eq mkDS(X ++ Y, As) =
   xor(mkDS(X,As), mkDS(Y,As),
       [mkDS(X,As)] xor [mkDS(X,As)]) .
eq mkDS( (*)X, As) =
   previously(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS( <*>X, As) =
   eventuallyPast(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS( [*]X, As) =
   alwaysPast(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS(X Ss Y, As) =
   since(mkDS(X,As), mkDS(Y,As), [mkDS(Y,As)]) .
eq mkDS(X Sw Y, As) =
   since(mkDS(X,As), mkDS(Y,As),
         [mkDS(X,As)] or [mkDS(Y,As)]) .
eq mkDS(start(X), As) = start(mkDS(X,As),false) .
eq mkDS(end(X), As) = end(mkDS(X,As), false) .
eq mkDS([X,Y}s, As) =
   interval(mkDS(X,As), mkDS(Y,As),
            [mkDS(Y,As)] and not [mkDS(Y,As)]) .
eq mkDS([X,Y}w, As) =
   interval(mkDS(X,As), mkDS(Y,As),
            not [mkDS(Y,As)]) .
```

The data structure associated to a past time formula is essentially its syntax tree augmented with a boolean bit for each node. Each boolean bit will store the result of the satisfaction relation between the current execution trace and the corresponding subformula. The only thing left is to define how the formula data structures, or more precisely their bits, modify when a new event is received. This is defined below, using the operator `_{_} : FormulaDS AtomState -> FormulaDS` provided by the module `Formula`:

```
eq atom(A, B){As} = atom(A, (A{As} == true)) .
eq and(Dx, Dy, B){As} =
   and(Dx{As}, Dy{As},[Dx{As}] and [Dy{As}]) .
eq xor(Dx, Dy, B){As} =
   xor(Dx{As}, Dy{As},[Dx{As}] xor [Dy{As}]) .
eq previously(D,B){As} = previously(D{As},[D]) .
eq eventuallyPast(D, B){As} =
   eventuallyPast(D{As}, [D{As}] or B) .
eq alwaysPast(D, B){As} =
   alwaysPast(D{As}, [D{As}] and B) .
eq since(Dx, Dy, B){As} =
   since(Dx{As}, Dy{As},
         [Dy{As}] or [Dx{As}] and B) .
eq start(Dx,B){As} =
   start(Dx{As}, [Dx{As}] and not B) .
eq end(Dx,B){As} =
   end(Dx{As}, not [Dx{As}] and B) .
eq interval(Dx, Dy, B){As} =
   interval(Dx{As}, Dy{As}, not [Dy{As}] and
            ([Dx{As}] or B)) .
endfm
```

The operator `_==_` is built-in and takes two terms of same sort, reduces them to their normal forms, and then returns true if they are equal and false otherwise.

4.5 Monitoring with Maude

In this subsection we give more details on how the actual rewriting based monitoring process works. When the JPaX system is started, the user is supposed to have already specified several formulae in a file containing monitoring requirements. The first thing JPaX does is to start a Maude process, load the past time LTL semantics described above, and then set Maude to run in its *loop mode*, which is an execution mode in which Maude maintains a state term which the user (potentially another process, such as JPaX) can modify interactively. Then JPaX sends Maude all the requirement formulae that the user wants to monitor. Maude stores them in its loop state and waits for JPaX to send events. Notice that the above is general and applies to any logic.

When JPaX receives the first event from the instrumented program that is relevant for the past time LTL analysis module, it just sends it to Maude. On receiving the first event, say `As`, Maude needs to generate the formula data structures for all the formulae to be monitored. It does so by replacing each formula `F` in the loop state by the normal form of the term `mkDS(F, As)`. Then it waits for JPaX to submit further events. Each time a new relevant event `As` is received by JPaX from the instrumented program, it just forwards it to Maude. Then Maude replaces each formula data structure `D` in its loop state by `D{As}` and then waits for further events. If at any moment `[D]` is false for the data structure `D` associated to a formula `F`, then Maude sends an error message to JPaX , which further warns the user appropriately.

It should be obvious that the runtime complexity of the rewriting monitoring algorithm is $O(m)$ to process an event, where m is the size of the past time LTL formula to monitor. That is, the algorithm only needs to traverse the data structure representing the formula bottom-up for each new event, and update one bit in each node. So the overall runtime complexity is $O(n \cdot m)$, where n is the number of events to be monitored. This is the best one can asymptotically hope from a runtime monitoring algorithm, but of course, there is room for even faster algorithms in practical situations, as the one presented in the next section. The main benefit of the rewriting algorithm presented in this section is that it

J. of STTT, Volume 6(2), pp 158-173. 2004

falls under the general framework by which one can easily add or experiment with new monitoring logics within the JPaX system.

The Maude code performing the above steps is relatively straightforward but rather ugly, so we prefer not to present it here. Additionally, Maude's support for interprocess communication is planned to be changed soon, so this code would become soon obsolete.

5 Synthesizing Monitors for Safety Properties

The rewriting algorithm above is a very good choice in the context of the current version of JPaX, because it gives us flexibility and is efficient enough to process events at a faster rate than they can actually be sent by JPaX. However, there might be situations in which a full scale AC rewriting engine like Maude is not available, such as within an embedded system, or in which as little runtime overhead as possible is allowed, such as in real time applications. In this section we present a dynamic programming based algorithm, also based on the recursive semantics of past time LTL in Subsection 3.3, which takes as input a formula and generates source code which can further be compiled into an efficient executable monitor for that formula. This algorithm can be used in two different ways. On the one hand, it can be used as an efficient external monitor to take an action when a formula is violated, such as to report an error to a user, to reboot the system, to send a message, or even to generate a correcting task. On the other hand, it can be used in a context in which one allows past time LTL annotations in the source code of a program, where the logical annotations can be expanded into source code which is further compiled together with the original program. These two use modes, offline versus inline, are further explained in Subsection 5.4.

5.1 The Algorithm Illustrated by an Example

In this section we show via an example how to generate dynamic programming code for a concrete *ptLTL*-formula. We think that this example would practically be sufficient for the reader to foresee our general algorithm presented in the next subsection. Let $\uparrow p \rightarrow [q, \downarrow (r \vee s))_s$ be the *ptLTL*-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 $\varphi_0, \varphi_1, ..., \varphi_8$ be such an enumeration:

$\varphi_0 = \uparrow p \rightarrow [q, \downarrow (r \vee s))_s$,
$\varphi_1 = \uparrow p$,
$\varphi_2 = p$,
$\varphi_3 = [q, \downarrow (r \vee s))_s$,
$\varphi_4 = q$,
$\varphi_5 = \downarrow (r \vee s)$,
$\varphi_6 = r \vee s$,
$\varphi_7 = r$,
$\varphi_8 = s$.

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 $t = s_1 s_2 ... s_n$ of n events. The generated program will maintain a state via a function *update* : **State** $\times$ *Event* $\rightarrow$ **State**, which updates the state with a given event.

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 $t_i \models \varphi_j$. Then one can fill the table according to the recursive semantics of past time LTL as described in Subsection 3.3. 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 the entire 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 = s1 s2 ... sn;
/* Initialization of state and pre */
state ← update(state, s1);
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, si);
    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.

Initialization The initialization phase consists of initializing the *state* variable and the *pre* array. The first event s_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 fixed *ptLTL* 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 *ptLTL* formula. The memory required is constant, since the length of the two arrays is the size of the *ptLTL* formula. However, one may want to also include the size of the formula, say m, into the analysis; then the time complexity is obviously $\Theta(n \cdot m)$ while memory required is $2 \cdot (m + 1)$ bits. The authors conjecture that it's hard to find an algorithm running faster than the above in practical situations, though some slight optimizations are possible (see Section 5.3).

5.2 The Algorithm Formalized

We now formally describe our algorithm that synthesizes a dynamic programming algorithm from a *ptLTL*-formula. It takes as input a formula and generates a program as the one above, containing a "for" loop which traverses the trace of events, while validating or invalidating the formula. The generated program is printed using the function **output**, which takes one or more string or integer parameters which are concatenated in the output. This algorithm is designed to generate pseudocode, but it can easily be adapted to generate code in any imperative programming language:

```
INPUT: past time LTL formula φ
let φ_0, φ_1, ..., φ_m be the subformulae of φ;
output("State state ← {};");
output("bit pre[0..m];");
output("bit now[0..m];");
output("INPUT: trace t = s_1 s_2 ... s_n;");
output("/* Initialization of state and pre */");
output("state ← update(state, s_1);");
for j = m downto 0 do {
    output("    pre[", j, "] ← ");
    if φ_j is a variable then
        output(φ_j, "(state);");
    if φ_j is true then output("true;");
    if φ_j is false then output("false;");
    if φ_j = ¬φ_j' then
        output("not pre[",j', "];");
    if φ_j = φ_j1 op φ_j2 then
        output("pre[",j_1, "] op pre[",j_2, "];");
    if φ_j = ⊙φ_j1 then
        output("pre[", j_1, "];");
    if φ_j = ⊙φ_j1 then
        output("pre[", j_1, "];");
    if φ_j = ⊡φ_j1 then
        output("pre[", j_1, "];");
    if φ_j = φ_j1 S_s φ_j2 then
        output("pre[", j_2, "];");
    if φ_j = φ_j1 S_w φ_j2 then
        output("pre[",j_1, "] or pre[", j_2, "];");
    if φ_j = [φ_j1, φ_j2)_s then
        output("pre[",j_1, "] and not pre[", j_2, "];");
    if φ_j = [φ_j1, φ_j2)_w then
```

J. of STTT, Volume 6(2), pp 158-173. 2004


```
            output("not pre[", j2, "];");
        if φj =↑ φj' then output("false;");
        if φj =↓ φj' then output("false;");
    };
    output("/* Event interpretation loop */");
    output("for i = 2 to n do {");
    for j = m downto 0 do {
        output("      now[", j, "] ← ");
        if φj is a variable then output(φj, "(state);");
        if φj is true then output("true;");
        if φj is false then output("false;");
        if φj = ¬φj' then output("not now[",j', "];");
        if φj = φj1 op φj2 then
            output("now[",j1, "] op now[", j2, "];");
        if φj = ⊙φj1 then output("pre[", j1, "];");
        if φj = ⟐φj1 then
            output("pre[", j, "] or now[",j1, "]");
        if φj = ⊡φj1 then
            output("pre[", j, "] and now[",j1, "]");
        if φj = φj1 Ss φj2 then
            output("(pre[", j, "] and now[",j1, "]) or
                now[", j2, "];");
        if φj = φj1 Sw φj2 then
            output("(pre[", j, "] and now[",j1, "]) or
                now[", j2, "];");
        if φj = [φj1, φj2)s then
            output("(pre[", j, "] or now[",j1, "]) and
                not now[", j2, "];");
        if φj = [φj1, φj2)w then
            output("(pre[", j, "] or now[",j1, "]) and
                not now[", j2, "];");
        if φj =↑ φj' then
            output("now[", j', "] and
                not pre[", j', "];");
        if φj =↓ φj' then
            output("not now[", j', "] and
                pre[", j', "];");
    };
    output("      if now[0] = 0 then
            output(''property violated'');");
    output("      pre ← now;");
    output("}");
```

op is any binary propositional connective. Since we have already given a detailed explanation of the example in the previous section, we shall only give a very brief description of this algorithm.

The formula should be first visited top down to assign increasing numbers to subformulae as they are visited. Let $\varphi_0, \varphi_1, ..., \varphi_m$ be the list of all subformulae. Because of the recursive nature of *ptLTL*, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i-1} \models \varphi_{j'}$ for all $j \leq j' \leq m$.

Before we generate the main loop, we should first generate code for initializing the array $pre[0..m]$, basically giving it the truth values of the subformulae on the initial state, conceptually being an infinite trace with repeated occurrences of the initial state. After that, the generated main event loop will process the events. The loop body will update/calculate the array *now* and in the end will move it into the array *pre* to serve as basis for the next iteration. After each iteration i, $now[0]$ tells whether the formula is validated by the trace $s_1 s_2 ... s_i$.

Since the formula enumeration procedure is linear, the algorithm synthesizes a dynamic programming algorithm from an *ptLTL* formula in linear time with the size of the formula. The boolean operations used above are usually very efficiently implemented on any microprocessor and the arrays of bits *pre* and *now* are small enough to be kept in cache. Moreover, the dependencies between instructions in the generated "for" loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine's resources. Consequently, the generated code is expected to run very fast. We shall next illustrate how such optimizations can be part of the translation algorithm.

5.3 Optimizing the Generated Code

The generated code presented in Subsection 5.1 is not optimal. Even though a smart compiler can in principle generate good machine code from it, 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 ← update(state, si)
now[3] ← r(state) or s(state)
now[2] ← (pre[2] or q(state)) and
         not (not now[3] and pre[3])
now[1] ← p(state)
if ((not (now[1] and not pre[1]) or now[2]) = 0)
    then output(''property violated'');
```

that can be further optimized by boolean simplifications:

```
state ← update(state, si)
now[3] ← r(state) or s(state)
now[2] ← (pre[2] or q(state)) and
         (now[3] or not pre[3])
now[1] ← 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 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 the inline monitoring approach, one would of course want to evaluate the atomic predicates 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, more precisely, as a term over the conditional operation $\_?\_:\_$, where '$e_1?e_2:e_3$' means: *"if e_1 then e_2 else e_3"*. For example, $pre[3]\ ?\ pre[2]\ ?\ now[3]\ :\ q(state)\ :\ pre[2]\ ?\ 1\ :\ q(state)$ (see [18] for a formal definition). Therefore, one is faced with the following optimization problem:

> Given a boolean formula φ using propositions a_1, a_2, ..., a_n of costs c_1, c_2, ..., c_n, respectively, find a $(\_?\_:\_)$-expression that optimally implements φ.

We have implemented a procedure in Maude, on top of propositional calculus, 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, ignoring the costs. Applied on the code above, it yields:

$$
\begin{array}{l}
state \leftarrow update(state, s_i)\\
now[3] \leftarrow r(state)\ ?\ 1\ :\ s(state)\\
now[2] \leftarrow pre[3]\ ?\ pre[2]\ ?\ now[3]\ :\\
\qquad\qquad q(state)\ :\ pre[2]\ ?\ 1\ :\ q(state)\\
now[1] \leftarrow p(state)\\
\textbf{if}\ (pre[1]\ ?\ 0\ :\ now[2]\ ?\ 0\ :\ now[1])\\
\qquad \textbf{then output}(\texttt{``property violated''});
\end{array}
$$

We would like to extend our procedure to take the evaluation costs of 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 at this incipient stage 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.

5.4 Implementation of Offline and Inline Monitoring

In this section we briefly describe our efforts to implement the above described algorithm to create monitors for observing the execution of Java programs in PathExplorer. We present two approaches that we have pursued. In the *off-line* approach we create a monitor that runs in parallel with the executing program, potentially on a different computer, receiving events from the running program, and checking on-the-fly that the formulae are satisfied. In this approach the formulae to be checked are given in a separate specification. In the *inline* approach, formulae are written as comments in the program text, and are then expanded into Java code that is inserted after the comments.

5.4.1 Offline Monitoring

The code generator for off-line monitoring has been written in Java, using JavaCC [24], an environment for writing parsers and for generating and manipulating abstract syntax trees. The input to the code generator is a specification given in a file separate from the program. The specification for our example looks as follows (the default interpretation of intervals is "strong"):

```
specification Example is
 P = start(p) -> [q,end(r|s));
end
```

Several named formulae can be listed; here we have only included one, named `P`. The translator reads this specification and generates a single Java class, called `Formulae`, which contains all the machinery for evaluating all the formulae (in this case one) in the specification. This class must then be compiled and instantiated as part of the monitor. The class contains an `evaluate()` method which is applied after each state change and which will evaluate all the formulae. The class constructor takes as parameter a reference to the object that represents the state, such that any updates to the state by the monitor, based on received events, can be seen by the `evaluate()` method. The generated `Formulae` class for the above specification looks as follows:

```
class Formulae{
 abstract class Formula{
  protected String name;  protected State state;
  protected boolean[] pre; protected boolean[] now;

  public Formula(String name,State state){
   this.name = name; this.state = state;
  }
  public String getName(){return name;}
  public abstract boolean evaluate();
 }
 private List formulae = new ArrayList();
 public void evaluate(){
  Iterator it = formulae.iterator();
  while(it.hasNext()){
   Formula formula = (Formula)it.next();
   if(!formula.evaluate()){
    System.out.println("Property " + formula.getName() +
                       " violated");
 }}}
 class Formula_P extends Formula{
  public boolean evaluate(){
```

```
    now[8] = state.holds("s");
    now[7] = state.holds("r");
    now[6] = now[7] || now[8];
    now[5] = !now[6] && pre[6];
    now[4] = state.holds("q");
    now[3] = (pre[3] || now[4]) && !now[5];
    now[2] = state.holds("p");
    now[1] = now[2] && !pre[2];
    now[0] = !now[1] || now[3];
    System.arraycopy(now,0,pre,0,9);
    return now[0];
   }
   public Formula_P(State state){
    super("P",state);
    pre = new boolean[9]; now = new boolean[9];
    pre[8] = state.holds("s");
    pre[7] = state.holds("r");
    pre[6] = pre[7] || pre[8];
    pre[5] = false;
    pre[4] = state.holds("q");
    pre[3] = pre[4] && !pre[5];
    pre[2] = state.holds("p");
    pre[1] = false;
    pre[0] = !pre[1] || pre[3];
   }
  }
  public Formulae(State state){
   formulae.add(new Formula_P(state));
  }
 }
```

The class contains an inner abstract[3] class `Formula` and, in the general case, an inner class `Formula_X` extending the `Formula` class for each formula in the specification, where `X` is the formula's name. In our case there is one such `Formula_P` class. The abstract `Formula` class declares the `pre` and `now` arrays, without giving them any size, since this is formula specific. An abstract `evaluate` method is also declared. The class `Formula_P` contains the real definition of this `evaluate()` method. The constructor for this class in addition initializes the sizes of `pre` and `now` depending on the size of the formula, and also initializes the `pre` array.

In order to handle the general case where several formulae occur in the specification, and hence many `Formula_X` classes are defined, we need to create instances for all these classes and store them in some data structure where they can be accessed by the outermost `evaluate()` method. The `formulae` list variable is initialized to contain all these instances when the constructor of the `Formulae` class is called. The outermost `evaluate()` method, on each invocation, goes through this list and calls `evaluate()` on each single formula object.

5.4.2 Inline Monitoring

The general architecture of PAX was mainly designed for offline monitoring in order to accommodate applications where the source code is not available or where the monitored process is not even a program, but some kind of physical device. However, it is often the case that the source code of an application *is* available and that one is willing to accept extra code for testing purposes. Inline monitoring has actually higher precision because one knows exactly where an event was emitted in the execution of the program. Moreover, one can even throw exceptions when a safety property is violated, like in Temporal Rover [10], so the running program has the possibility to recover from an erroneous execution or to guide its execution in order to avoid undesired behaviors.

In order to provide support for inline monitoring, we developed some simple scripts that replace temporal annotations in Java source code by actual monitoring code, which throws an exception when the formula is violated. In [14] we show an example of expanded code for future time LTL. The "for" loop and the update of the state in the generic algorithm in Section 5.1 are not needed anymore because the atomic predicates use directly the current state of the program when the expanded code is reached during the execution. In [4] the tool Java-MoP is described, which implements the presented algorithm as a logic plug-in for inline monitoring (as well as for off-line monitoring).

The following code snippets illustrate the inline approach. Assume a class `A`, that defines four integer variables and a method `m`, which contains the past time temporal logic formula from above. Now the propositions `p`, `q`, `r` and `s` are defined to refer to the four variables. The intention is that whenever the program point of the comment is reached, the formula will be evaluated.

```
class A{
  int a,b,c,d;
  void m(){
    ...
    /* @monitor
       proposition p = a>0;
       proposition q = b>0;
       proposition r = c>0;
       proposition s = d>0;
       property P = start(p) -> [q,end(r|s));
    */
    ...
  }
}
```

This class is now automatically translated into the following, where code representing the semantics of the formula has been inserted at the position of the formula comment, and in the constructor:

```
class A{
  int a,b,c,d;
  boolean[] pre = new boolean[9];
  boolean[] now = new boolean[9];

  public A(){
    pre[8] = d>0;
    pre[7] = c>0;
    pre[6] = pre[7] || pre[8];
    pre[5] = false;
    pre[4] = b>0;
    pre[3] = pre[4] && !pre[5];
    pre[2] = a>0;
    pre[1] = false;
    pre[0] = !pre[1] || pre[3];
  }

  void m(){
    ...
    now[8] = d>0;
    now[7] = c>0;
    now[6] = now[7] || now[8];
    now[5] = !now[6] && pre[6];
    now[4] = b>0;
```

[3] An abstract class is a class where some methods are abstract, by having no body. Implementations for these methods will be provided in extending subclasses.

```
      now[3] = (pre[3] || now[4]) && !now[5];
      now[2] = a>0;
      now[1] = now[2] && !pre[2];
      now[0] = !now[1] || now[3];
      System.arraycopy(now,0,pre,0,9);
      if(!now[0])throw Violated("P");
      ...
    }
  }
```

It is essentially the same code as in the offline case, except that the looping constructs have been removed.

It is inline monitoring that motivated us to optimize the generated code as much as possible as in Subsection 5.3. Since the running program and the monitor are a single process now, the time needed to execute the monitoring code can significantly influence the otherwise normal execution of the monitored program.

6 Conclusion

Two efficient algorithms for monitoring safety requirements expressed using past time linear temporal logic were presented, one based on rewriting and implemented in Maude, and the other based on dynamic programming, synthesizing specialized monitors from formulae. They both check that a finite sequence of events emitted by a running program satisfies a formula. Operators convenient for monitoring were considered and shown equivalent to standard past time temporal operators.

These algorithms have been implemented in PathExplorer, a runtime verification tool currently under development. The synthesis algorithm has also been implemented (as a plug-in) in the Java-MoP tool [4], which is a general framework for supporting program monitoring for user provided logics; and in the JMPaX tool [35], which extends part of this work to partial order models instead of simple traces.

It is our intention to investigate how the presented algorithms can be refined to work for a logic that combines past and future time temporal logic and that can refer to real-time and data values. Other kinds of runtime verification are also investigated, such as, for example, techniques for detecting error potentials in multithreaded programs. Recent work on detecting high-level data races is described in [2].

A number of experiments have been carried out with PathExplorer on a planetary rover application written in 35,000 lines of C++. The experiments range from concurrency analysis (deadlock and data race analysis) to monitoring of temporal logic formulae combined with test case generation, as described in [1]. A model checker is used to generate test cases, where a test case consists of input to the application plus a set of temporal formulae that the execution of the application on that input must satisfy. When running this testing environment, hundreds of test cases are generated and the execution of these are monitored against the generated formulae. Initial experiments have been made with a logic that combines past time and future time temporal logic and supports real-time and data reasoning. A bug was detected in the rover application in the very first such experiment we made. A thread did not detect a premature termination of a certain task in a timely manner. The programmer had forgotten to insert this termination check and was reminded by a single run of the testing environment. This testing environment is planned to become part of the rover application programmer's testing toolbox.

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, editors, *Abstract State Machines 2003, LNCS 2589, Taormina, Italy*, pages 87–107. Springer, March 2003.
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*, April 2003. Angers, France.
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*, LNCS, pages 268–283, Genova, Italy, April 2001. Springer-Verlag.
4. F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proceedings of the 3rd Workshop on* Runtime Verification (RV'03), volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 106–125. Elsevier Science, 2003.
5. 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, Mar. 1999. Maude System documentation at `http://maude.csl.sri.com/papers`.
6. M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag. System Description.
7. M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. A Maude Tutorial, Mar. 2000. Manuscript at `http://maude.csl.sri.com/papers`.
8. 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*, pages 439–448, Limerick, Ireland, June 2000. ACM Press.
9. C. Demartini, R. Iosif, and R. Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.

10. D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
11. D. M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of the 1st Conference on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 409–448. Springer, 1989.
12. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, Jan. 1997. ACM Press.
13. E. Gunter and D. Peled. Tracing the Executions of Concurrent Programs. In K. Havelund and G. Roşu, editors, *Proceedings of Runtime Verification (RV'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
14. K. Havelund, S. Johnson, and G. Roşu. Specification and Error Pattern Based Program Monitoring. In *European Space Agency Workshop on On-Board Autonomy*, pages 323–330, Noordwijk, The Netherlands, 2001. European Space Agency.
15. K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, Aug. 2001.
16. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
17. K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001. Canadian Space Agency. Paper AM126 on CD-ROM.
18. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
19. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
20. K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In M. C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer, 1996.
21. G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
22. 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*, pages 597–607, Los Angeles, California, USA, May 1999. IEEE/ACM.
23. J. Hsiang. *Refutational Theorem Proving using Term Rewriting Systems*. PhD thesis, University of Illinois at Champaign-Urbana, 1981.
24. JavaCC. Url. `http://www.webgain.com/products/java_cc`.
25. JTrek. Web page. `http://www.compaq.com/java/download`.
26. 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*. CSREA Press, 1999.
27. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
28. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
29. N. Markey. Temporal logic with past is exponentially more succinct. *EATCS Bull*, 79:122–128, 2003.
30. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 1:73–155, 1992.
31. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proceedings, WADT'97 (Workshop on Algebraic Development Techniques)*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.
32. D. Y. Park, U. Stern, J. U. Skakkebaek, and D. L. Dill. Java Model Checking. In *Proceedings of the Automated Software Engineering Conference*, pages 253–256. IEEE Computer Society, September 2000.
33. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
34. G. Roşu and K. Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. Technical Report TR 01-08, NASA - RIACS, May 2001.
35. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint* European Software Engineering Conference *and* ACM SIGSOFT Symposium on the Foundations of Software Engineering, (ESEC/FSE'03). ACM, 2003.
36. S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer, 2000.
37. 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*, pages 3–12. IEEE CS Press, Sept. 2000.