Synthesizing Monitors for Safety Properties

Klaus Havelund[1] and Grigore Roşu[2]

[1] Kestrel Technology
[2] Research Institute for Advanced Computer Science
http://ase.arc.nasa.gov/{havelund,grosu}
Automated Software Engineering Group
NASA Ames Research Center
Moffett Field, California, 94035, USA

Abstract. The problem of testing a linear temporal logic (LTL) formula on a finite execution trace of events, generated by an executing program, occurs naturally in runtime analysis of software. An algorithm which takes a past time LTL formula and generates an efficient dynamic programming algorithm is presented. The generated algorithm tests whether the formula is satisfied by a finite trace of events given as input and runs in linear time, its constant depending on the size of the LTL formula. The memory needed is constant, also depending on the size of the formula. Further optimizations of the algorithm are suggested. Past time operators suitable for writing succinct specifications are introduced and shown definitionally equivalent to the standard operators. This work is part of the PathExplorer project, the objective of which it is to construct a flexible framework for monitoring and analyzing program executions.

1 Introduction

The work presented in this paper is part of a project at NASA Ames Research Center, called PathExplorer [10, 9, 5, 8, 19], that aims at developing a practical testing environment for NASA software developers. The basic idea of the project is to extract an execution trace of an executing program and then analyze it to detect errors. The errors we are considering at this stage are multi-threading errors such as deadlocks and data races, and non-conformance with linear temporal logic specifications. Only the latter issue is addressed in this paper.

Linear Temporal Logic (LTL) [18, 16] 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 [11, 6]). However, such 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 [7, 21, 3, 12, 2, 17]. Stateless model checkers [20] 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 NP-complete 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 adhoc 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 TempRover (TR) [4], which has admittedly motivated us in a major way to start this work. In TR, one states LTL properties as annotations of the program, these being then replaced by appropriate code, that is executed whenever reached[1]. The MaC tool [15] is another example of a runtime monitoring tool that has inspired this work. Here Java bytecode is 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 kind of interval logic convenient for expressing monitoring properties in a succinct way. Our main theoretical contribution in this paper is Proposition 1 which shows that the MaC logic, together with 10 others, is equivalent to the standard past time temporal logic. The MaC tool represents a formula as an abstract tree generated by a parser, and repeated evaluations are then done by evaluating the entire tree each time using a general purpose tree traversing evaluation algorithm. What we suggest is to generate a special purpose evaluation program for each formula, that essentially specializes the combination of the general purpose evaluation with the particular parse tree. We further suggest an extra optimization of the generated algorithm, thereby obtaining the most efficient evaluation possible.

Section 2 gives a short description of the PathExplorer architecture, putting the presented work in context. Section 3 discusses various past time logics and shows their equivalences. Section 4 uses an example to present the algorithm for translating a past time formula to code. Section 5 presents a formalization of the algorithm used to generate the code. Section 6 describes our efforts to implement the presented algorithm in PathExplorer for monitoring of Java programs. Section 7 suggests some optimizations and Section 8 concludes the paper.

---

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

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 that they satisfy certain user-given constraints. The constraints can be of different kinds and defined in different languages. Each kind of constraint is represented by a rule. Such a rule in principle implements a particular logic or program analysis algorithm. Currently there are rules for checking deadlock potentials, datarace potentials, and for checking temporal logic formulae in different logics. Amongst the latter, several rules have been implemented for checking future time temporal logic, and the work presented in this paper is the basis for a rule for checking past time logic formulae. In general, the user can program new rules and in this way 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. The dataflow between any two components is a stream of events in simple text format, without any apriori 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 rules. Each rule is running in its own process with one input pipe, only dealing with events that are relevant to the rule. For this purpose each rule is equipped with an event parser. The dispatcher takes as input a configuration script, which specifies from where to read the program execution events, and then a list of commands - a command for each rule that starts the rule in a process.

The program or system to be observed must be instrumented to emit execution events to the dispatcher. We have currently implemented an automated instrumentation module for Java bytecode using the Java bytecode engineering tool JTrek [14]. Given information about what kind of events to be emitted, this module instruments the bytecode by inserting extra code for emitting events. Typically, for temporal logic monitoring, one specifies what variables to be observed and in particular what predicates over these variables. The code will then be instrumented to emit changes in these predicates, more specifically toggles in atomic propositions corresponding to these predicates. The instrumentation module together with PathExplorer is called Java PathExplorer (JPAX).

3 Finite Trace Linear Temporal Logic

We briefly remind the reader the basic notions of finite trace linear past time temporal logic, and also establish some conventions and introduce some operators that we found particularly useful for runtime monitoring. Syntactically, we allow the following formulae, where A is a set of "atomic propositions":

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

The propositional binary operators, op, are the standard ones, 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", $\downarrow$ "end F", and $[F_1, F2)$ "interval F_1, F_2".

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. Such events could indicate when variables are changed or when locks are acquired or released. 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. 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 the operators defined above is:

$$\begin{array}{ll} t \models \mathit{true} & \text{is always true,} \\ t \models \mathit{false} & \text{is always false,} \\ t \models a & \text{iff } a(s_n) \text{ holds,} \\ t \models \neg F & \text{iff it is not the case that } t \models F, \\ t \models F_1 \ \mathit{op} \ F_2 & \text{iff } t \models F_1 \text{ and/or/implies/iff } t \models F_2 \text{, when } \mathit{op} \text{ is } \wedge/\vee/\rightarrow/\leftrightarrow, \\ t \models \odot F & \text{iff } t' \models F \text{, where } t' = t_{n-1} \text{ if } n > 1 \text{ and } t' = t \text{ if } n = 1, \\ t \models \diamond F & \text{iff } t_i \models F \text{ for some } 1 \leq i \leq n, \\ t \models \boxdot F & \text{iff } t_i \models F \text{ for all } 1 \leq i \leq n, \\ t \models F_1 \ \mathcal{S}_s \ F_2 & \text{iff } t_j \models F_2 \text{ for some } 1 \leq j \leq n \text{ and } t_i \models F_1 \text{ for all } j < i \leq n, \\ t \models F_1 \ \mathcal{S}_w \ F_2 & \text{iff } t \models F_1 \ \mathcal{S}_s \ F_2 \text{ or } t \models \boxdot F_1, \\ t \models \uparrow F & \text{iff } t \models F \text{ and it is not the case that } t \models \odot F, \\ t \models \downarrow F & \text{iff } t \models \odot F \text{ and it is not the case that } t \models F, \\ t \models [F_1, F_2)_s & \text{iff } t_j \models F_1 \text{ for some } 1 \leq j \leq n \text{ and } t_i \not\models F_2 \text{ for all } j \leq i \leq n, \\ t \models [F_1, F_2)_w & \text{iff } t \models [F_1, F_2)_s \text{ or } t \models \boxdot \neg F_2. \end{array}$$

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. 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".

The non-standard operators $\uparrow$, $\downarrow$, $[\_, \_)_s$, and $[\_, \_)_w$ were inspired by work in runtime verification in [15]. 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.

What makes past time temporal logic such a good candidate 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:

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

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 propositional operators plus the operations in *Ops*. Then

Proposition 1. *The 12 logics*[2] $ptLTL\restriction_{\{\odot,\mathcal{S}_s\}}$, $ptLTL\restriction_{\{\odot,\mathcal{S}_w\}}$, $ptLTL\restriction_{\{\odot,[)_s\}}$, *and* $ptLTL\restriction_{\{\odot,[)_w\}}$, $ptLTL\restriction_{\{\uparrow,\mathcal{S}_s\}}$, $ptLTL\restriction_{\{\uparrow,\mathcal{S}_w\}}$, $ptLTL\restriction_{\{\uparrow,[)_s\}}$, $ptLTL\restriction_{\{\uparrow,[)_w\}}$, *and* $ptLTL\restriction_{\{\downarrow,\mathcal{S}_s\}}$, $ptLTL\restriction_{\{\downarrow,\mathcal{S}_w\}}$, $ptLTL\restriction_{\{\downarrow,[)_s\}}$, $ptLTL\restriction_{\{\downarrow,[)_w\}}$, *are all equivalent.*

Proof. The equivalences follow by the following easy to check properties:

$$
\begin{array}{rl}
\diamond F &= \mathit{true}\ \mathcal{S}_s\ F \\
\boxdot F &= \neg\diamond \neg F \\
F_1\ \mathcal{S}_w\ F_2 &= (\boxdot F_1) \vee (F_1\ \mathcal{S}_s\ F_2) \\
\hline
\boxdot F &= F\ \mathcal{S}_w\ \mathit{false} \\
\diamond F &= \neg\boxdot\neg F \\
F_1\ \mathcal{S}_s\ F_2 &= (\diamond F_2) \wedge (F_1\ \mathcal{S}_w\ F_2) \\
\hline
\uparrow F &= F \wedge \neg \odot F \\
\downarrow F &= \neg F \wedge \odot F \\
[F_1, F_2)_s &= \neg F_2 \wedge ((\odot \neg F_2)\ \mathcal{S}_s\ F_1) \\
[F_1, F_2)_w &= \neg F_2 \wedge ((\odot \neg F_2)\ \mathcal{S}_w\ F_1) \\
\hline
\downarrow F &= \uparrow \neg F \\
\uparrow F &= \downarrow \neg F \\
[F_1, F_2)_w &= (\boxdot\neg F_2) \vee [F_1, F_2)_s \\
[F_1, F_2)_s &= (\diamond F_1) \wedge [F_1, F_2)_w \\
\hline
\odot F &= (F \to \neg \uparrow F) \wedge (\neg F \to \downarrow F) \\
F_1\ \mathcal{S}_s\ F_2 &= F_2 \vee [\ \odot F_2, \neg F_1)_s
\end{array}
$$

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

For example, the definition of $\odot F$ 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.

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, so we want to keep the formulae as concise as possible.

4 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 section. Let $\uparrow p \to [q, \downarrow (r \vee s))_s$ 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:

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

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 $t = e_1 e_2 ... e_n$ of n events. The generated program will maintain a state via a function *update* : **State** $\times$ *Event* $\to$ **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 $t_i \models \varphi_j$. 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 $state \leftarrow \{\}$;
bit $pre[0..8]$;
bit $now[0..8]$;
INPUT: trace $t = e_1 e_2 ... e_n$;
/* Initialization of *state* and *pre* */
$state \leftarrow update(state, e_1)$;
$pre[8] \leftarrow s(state)$;
$pre[7] \leftarrow r(state)$;
$pre[6] \leftarrow pre[7]\ or\ pre[8]$;
$pre[5] \leftarrow$ **false**;
$pre[4] \leftarrow q(state)$;
$pre[3] \leftarrow pre[4]\ and\ not\ pre[5])$;
$pre[2] \leftarrow p(state)$;
$pre[1] \leftarrow$ **false**;
$pre[0] \leftarrow not\ pre[1]\ or\ pre[3]$;
/* Event interpretation loop */
for $i = 2$ **to** n **do {**
    $state \leftarrow update(state, e_i)$;
    $now[8] \leftarrow s(state)$;
    $now[7] \leftarrow r(state)$;
    $now[6] \leftarrow now[7]\ or\ now[8]$;
    $now[5] \leftarrow not\ now[6]\ and\ pre[6]$;
    $now[4] \leftarrow q(state)$;
    $now[3] \leftarrow (pre[3]\ or\ now[4])\ and\ not\ now[5])$;
    $now[2] \leftarrow p(state)$;
    $now[1] \leftarrow now[2]\ and\ not\ pre[2]$;
    $now[0] \leftarrow not\ now[1]\ or\ now[3]$;
    if $now[0] = 0$ **then output**(`''property violated''`);
    $next \leftarrow now$;
};

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. The latter solution is in fact the one implemented in PAX, where the observer runs in parallel with the executing program.

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 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 the memory required is $2 \cdot (m + 1)$ bits. The authors think that it's hard to find an algorithm running faster than the above in practical situations, though some slight optimizations can be imagined (see Section 7).

5 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. To keep the presentation simple, we only show the code for $ptLTL\lceil_{\{\uparrow,\downarrow,[)_s\}}$ formulae. The generated program is printed using the function **output**, which is overloaded to take one or more text parameters which are concatenated in the output.

INPUT: LTL formula φ
let $\varphi_0, \varphi_1, ..., \varphi_m$ be the subformulae of φ;
output("**State** *state* $\leftarrow \{\}$;");
output("**bit** *pre*$[0..m]$;");
output("**bit** *now*$[0..m]$;");
output("INPUT: trace $t = e_1 e_2 ... e_n$;");
output("/* Initialization of *state* and *pre* */");
output("*state* $\leftarrow$ *update*(*state*, e_1);");
for $j = m$ **downto** 0 **do** {
    output("   *pre*[", j, "] $\leftarrow$ ");
    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 $\varphi_j = \neg\varphi_{j'}$ **then output**("*not pre*[", j', "];");
    if $\varphi_j = \varphi_{j_1}$ *op* φ_{j_2} **then output**("*pre*[", j_1, "] *op pre*[", j_2, "];");
    if $\varphi_j = [\varphi_{j_1}, \varphi_{j_2})_s$ **then output**("*pre*[", j_1, "] *and not pre*[", j_2, "];");
    if $\varphi_j = \uparrow \varphi_{j'}$ **then output**("**false**;");
    if $\varphi_j = \downarrow \varphi_{j'}$ **then output**("**false**;");
};
output("/* Event interpretation loop */");
output("**for** $i = 2$ **to** n **do** {");
for $j = m$ **downto** 0 **do** {
    output("   *now*[", j, "] $\leftarrow$ ");
    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 $\varphi_j = \neg\varphi_{j'}$ **then output**("*not now*[", j', "];");
    if $\varphi_j = \varphi_{j_1}$ *op* φ_{j_2} **then output**("*now*[", j_1, "] *op now*[", j_2, "];");
    if $\varphi_j = [\varphi_{j_1}, \varphi_{j_2})_s$ **then**
        output("(*pre*[", j, "] *or now*[", j_1, "]) *and not now*[", j_2, "];");
    if $\varphi_j = \uparrow \varphi_{j'}$ **then**
        output("*now*[", j', "] *and not pre*[", j', "];");
    if $\varphi_j = \downarrow \varphi_{j'}$ **then**
        output("*not now*[", j', "] *and pre*[", j', "];");
};
output("   **if** *now*$[0] = 0$ **then output**(`''property violated''`);");
output("   *next* $\leftarrow$ *now*;");
output("}");

where *op* is any 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 the 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 insures 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, *next*$[0]$ tells whether the formula is validated by the trace $e_1 e_2 ... e_i$.

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. Later we shall illustrate how such an optimization can be part of the translation algorithm.

6 Implementation of Offline and Inline Monitoring

In this section we briefly describe our efforts to implement in PathExplorer the above described algorithm to create monitors for observing the execution of Java programs. We present two approaches that we have pursued. In the first *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 second *inline* approach, formulae are written as comments in the program text, and are then expanded into Java code that is inserted after the comments.

6.1 Offline Monitoring

The code generator for off-line monitoring has been written in Java, using JavaCC [13], 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 states by the monitor based on received events can bee 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 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 datastructure 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, each time called, goes through this list and calls `evaluate()` on each single formula object.

6.2 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 TempRover [4], 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 [5] we show an example of expanded code for future time LTL. We have not implemented the script to automatically expand past time LTL formulae yet, but the expanded code would essentially look like the body of the method `evaluate()` above. The "for" loop and the update of the state in the generic algorithm in Section 4 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.

It is inline monitoring that motivated us optimize the generated code as much as possible. 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.

7 Optimizing the Generated Code

The generated code presented in Section 4 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):

$$
\begin{array}{l}
\mathit{state} \leftarrow \mathit{update}(\mathit{state}, e_i) \\
\mathit{now}[3] \leftarrow r(\mathit{state}) \ \mathit{or}\ s(\mathit{state}) \\
\mathit{now}[2] \leftarrow (\mathit{pre}[2] \ \mathit{or}\ q(\mathit{state})) \ \mathit{and\ not}\ (\mathit{not\ now}[3] \ \mathit{and}\ \mathit{pre}[3]) \\
\mathit{now}[1] \leftarrow p(\mathit{state}) \\
\textbf{if}\ ((\mathit{not}\ (\mathit{now}[1] \ \mathit{and\ not}\ \mathit{pre}[1]) \ \mathit{or}\ \mathit{now}[2]) = 0) \\
\quad \textbf{then output}(\texttt{``property violated''});
\end{array}
$$

which can be further optimized by boolean simplifications:

$$
\begin{array}{l}
\mathit{state} \leftarrow \mathit{update}(\mathit{state}, e_j) \\
\mathit{now}[3] \leftarrow r(\mathit{state}) \ \mathit{or}\ s(\mathit{state}) \\
\mathit{now}[2] \leftarrow (\mathit{pre}[2] \ \mathit{or}\ q(\mathit{state})) \ \mathit{and}\ (\mathit{now}[3] \ \mathit{or\ not}\ \mathit{pre}[3]) \\
\mathit{now}[1] \leftarrow p(\mathit{state}) \\
\textbf{if}\ (\mathit{now}[1] \ \mathit{and\ not}\ \mathit{pre}[1] \ \mathit{and\ not}\ \mathit{now}[2]) \\
\quad \textbf{then output}(\texttt{``property violated''});
\end{array}
$$

The most expensive part of the code above is clearly the function calls, namely $p(\mathit{state})$, $q(\mathit{state})$, $r(\mathit{state})$, and $s(\mathit{state})$. 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 operation $\_?\_:\_$, for example, $\mathit{pre}[3]\ ?\ \mathit{pre}[2]\ ?\ \mathit{now}[3]\ :\ q(\mathit{state})\ :\ \mathit{pre}[2]\ ?\ 1\ :\ q(\mathit{state})$ (see [9] for a formal definition). Therefore, one is faced with the following optimum 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 [1], 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, ignoring the costs. Applied on the code above, it yields:

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

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.

8 Conclusion

A synthesis algorithm has been described which generates from a past time temporal logic formula an algorithm which checks that a finite sequence of events satisfies the formula. The algorithm has been implemented in PathExplorer, a runtime verification tool currently being developed. Operators convenient for monitoring were presented and shown equivalent to standard past time temporal operators. It is our intention to investigate how the presented algorithm can be refined to work for logics 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 multi-threaded programs.

References

1. Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings, First International Workshop on Rewriting Logic and its Applications.* Elsevier Science, 1996. Volume 4, *Electronic Notes in Theoretical Computer Science.*
2. James Corbett, Matthew B. Dwyer, John Hatcliff, Corina S. Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
3. Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
4. Doron Drusinsky. The Temporal Rover and the ATG Rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
5. Klaus Havelund, Scott Johnson, and Grigore Roşu. Specification and error pattern based program monitoring. In *European Space Agency Workshop on On-Board Autonomy*, Noordwijk, Holland, 2001.

6. Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop*, Paris, France, November 1998. To appear in IEEE Transactions of Software Engineering.
7. Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
8. Klaus Havelund and Grigore 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.
9. Klaus Havelund and Grigore Roşu. Monitoring java programs with java pathexplorer. In Klaus Havelund and Grigore Roşu, editors, *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
10. Klaus Havelund and Grigore Roşu. Monitoring programs using rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*. Institute of Electrical and Electronics Engineers, 2001. San Diego, California.
11. Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer, 1996.
12. Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE'99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
13. JavaCC. Web page. `http://www.webgain.com/products/java_cc`.
14. Compaq JTrek. Web page. `http://www.compaq.com/java/download`.
15. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
16. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
17. David Y.W. Park, Urlich Stern, and David L. Dill. Java Model Checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.
18. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
19. Grigore Roşu and Klaus Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical Report TR 01-08, NASA - RIACS, May 2001.
20. Scott D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer, 2000.
21. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.