617638

A Rewriting-based Approach to Trace Analysis

Klaus Havelund
havelund@email.arc.nasa.gov
Kestrel Technology
NASA Ames Research Center

Grigore Roşu
grosu@cs.uiuc.edu
Department of Computer Science
University of Illinois at Urbana-Champaign

July 24, 2002

Abstract

We present a rewriting-based algorithm for efficiently evaluating future time Linear Temporal Logic (LTL) formulae on finite execution traces online. While the standard models of LTL are infinite traces, finite traces appear naturally when testing and/or monitoring real applications that only run for limited time periods. The presented algorithm is implemented in the Maude executable specification language and essentially consists of a set of equations establishing an executable semantics of LTL using a simple formula transforming approach. The algorithm is further improved to build automata on-the-fly from formulae, using memoization. The result is a very efficient and small Maude program that can be used to monitor program executions. We furthermore present an alternative algorithm for synthesizing provably minimal observer finite state machines (or automata) from LTL formulae, which can be used to analyze execution traces without the need for a rewriting system, and can hence be used by observers written in conventional programming languages. The presented work is part of an ambitious runtime verification and monitoring project at NASA Ames, called PATHEXPLORER, and demonstrates that rewriting can be a tractable and attractive means for experimenting and implementing program monitoring logics.

1 Introduction

Future time Linear Temporal Logic (future time LTL, or LTL for short) was introduced by Pnueli in 1977 [26] for stating properties about reactive and concurrent systems. LTL provides temporal operators that refer to the future/remaining part of an execution trace relative to a current point of reference. The standard models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests. Methods, such as model checking, have been developed for proving programs correct with respect to requirements specified as LTL formulae. Several systems are currently being developed that apply model checking to software systems written in Java, C and C++ [12, 31, 4, 19, 3, 25, 7, 30]. However, for very large systems, there is little hope that one can actually prove correctness, and one must in those cases rely on debugging and testing. To further strengthen system reliability, one may therefore want to monitor a program execution during operation and determine whether it conforms to its LTL specification. Any violation of the specification can then be used to guide the execution into a safe state, either manually or automatically. We present a rewriting algorithm for efficiently *evaluating* LTL formulae on finite execution traces *online*, that is, processing each event as it arrives, in contrast to storing all the events and then analyzing them backwards, as in [28] (see also Subsection 3.3), which requires too much space to be practical.

The important question is how to efficiently test LTL formulae of finite trace models, and the main decision here is what data structure one should use to represent the formula such that it can be used to efficiently analyze the trace as it is traversed. We will present such a data structure. We will present and

implement our logics and algorithms in Maude [1], a high-performance system supporting both membership equational logic [23] and rewriting logic [22]. The current version of Maude can do up to 3 million rewritings per second on 800MHz processors, and its compiled version is intended to support 15 million rewritings per second[1]. The algorithm is expressed as a set of equations establishing an executable semantics of LTL using a simple formula transforming approach. The result is a very efficient and small Maude program that can be used to monitor program executions. The decision to use Maude has made it very easy to experiment with logics and algorithms.

We furthermore present an alternative algorithm for *generating* a minimal special observer finite state machine (FSM), or an automaton, from an LTL formula. Observer FSMs can be used to analyze execution traces without the need for a rewriting system, and can hence be used by observers written in traditional programming languages. The FSM-generator is implemented in Maude in about 200 lines of code.

The idea of using temporal logic in program testing is not new, and has already been pursued in the commercial Temporal Rover tool (TR) [5], and in the MaC tool [21]. Both tools have greatly inspired our work. In [27, 24] various algorithms to generate testing automata from temporal logic formulae are described. Our basic contribution in this paper is to show how a rewriting system, such as Maude, makes it possible to experiment with monitoring logics very efficiently and elegantly, and furthermore can be used as a practical program monitoring engine. This approach makes it possible to formalize ideas in a framework close to standard mathematics. The formula transforming approach suggested is a new and efficient way of testing LTL formulae.

In previous work we described a dynamic programming algorithm for checking LTL formulae on execution traces [28] (see also Subsection 3.3). This algorithm evaluates a formula bottom-up for each point in the trace, going backwards from the final state, towards the initial state. Unfortunately, despite its linear complexity this algorithm cannot be used online. In [17, 10] we dualize the dynamic programming technique and apply it to past time LTL, in which case the trace more naturally can be examined in a forward direction, and show how future time and past time LTL formulae can be embedded as comments in code and get expanded into Java code fragments to get executed whenever reached. [6] presents a Büchi automata inspired algorithm adapted to finite trace LTL. The Maude rewriting implementation of LTL described in this paper, besides its simplicity, elegance and efficiency, offers a greater flexibility in experimenting with temporal logics.

The work in this paper originates in [16] and some of it was presented at the Automated Software Engineering conference [15]. What was not presented in [15] is a significant improvement to the main rewriting algorithm which increases its efficiency by almost an order of magnitude (see Subsection 4.3). Furthermore, this paper presents an algorithm for generating minimal observer automata from LTL formulae (Section 5). This work constitutes part of the PATHEXPLORER project at NASA Ames, and in particular of the Java PATHEXPLORER (JPAX) tool [13, 14] for monitoring Java program executions. JPAX facilitates automated instrumentation of Java byte code, using JTREK [2], which then emits relevant events to an observer during execution (see Figure 1). The observer can be running a Maude process as a special case, hence Maude's rewriting engine can be used to drive a temporal logic operational semantics with program execution events. The observer may run on a different computer, in which case the events are transmitted over a socket. The system is driven by a specification, stating what properties to be proved and what parts of the code to be instrumented. When the observer receives the events it dispatches these to a set of observer rules, each rule performing a particular analysis that has been requested. In addition to checking temporal logic requirements, rules have also been programmed to perform error pattern analysis of multi-threaded programs, identifying deadlock and datarace potentials.

Section 2 contains preliminaries, including an introduction to Maude, propositional logic and the standard definition of propositional LTL with its infinite trace models. Section 3 presents a finite trace semantics

[1] Personal communication by José Meseguer.

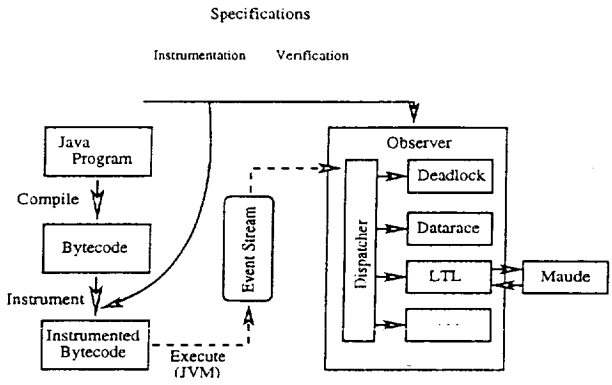

Figure 1: Overview of JPAX

for LTL, its implementation in Maude, as well as briefly a dynamic programming implementation that analyzes the execution trace backwards. Although abstract and elegant, these implementations are either not efficient or not practical; Section 4 presents an efficient and practical implementation using a formula transformation approach, and an even more efficient implementation based on hashing. Section 5 presents an algorithm for generating observer automata from LTL formulae. Such automata may be used in observers written in traditional programming languages, and may in certain cases be more efficient than the rewriting approach. Finally, Section 6 contains conclusions and a description of future work.

2 Preliminaries

This section briefly introduces Maude, a rewriting-based specification and verification system, then a relatively standard procedure to reduce propositional formulae, and then reminds the propositional LTL with its infinite trace models.

2.1 Maude and Logics for Program Monitoring

Maude [1] is a freely distributed high-performance system in the OBJ [9] algebraic specification family, supporting both rewriting logic [22] and membership equational logic [23]. Because of its efficient rewriting engine, able to execute 3 million rewriting steps per second on standard PCs, and because of its metalanguage features, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, and even programming languages. We were delighted to notice how easily we could implement and efficiently validate our algorithms for testing LTL formulae on finite event traces in Maude, admittedly a tedious task in C++ or Java, and hence decided to use Maude at least for the prototyping stage of our runtime check algorithms.

We very briefly and informally remind some of Maude's features, referring the interested reader to the manual [1] for more details. Maude supports modularization in the OBJ style. There are various kinds of modules, but we are using only functional modules which follow the pattern "`fmod <name> is <body> endfm`". The body of a functional module consists of a collection of declarations, of which we are using importing, sorts, subsorts, operations, variables and equations, usually in this order.

We next introduce some modules that we think are general enough to be used within any logical environment for program monitoring that one would want to implement by rewriting. The next one simply defines atomic propositions as an abstract data type having one sort, `Atom` and no operations or constraints:

```
fmod ATOM is sort Atom .  endfm
```

The actual names of atomic propositions will be automatically generated in another module that extends ATOM, as constants of sort Atom. These will be generated by the observer at the initialization of monitoring, from the actual properties that one wants to monitor.

An important aspect of program monitoring is that of an (abstract) execution trace, which consists of a finite list of events. We abstract a single event by a list of atoms, those that hold after the action that generated the event took place. The values of the atomic propositions are updated by the observer according to the actual state of the executing program and then sent to Maude as a term of sort Event:

```
fmod TRACE is protecting ATOM .
  sorts Event Event* Trace .  subsorts Atom < Event < Event* Trace .
  op nil : -> Event .
  op __ : Atom Event -> Event [prec 23] .
  op _* : Event -> Event* .
  op _,_ : Event Trace -> Trace [prec 25] .
endfm
```

The statement protecting ATOM imports the module ATOM. The above is a compact way to use *mix-fix*[2] and order-sorted notation to define an abstract data type of traces: a trace is a comma separated list of events, where an event is itself a list of atoms. The subsorts declaration declares Atom to be a subsort of Event, which in turn is a subsort of Event* as well as of Trace. Since elements of a subsort can occur as elements of a supersort without explicit lifting, we have as a consequence that a single event is also a trace, consisting of this one event. Likewise, an atomic proposition can occur as an event, containing only this atomic proposition. Note that there is no definition of an empty trace. Operations can have attributes, such as the precedences above, which are written between square brackets. The attribute prec gives a precedence to an operator[3], thus eliminating the need for most parentheses. Notice the special sort Event* which stay for terminal events, i.e., events that occur at the end of traces. Any event can potentially occur at the end of a trace. It is often the case that ending events are treated differently, like in the case of finite trace linear temporal logic; for this reason, we have introduced the operation _* which marks an event as terminal.

Syntax and semantics are basic requirements to any logic, in particular to those logics needed for monitoring. The following module introduces what we believe are the basic ingredients of monitoring logics. We found the following very useful for our logics, but of course, the user is free to change it if he/she finds it inconvenient:

```
fmod LOGICS-BASIC is protecting TRACE .
  sort Formula .  subsort Atom < Formula .
  ops true false : -> Formula .
  op [_] : Formula -> Bool [strat (1 0)] .
  eq [true] = true .  eq [false] = false .

  vars A A' : Atom .  var T : Trace .
  var E : Event .  var E* : Event* .
  op _{_} : Formula Event* -> Formula [prec 10] .
  eq true{E*} = true .  eq false{E*} = false .
  eq A{nil} = false .
  eq A{A'}   = if A == A' then true else false fi .
  eq A{A' E} = if A == A' then true else A{E} fi .
  eq A{E *} = A{E} .

  op _|=_ : Trace Formula -> Bool [prec 30] .
  eq T |= true  = true .
  eq T |= false = false .
  eq E   |= A = [A{E}] .
  eq E,T |= A = E |= A .
endfm
```

[2]Underscores are places for arguments.

[3]The lower the precedence number, the tighter the binding.

The first block of declarations introduces the sort `Formula` which can be thought of as a generic sort for any well-formed formula in any logic. There are two designated formulae, namely `true` and `false`, with the obvious meaning, together with a "projection", denoted `[_]`, of any formula into a boolean expression. The only role of this operation is to check whether a logical formula is violated or not, each logic being allowed to refine this operator according to its policy. Its attribute says that this operation should always be evaluated eagerly; numbers in the strategy declaration stay for argument positions that are numbered from left to right, 0 staying for the operator itself. The sort `Bool` is built-in to Maude and has two constants `true` and `false` which are different from those of sort `Formula`, and a generic operator `if_then_else_fi`. The second block defines the operation `_{_}` which takes a formula and an event and yields another formula. The intuition for this operation is that it "evaluates" the formula in the new state and produces a proof obligation as another formula for the subsequent events, if needed. If the returned formula is `true` or `false` then it means that the formula was satisfied or violated, regardless of the rest of the execution trace; in this case, a message can be returned by the observer. As we'll soon see, each logic will further complete the definition of this operator. Finally, the satisfaction relation is defined. That is, two equations deal with the formulae `true` and `false` and should be obvious. The last two equations state that a trace, consisting either of a single event or of several, satisfies an atomic proposition if evaluating that atomic proposition on the event yields true.

2.2 Propositional Calculus

A rewriting decision procedure for propositional calculus due to Hsiang [20] is adapted and presented. It provides the usual connectives `_/\_` (and), `_++_` (exclusive or), `_\/_` (or), `!_` (negation), `_->_` (implication), and `_<->_`(equivalence). The procedure reduces tautology formulae to the constant `true` and all the others to some canonical form modulo associativity and commutativity. An unusual aspect of this procedure is that the canonical forms consist of exclusive or of conjunctions. Even if propositional calculus is very basic to almost any logical environment, we decided to keep it as a separate logic instead of being part of the logic infrastructure of JPAX. One reason for this decision is that its semantics could be in conflict with other logics, for example ones in which conjunctive normal forms are desired.

An OBJ3 code for this procedure appeared in [9]. Below we give its obvious translation to Maude together with its finite trace semantics, noticing that Hsiang [20] showed that this rewriting system modulo associativity and commutativity is Church-Rosser and terminates. The Maude team was probably also inspired by this procedure, since the builtin `BOOL` module is very similar.

```
fmod PROP-CALC is extending LOGICS-BASIC .
*** Constructors ***
  op _/\_ : Formula Formula -> Formula [assoc comm prec 15] .
  op _++_ : Formula Formula -> Formula [assoc comm prec 17] .
  vars X Y Z : Formula .
  eq true /\ X = X .
  eq false /\ X = false .
  eq X /\ X = X .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ (Y ++ Z) = X /\ Y ++ X /\ Z .
*** Derived operators ***
  op _\/_  : Formula Formula -> Formula [assoc prec 19] .
  op !_    : Formula -> Formula [prec 13] .
  op _->_  : Formula Formula -> Formula [prec 21] .
  op _<->_ : Formula Formula -> Formula [prec 23] .
  eq X \/ Y = X /\ Y ++ X ++ Y .
  eq ! X = true ++ X .
  eq X -> Y = true ++ X ++ X /\ Y .
  eq X <-> Y = true ++ X ++ Y .
*** Finite trace semantics
```

```
  var T : Trace .  var E* : Event* .
  eq T |= X /\ Y = T |= X and T |= Y .
  eq T |= X ++ Y = T |= X xor T |= Y .
  eq (X /\ Y)(E*) = X(E*) /\ Y(E*) .
  eq (X ++ Y)(E*) = X(E*) ++ Y(E*) .
  eq [X /\ Y] = [X] and [Y] .
  eq [X ++ Y] = [X] xor [Y] .
endfm
```

Operators are again declared in mix-fix notation and have attributes between squared brackets, such as `assoc`, `comm` and `prec <number>`. Once the module above is loaded[4] in Maude, reductions can be done as follows:

```
red a -> b /\ c  <->  (a -> b) /\ (a -> c) .   ***> should be true
red a <-> ! b .                                ***> should be a ++ b
```

Notice that one should first declare the constants `a`, `b` and `c`. The last six equations are related to the semantics of propositional calculus. Since `[_]` is eagerly evaluated, `[X]` will first evaluate `X` using propositional calculus reasoning and then will apply one of the last two equations if needed; these equations will not be applied normally in practical reductions, they are useful only in the correctness proof in Theorem 1.

2.3 Linear Temporal Logic

Classical LTL provides in addition to the propositional logic operators the temporal operators `[]_` (always), `<>_` (eventually), `_U_` (until), and `o_` (next). An LTL standard model is a function $t : \mathcal{N}^+ \rightarrow 2^{\mathcal{P}}$ for some set of atomic propositions $\mathcal{P}$, i.e., an infinite trace over the alphabet $2^{\mathcal{P}}$, which maps each time point (a natural number) into the set of propositions that hold at that point. The operators have the following interpretation on such an infinite trace. Assume formulae `X` and `Y`. The formula `[]X` holds if `X` holds in all time points, while `<>X` holds if `X` holds in some future time point. The formula `X U Y` (`X` until `Y`) holds if `Y` holds in some future time point, and until then `X` holds (so we consider strict until). Finally, `o X` holds for a trace if `X` holds in the suffix trace starting in the next (the second) time point. The propositional operators have their obvious meaning. As an example illustrating the semantics, the formula `[](X -> <>Y)` is true if for any time point (`[]`) it holds that if `X` is true then eventually (`<>`) `Y` is true. Another similar property is `[](X -> o(Y U Z))`, which states that whenever `X` holds then from the next state `Y` holds until eventually `Z` holds. It's standard to define a core LTL using only atomic propositions, the propositional operators `!_` (not) and `_/\_` (and), and the temporal operators `o_` and `_U_`, and then define all other propositional and temporal operators as derived constructs. Standard equations are `<>X = true U X` and `[]X = !<>!X`.

3 Finite Trace Linear Temporal Logic

As already explained, our goal is to develop a framework for testing software systems using temporal logic. Tests are performed on finite execution traces and we therefore need to formalize what it means for a finite trace to satisfy an LTL formula. We first present a semantics of finite trace LTL using standard mathematical notation. Then we present a specification in Maude of a finite trace semantics. Whereas the former semantics uses universal and existential quantification, the second Maude specification is defined using recursive definitions that have a straightforward operational rewriting interpretation and which therefore can be executed.

[4] Either by typing it or using the command "`in <filename>`".

3.1 Finite Trace Semantics

As mentioned in Subsection 2.1, a trace is viewed as a sequence of program states, each state denoting the set of propositions that hold at that state. We shall outline the finite trace LTL semantics using standard mathematical notation rather than Maude notation. Assume two total functions on traces, *head* : `Trace` $\rightarrow$ `Event` returning the head event of a trace and *length* returning the length of a finite trace, and a partial function *tail* : `Trace` $\rightarrow$ `Trace` for taking the tail of a trace. That is, $head(e,t) = head(e) = e$, $tail(e,t) = t$, and $length(\mathtt{e}) = 1$ and $length(e,t) = 1 + length(t)$. Assume further for any trace t, that t_i denotes the suffix trace that starts at position i, with positions starting at 1. The satisfaction relation $\models \subseteq$ `Trace` $\times$ `Formula` defines when a trace t satisfies a formula f, written $t \models f$, and is defined inductively over the structure of the formulae as follows, where `A` is any atomic proposition and `X` and `Y` are any formulae:

$t \models$ `A`	iff	`A` $\in head(t)$
$t \models$ `true`	iff	*true*,
$t \models$ `false`	iff	*false*,
$t \models$ `X /\ Y`	iff	$t \models$ `X` and $t \models$ `Y`,
$t \models$ `X ++ Y`	iff	$t \models$ `X` xor $t \models$ `Y`,
$t \models$ `[]X`	iff	$(\forall\, i \leq length(t))\ t_i \models$ `X`
$t \models$ `<>X`	iff	$(\exists\, i \leq length(t))\ t_i \models$ `X`
$t \models$ `X U Y`	iff	$(\exists\, i \leq length(t))\ (t_i \models$ `Y` and $(\forall\, j < i)\ t_j \models$ `X`$)$
$t \models$ `o X`	iff	(if $tail(t)$ is defined then $tail(t) \models$ `X` else $t \models$ `X`$)$

Notice that finite trace LTL can behave quite differently from standard infinite trace LTL. For example, there are formulae which are not valid in infinite trace LTL but valid in finite trace LTL, such as `<>([]A \/ []!A)`, and there are formulae which are satisfiable in infinite trace LTL and not satisfiable in finite trace LTL, such as the negation of the above. The formula above is satisfied by any finite trace because the last event/state in the trace either contains `A` or it doesn't.

3.2 Finite Trace Semantics in Maude

Now it can be relatively easily seen that the following Maude specification correctly "implements" the finite trace semantics of LTL described above. The only important deviation from the rigorous mathematical formulation described above is that the quantifiers over finite sets of indexes are expressed recursively.

```
fmod LTL is extending PROP-CALC .
*** syntax
  op []_ : Formula -> Formula [prec 11] .
  op <>_ : Formula -> Formula [prec 11].
  op _U_ : Formula Formula -> Formula [prec 14] .
  op o_  : Formula -> Formula [prec 11] .
*** semantics
  vars X Y : Formula .
  var E : Event .  var T : Trace .
  eq E   |= [] X  = E |= X .
  eq E,T |= [] X  = E,T |= X and T |= [] X .
  eq E   |= <> X  = E |= X .
  eq E,T |= <> X  = E,T |= X or T |= <> X .
  eq E   |= X U Y = E |= Y .
  eq E,T |= X U Y = E,T |= Y or E,T |= X and T |= X U Y .
  eq E   |= o X   = E |= X .
  eq E,T |= o X   = T |= X .
endfm
```

Notice that only the temporal operators needed declarations and semantics, the others being already defined in `PROP-CALC` and `LOGICS-BASIC`, and that the definitions that involved the functions *head* and *tail* were replaced by two alternative equations.

One can now directly verify LTL properties on finite traces using Maude's rewriting engine. Consider as an example a traffic light that switches between the colors *green*, *yellow*, and *red*. The LTL property that after *green* comes *yellow*, and its negation, can now be verified on a finite trace using Maude's rewriting engine, by typing commands to Maude such as (assuming that states can be repeated, making use of the next-operator undesirable):

```
reduce green, yellow, red, green, yellow, red, green, yellow, red, red
       |= [](green -> !red U yellow) .
reduce green, yellow, red, green, yellow, red, green, yellow, red, red
       |= !([](green -> !red U yellow)) .
```

which should return the expected answers, i.e., `true` and `false`, respectively. The algorithm above does nothing but blindly follows the mathematical definition of satisfaction and even runs reasonably fast for relatively small traces. For example, it takes[5] about 30ms (74k rewrite steps) to reduce the first formula above and less than 1s (254k rewrite steps) to reduce the second on traces of 100 events (10 times larger than the above). Unfortunately, this algorithm doesn't seem to be tractable for large event traces, even if run on very fast platforms. As a concrete practical example, it took Maude 7.3 million rewriting steps (3 seconds) to reduce the first formula above and 2.4 billion steps (1000 seconds) for the second on traces of 1,000 events; it couldn't finish in one night (more than 10 hours) the reduction of the second formula on a trace of 10,000 events. Since the event traces generated by an executing program can easily be larger than 10,000 events, the trivial algorithm above can not be used in practice.

A rigorous complexity analysis of the algorithm above is hard (because it has to take into consideration the evaluation strategy used by Maude for terms of sort `Bool`) and not worth the effort. However, a simplified analysis can be easily made if one only counts the maximum number of atoms of the form `event |= atom` that can occur during the rewriting of a satisfaction term, as if all the boolean reductions were applied after all the other reductions: let us consider a formula `X = [](([](...([]A)...))` where the always operator is nested m times, and a trace `T` of size n, and let $T(n, m)$ be the total number of basic satisfactions `event |= atom` that occur in the normal form of the term `T |= X` if no boolean reductions were applied. Then, the recurrence formula $T(n, m) = T(n-1, m) + T(n, m-1)$ follows immediately from the specification above. Since $\binom{m}{n} = \binom{m}{n-1} + \binom{m-1}{n-1}$, it follows that $T(n, m) > \binom{m}{n}$, that is, $T(n, m) = \Theta(n^m)$, which is of course unacceptable.

3.3 Traversing the Execution Trace Backwards

The satisfaction relation above for finite trace LTL can therefore be defined recursively, both on the structure of the formulae and on the size of the execution trace. As is often the case for functions defined this way, an efficient dynamic programming algorithm can be generated from any LTL formula. We only show such an algorithm for a particular formula, referring the interested reader to [28] which shows how these algorithms can be generated in linear time and space. The formula we choose below is artificial (and will not be used later in the paper), but contains all four temporal operators.

Let $\Box((p\ \mathcal{U}\ q) \rightarrow \Diamond(q \rightarrow \circ r))$ be an LTL formula and let $\varphi_1, \varphi_2, ..., \varphi_{10}$ be its subformulae, in breadth-first order:

[5] On a 1.7GHz, 1Gb memory PC.

$$
\begin{aligned}
\varphi_1 &= \Box((p\,\mathcal{U}\,q) \rightarrow \Diamond(q \rightarrow \circ r)),\\
\varphi_2 &= (p\,\mathcal{U}\,q) \rightarrow \Diamond(q \rightarrow \circ r),\\
\varphi_3 &= p\,\mathcal{U}\,q,\\
\varphi_4 &= \Diamond(q \rightarrow \circ r),\\
\varphi_5 &= p,\\
\varphi_6 &= q,\\
\varphi_7 &= q \rightarrow \circ r,\\
\varphi_8 &= q,\\
\varphi_9 &= \circ r,\\
\varphi_{10} &= r.
\end{aligned}
$$

Let also *now[1..10]* and *next[1..10]* be two arrays of bits (or boolean values); their length is exactly the size of the LTL formula φ. Then the following algorithm can be generated in linear time. This algorithm will take as input an execution trace t and returns 0 or 1, saying whether the formula φ was or not violated by the trace t.

INPUT: trace $t = e_1 e_2 ... e_n$
next[10] $\leftarrow (r \in e_n)$;
next[9] $\leftarrow$ *next*[10];
next[8] $\leftarrow (q \in e_n)$;
next[7] $\leftarrow$ *next*[8] *implies next*[9];
next[6] $\leftarrow (q \in e_n)$;
next[5] $\leftarrow (p \in e_n)$;
next[4] $\leftarrow$ *next*[7];
next[3] $\leftarrow$ *next*[6];
next[2] $\leftarrow$ *next*[3] *implies next*[4];
next[1] $\leftarrow$ *next*[2];
for $i = n - 1$ **downto** 1 **do** {
    now[10] $\leftarrow (r \in e_i)$;
    now[9] $\leftarrow$ *next*[10];
    now[8] $\leftarrow (q \in e_i)$;
    now[7] $\leftarrow$ *now*[8] *implies now*[9];
    now[6] $\leftarrow (q \in e_i)$;
    now[5] $\leftarrow (p \in e_i)$;
    now[4] $\leftarrow$ *now*[7] *or next*[4];
    now[3] $\leftarrow$ *now*[6] *or* (*now*[5] *and next*[3]);
    now[2] $\leftarrow$ *now*[3] *implies now*[4];
    now[1] $\leftarrow$ *now*[2] *and next*[1];
    next $\leftarrow$ *now* }
output(*next*[1]);

The algorithm above can be further optimized, noticing that only the bits 10, 4, 3 and 1 are needed in the vectors *now* and *next*, as we did for past time LTL in [17]. The analysis of this algorithm is straightforward. Its time complexity is $\Theta(n \cdot m)$ while the memory required is $2 \cdot m$ bits, where n is the length of the trace and m is the size of the LTL formula.

The dynamic programming technique presented in this subsection is as efficient as one can hope, but, unfortunately, has a major drawback: it needs to traverse the execution trace backwards. From a practical perspective, that means that the instrumented program is run for some period of time while its execution trace is saved, and then, after the program was stopped, its execution trace is traversed backwards and (efficiently) analyzed. Besides the obvious inconvenience due to storing potentially huge execution traces, this method cannot be used to monitor programs online in practice, synchronously, that is, to issue a warning *exactly* at the time when the event violating the formula was observed.

4 An Efficient Rewriting Algorithm

In this section we shall present a more efficient rewriting semantics for LTL, based on the idea of consuming the events in the trace, one by one, and updating a data structure (which is also a formula) corresponding to the effect of the event on the value of the formula. An important advantage of this algorithm is that it often detects when a formula is violated or validated before the end of the execution trace, so, unlike the algorithms above, it is suitable for online monitoring. Our decision to write an operational semantics this way was motivated by an attempt to program such an algorithm in Java, where such a solution would be natural. The presented rewriting-based algorithm is linear in the size of the execution trace and worst-case exponential in the size of the monitored LTL formula.

4.1 The Main Algorithm

We implement this algorithm by extending the definition of the operation `_{_} : Formula Event* -> Formula` to temporal operators, with the following intuition. Assuming a trace `E,T` consisting of an event `E` followed by a trace `T`, then a formula `X` holds on this trace if and only if `X{E}` holds on the remaining trace `T`. If the event `E` is terminal then `X{E *}` holds if and only if `X` holds under standard LTL semantics on the infinite trace containing only the event `E`.

```
fmod LTL-REVISED is protecting LTL .
  vars X Y : Formula .
  var E : Event .  var T : Trace .
  eq ([] X){E}   = [] X /\ X{E} .
  eq ([] X){E *} = X{E *} .
  eq (<> X){E} = <> X \/ X{E} .
  eq (<> X){E *} = X{E *} .
  eq (o X){E} = X .
  eq (o X){E *} = X{E *} .
  eq (X U Y){E} = Y{E} \/ X{E} /\ X U Y .
  eq (X U Y){E *} = Y{E *} .

  op _|-_ : Trace Formula -> Bool [strat (2 0)] .
  eq E   |- X = [X{E *}] .
  eq E,T |- X = T |- X{E} .
endfm
```

The rule for the temporal operator `[]X` should be read as follows: the formula `X` must hold now (`X{E}`) and also in the future (`[]X`). The sub-expression `X{E}` represents the formula that must hold for the rest of the trace for `X` to hold now. As an example, consider the formula `[]<>A`. This formula modified by an event `B C` (so `A` doesn't hold) yields the rewriting sequence `([]<>A){B C}` → `[]<>A /\ (<>A){B C}` → `[]<>A /\ (<>A \/ A{B C})` → `[]<>A /\ (<>A \/ false)` → `[]<>A /\ <>A`, while the same formula transformed by `A C` (so `A` holds) yields `([]<>A){A C}` → `[]<>A /\ (<>A){A C}` → `[]<>A /\ (<>A \/ A{A C})` → `[]<>A /\ (<>A \/ true)` → `[]<>A /\ true` → `[]<>A`, i.e., the same formula. Note that these rules spell out the semantics of each temporal operator. An alternative solution would be to define some operators in terms of others, as is typically the case in the standard semantics for

LTL. For example, we could introduce an equation of the form: `<>X = true U X`, and then eliminate the rewriting rule for `<>X` in the above module. This turns out to be less efficient because more rewrites are needed.

This module eventually defines a new satisfaction relation `_|-_` between traces and formulae. The term `T |- X` is evaluated now by an iterative traversal over the trace, where each event transforms the formula. Note that the new formula that is generated in each step is always kept small by being reduced to normal form via the equations in the PROP-CALC module in Subsection 2.2. In fact, the new formula consists of boolean combinations of sub-formulae of the initial formula, kept in a minimal canonical form. Therefore, the algorithm is linear in the size of the trace, and worst-case exponential in the size of the formula. However, it seems that this exponential complexity in the size of the formula is more of theoretical importance than practical, since in general the size of the formula grew only twice or less in our experiments.

Verification results are very encouraging and show that this optimized semantics is orders of magnitudes faster than the first semantics. Traces of less than 10,000 events are verified in milliseconds, while traces of 100,000 events never needed more than 3 seconds. This technique scales quite well; we were able to monitor even traces of hundreds of millions events. As a concrete example, we created an artificial trace by repeating 10 million times the 10 event trace in Subsection 3.2, and then checked it against the formula `[](green -> !red U yellow)`. There were needed 4.9 billion rewriting steps for a total of about 1,500 seconds. In Subsection 4.3 we will see how this algorithm can be made even more efficient, using memoization.

4.2 Correctness and Completeness

In this subsection we prove that the algorithm presented above is correct and complete with respect to the semantics of finite trace LTL presented in Section 3. The proof is done completely in Maude, but since Maude is not intended to be a theorem prover, we actually have to generate the proof obligations by hand. However, the proof obligations below could be automatically generated by a proof assistant like KUMO [8] or a theorem prover like PVS [29][6].

Theorem: For any trace `T` and any formula `X`, `T |= X` if and only if `T |- X`.

Proof: By induction, both on traces and formulae. We first need to prove two lemmas, namely that the following two equations hold in the context of both `LTL` and `LTL-REVISED`:

```
(∀ E : Event, X : Formula) E |= X = E |- X,
(∀ E : Event, T : Trace, X : Formula) E T |= X = T |= X(E).
```

We prove them by structural induction on the formula `X`. Constants `e` and `x` are needed in order to prove the first lemma via the theorem of constants. However, since we prove the second lemma by structural induction on `X`, we not only have to add two constants `e` and `t` for the universally quantified variables `E` and `T`, but also two other constants `y` and `z` standing for formulas which can be combined via operators to give other formulas. The induction hypothesis for the second lemma is added to the following specification as equations. Notice that we merged the two proofs to save space. A proof assistant like KUMO or PVS would prove them independently, generating only the needed constants for each of them.

```
fmod PROOF-OF-LEMMAS is
  extending LTL .
  extending LTL-REVISED .
  op e : -> Event .  op t : -> Trace .
  ops a b c : -> Atom .  ops y z : -> Formula .
  eq e |= y = e |- y .
  eq e |= z = e |- z .
  eq e,t |= y = t |= y(e) .
```

[6] We've already done it in PVS, but we prefer to use only Maude in this paper.

```
  eq e,t |= z = t |= z(e) .
  eq b(e) = true .
  eq c(e) = false .
endfm
```

It is worth reminding the reader at this stage that the functional modules in Maude have initial semantics, so proofs by induction are valid. Before proceeding further, the reader should be aware of the operational semantics of the operation _==_, namely that the two argument terms are first reduced to their normal forms which are then compared syntactically (but modulo associativity and commutativity); it returns `true` if and only if the two normal forms are equal. Therefore, the answer `true` means that the two terms are indeed semantically equal, while `false` only means that they couldn't be proved equal; they can still be equal.

```
red   (e |= a      == e |- a)
  and (e |= true   == e |- true)
  and (e |= false  == e |- false)
  and (e |= y /\ z == e |- y /\ z)
  and (e |= y ++ z == e |- y ++ z)
  and (e |= [] y   == e |- [] y)
  and (e |= <> y   == e |- <> y)
  and (e |= y U z  == e |- y U z)
  and (e |= o y    == e |- o y)

  and (e,t |= true   == t |= true(e))
  and (e,t |= false  == t |= false(e))
  and (e,t |= b      == t |= b(e))
  and (e,t |= c      == t |= c(e))
  and (e,t |= y /\ z == t |= (y /\ z)(e))
  and (e,t |= y ++ z == t |= (y ++ z)(e))
  and (e,t |= [] y   == t |= ([] y)(e))
  and (e,t |= <> y   == t |= (<> y)(e))
  and (e,t |= y U z  == t |= (y U z)(e))
  and (e,t |= o y    == t |= (o y)(e)) .
```

It took Maude 129 reductions to prove these lemmas. Therefore, one can safely add now these lemmas as follows:

```
fmod LEMMAS is
  protecting LTL .
  protecting LTL-REVISED .
  var E : Event .
  var T : Trace .  var X : Formula .
  eq E |= X = E |- X .
  eq E,T |= X = T |= X(E) .
endfm
```

We can now prove the theorem, by induction on traces. More precisely, we show:

$\mathcal{P}(\mathtt{E})$, and
$\mathcal{P}(\mathtt{T})$ implies $\mathcal{P}(\mathtt{E}, \mathtt{T})$, for all events `E` and traces `T`,

where $\mathcal{P}(\mathtt{T})$ is the predicate "for all formulas `X`, `T |= X` iff `T |- X`". This induction schema can be easily formalized in Maude as follows:

```
fmod PROOF-OF-THEOREM is protecting LEMMAS .
  op e : -> Event .
  op t : -> Trace .  op x : -> Formula .
  var X : Formula .
  eq t |= X = t |- X .
endfm

red e   |= x == e   |- x .
red e,t |= x == e,t |- x .
```

Notice the difference in role between the constant x and the variable X. The first reduction proves the base case of the induction, using the theorem of constants for the universally quantified variable X. In order to prove the induction step, we first applied the theorem of constants for the universally quantified variables E and T, then added $\mathcal{P}(t)$ to the hypothesis (the equation "eq t |= X = t |- X ."), and then reduced $\mathcal{P}(e\ t)$ using again the theorem of constants for the universally quantified variable X. Like in the proofs of the lemmas, we merged the two proofs to save space.

4.3 Further Optimization by Memoization

Even though the formula transforming algorithm in Subsection 4.1 can process 100 million events in about 25 minutes, which is relatively reasonable for practical purposes, it can be significantly improved by adding only 5 more characters to the existing Maude code presented so far. More precisely, one can replace the operation declaration

```
op _{_} : Formula Event* -> Formula [prec 10]
```

in module LOGICS-BASIC by the operation declaration

```
op _{_} : Formula Event* -> Formula [memo prec 10]
```

The attribute memo added to an operation declaration instructs Maude to memorize, or cache, the normal forms of terms rooted in that operation, i.e., those terms will be rewritten only once. Memoization is implemented by hashing, where the entry in the hash table is given by the term to be reduced and the value in the hash is its normal form. In our concrete example, memoization has the effect that any LTL formula will be transformed by a given event exactly once during the monitoring sequence; if the same formula and the same event occur in the future, the resulting modified formula is extracted from the hash table without applying any rewriting step. If one thinks of LTL in terms of automata, then our new algorithm corresponds to building the monitoring automaton *on the fly*. The obvious benefit of this technique is that only the *needed* part of the automaton is built, namely that part that is reachable during monitoring a particular sequence of events, which is practically very useful because the entire automaton associated to an LTL formula can be exponential in size, so storing it might become a problem.

The use of memoization brings a significant improvement in the case of LTL. For example, the same sequence of 100 million events, which took 1500 seconds using the algorithm presented in Subsection 4.1, takes only 185 seconds when one uses memoization, for a total of 2.2 rewritings per processed event and 540,000 events processed per second! We find this numbers amazingly good for any practical purpose we can think of and believe that, taking into account the simplicity, obvious correctness and elegance of the rewriting based algorithm (implemented basically by 8 rewriting rules in LTL-REVISED), it would be hard to argue for any other implementation of LTL monitoring. One should, however, be careful when one uses memoization because hashing slows down the rewriting engine. LTL is a happy case where memoization brings a significant improvement, but there might be monitoring logics where memoization could be less efficient. Experimentation is certainly needed if one designs a new logic for monitoring and wants to use memoization.

4.4 Synchronous versus Asynchronous Monitoring

There are many safety critical applications in which one would want to report a violation of a requirement as soon as possible, and to not allow the monitored program take any further action once a requirement is violated. We call this desired functionality *synchronous monitoring*. Otherwise, if a violation can only be detected after the monitored program is stopped and its entire execution trace is needed to perform the analysis, then we call it *asynchronous monitoring*. Notice that the dynamic programming algorithm in [28]

(see also Subsection 3.3) is *not* synchronous, because one can detect a violation only after a program is stopped and its execution trace is available for backwards traversal.

The algorithm presented in this section is also asynchronous because there are universally false formulae which are detected so only at the end of execution trace. Consider, for example, that one monitors the finite trace LTL formula `!<>([]A \/ []!A)`, which is false because at the end of any execution trace `A` either holds or not. However, in order for a monitor to detect such violation, it should implement a validity checker for finite trace LTL, such as the one in Subsection 5.2, and call it on the current formula after each event is processed. Checking validity of a finite trace LTL formula is very expensive (we are not aware of any theoretical result stating its exact complexity, but we believe that it is PSACE-complete, like for standard infinte trace LTL). We are currently considering providing a fully synchronous LTL monitoring module within JPAX, at the expense of calling a validity checker after each event, and let the user of the system choose either synchronous or asynchronous monitoring.

There are, however, many practical LTL formulae for which violation can be detected synchronously by the presented formula trasnforming rewriting-based algorithm presented in this section. Consider for example the sample formula of this paper, `[](green -> !red U yellow)`, which is violated if and only if a red event is observed after a green one. The monitoring requirement of our algorithm, which initially is the formula itself, will not be changed unless a green event is received, in which case it will change to `(!red U yellow) \/ [](green -> !red U yellow)`. A yellow event will turn it back into the initial formula, a green event will keep it unchanged, but a red event will turn it into `false`. If this is the case, then the monitor declares the formula violated and appropriate actions can be taken. Notice that the violation was detected *exactly* when it occured, and in general, that this formula can be monitored by our algorithm synchronously. A very interesting, practical and challanging problem is to find criteria that say when a formula can be synchronously monitored without the use of a validity checker.

5 Generating Efficient Monitors

Even though the rewriting based monitoring algorithm presented in the previous section performs quite well in practice, there can be situations in which one wants to minimize the monitoring overhead as much as possible. Additionally, despite its simplicity and elegance, the procedure above requires an efficient AC rewriting engine which may not be available or may not be desirable on some monitoring platforms, such as, for example, within an embedded system. In this section we present an algorithm built on the ideas in the previous section, also based on rewriting, which takes as input an LTL formula and generates a special finite state machine (FSM), called *binary transition tree finite state machine (BTT-FSM)*, that can be then used as an efficient monitor. An example of a BTT-FSM for the traffic light control requirements formula `[](green -> !red U yellow)` discussed previously in the paper can be seen in Figure 2 (see Figure 3 for a more formal representation). One should think of transitions using BTTs as naturally as possible; for example, if the BTT-FSM in Figure 2 is in state 1 and a non-terminal event is received, then: first evaluate the predicate `yellow`; if true then stay in state 1 else evaluate `green`; if false then stay in state 1 else evaluate `red`; if true then report "formula violated" else move to state 2. These FSMs can be either stored as data structures or generated as source code (case statements) which are further compiled into actual monitors. Our FSMs can be exponential in the number of states (as function of the size of the initial LTL formula) but they only need to evaluate *at most* all the atomic state predicates in order to proceed to the next state when a new event is received, so the runtime overhead is actually linear at worst. The size of our FSMs can become a problem when storage is a scarce resource, so we pay special attention to generating *optimal* FSMs. Interestingly, the number of state predicates to be evaluated tends to decrease with the number of states, so the overall monitoring overhead is also reduced. The drawback of generating an optimal BTT-FSM statically, i.e., before monitoring, is the exponential time/space required at startup (compilation). Therefore,

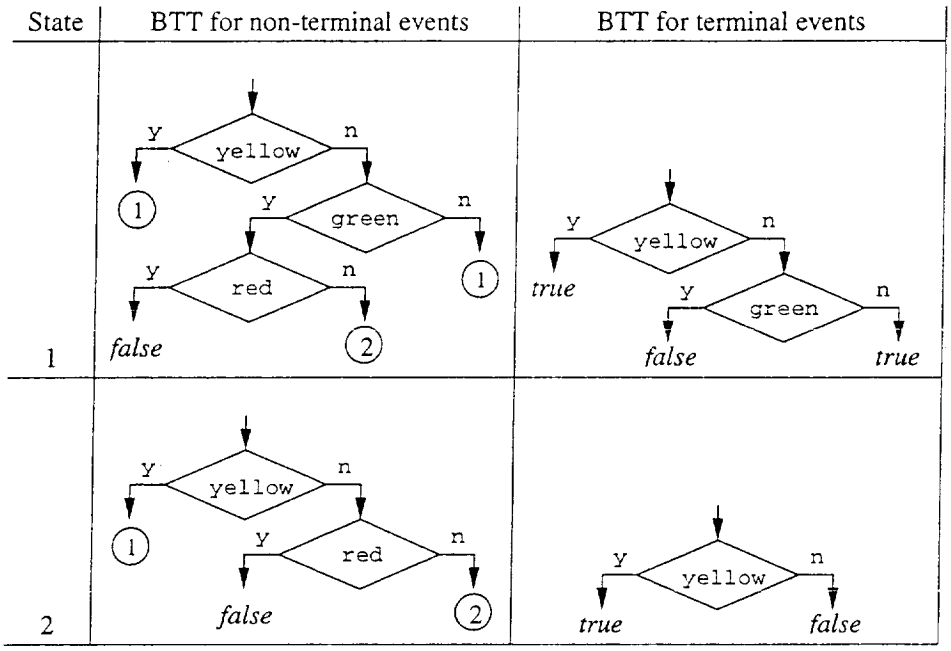

Figure 2: A BTT-FSM for the formula `[] (green -> !red U yellow)`

we recommend the algorithm below in situations where the LTL formulae to monitor are relatively small in size but the runtime overhead is desired to be minimal.

5.1 The Main Algorithm

Informally, our algorithm to generate minimal FSMs from LTL formulae uses the rewriting based algorithm presented in the previous section statically on all possible events, until the set of formulae to which the initial LTL formula can "evolve" stabilizes. More precisely, it builds a FSM whose states are formulae and whose transitions are the "events"; events are regarded here as boolean formulae describing the next (abstract) state of the monitored program in terms of the atomic predicates, i.e., formulae of the form $a_1 \wedge ... \wedge a_{k_1} \wedge \neg a_{k_1+1} \wedge ... \wedge \neg a_k$ where $a_1, ..., a_k$ are all the atomic state predicates occurring in the LTL formula; we let $s_1, ..., s_{2^k}$ denote all these state boolean formulae in no particular order. Since the size of any formula into which an LTL formula may evolve during an execution sequence is bounded (exponentially in the size of the initial formula; see the previous section), it follows that this algorithm will eventually terminate.

We use the notation $\varphi[p_1?\varphi_1, ..., p_j?\varphi_j]$ to state that the formula (or state in the FSM) φ changes to φ_i when an event satisfying the boolean formula p_i is received, for each i between 1 and some j smaller than or equal to 2^k. For a given newly generated formula/state φ, we generate the next potential formulae $\varphi_1, ..., \varphi_{2^k}$ by generating and analyzing $\varphi[s_1?\varphi_1, ..., s_{2^k}?\varphi_{2^k}]$, where φ_i is the formula $\varphi\{s_i\}$ obtained from φ after consuming the event s_i as described in the previous section, for each $i \in \{1, ..., 2^k\}$. To keep the number of states minimal, we rewrite $\varphi[..., p_{i_1}?\varphi_{i_1}, ..., p_{i_2}?\varphi_{i_2}, ...]$ to $\varphi[..., (p_{i_1} \vee p_{i_2})?\varphi_{i_1}, ...]$ whenever φ_{i_1} and φ_{i_2} are *equivalent* formulae under finite trace LTL semantics. In order to check this semantical equivalence we have implemented a validity checker that we call on the formula $\varphi_{i_1} \leftrightarrow \varphi_{i_2}$. Once the term $\varphi[p_1?\varphi_1, ..., p_j?\varphi_j]$ cannot be further simplified using the validity checker, the process of generating new formulae/states is iterated sequentially for the formulae $\varphi_1, ..., \varphi_j$. This procedure will terminate. Notice that each formula/state and its transitions thus generated, say $\varphi[p_1?\varphi_1, ..., p_j?\varphi_j]$, has the property that

exactly one of $p_1, ..., p_j$ is true for any newly received event, that *at most* one of $\varphi_1, ..., \varphi_j$ can be φ, *true* or *false*, and that if φ is *true* or *false* then $j = 1, p_1 = true$ and $\varphi_1 = \varphi$.

Obviously, this algorithm is exponential in the size of the initial LTL formula, so one should not expect it to terminate in reasonable time on large formulae. Once it terminates, the terms $\varphi[...]$ contain all those formulae φ to which the initial LTL formula can evolve during any monitoring session. What is left to do now is to say what happens when a monitoring session is terminated; according to our semantics for finite trace LTL described in Section 3, we assume that the last event is repeated infinitely. To faithfully implement this semantics, we extend the notation $\varphi[...]$ to $\varphi[...][s_1?\varphi'_1, ..., s_{2^k}?\varphi'_{2^k}]$, where φ'_i is the truth value $\varphi\{s_i \star\}$ obtained from φ considering that the event s_i is the last monitored event, as described in the previous section, for each $i \in \{1, ..., 2^k\}$. Then the same simplification technique as in the first step of the algorithm is used, reducing these terms to terms of the form $\varphi[...][p_t?true, p_f?false]$. Note that this second step takes much less than the previous step because validity is checked only for trivial formulae, and that *exactly* one of the boolean formulae p_t, p_f holds for any event claimed to be the last; since $p_f = \neg p_t$, we ignore p_f in what follows.

The two steps above therefore generate a special FSM encoded via terms $\varphi[p_1?\varphi_1, ..., p_j?\varphi_j][p_t?true]$. At monitoring time, this FSM is used as follows: if it is in state φ and a non-terminal event is received that makes p_i true (for $1 \leq i \leq j$) then the FSM moves into state φ_i; if a terminal event is received then either *true* or *false* is output, depending on whether p_t or p_f holds, respectively. Notice that the transitions for the nodes *true* and/or *false* are *true*[*true*?*true*][*true*?*true*] and *false*[*true*?*false*][*true*?*false*], respectively, so once the FSM gets into a *true* or *false* state it will remain there forever and it will return the appropriate output when the monitoring is stopped (one can slightly improve our algorithm by removing the *true*/*false* states and giving appropriate messages *before* entering them). The main computational component at *runtime* of our algorithm is therefore to evaluate the state formulae $p_1, ..., p_j$ which are boolean combinations of the atomic state propositions $a_1, ..., a_k$ occurring in the monitored LTL formula. Since some of these atomic propositions can be expensive to evaluate (for example one can question if an array is sorted), one would like to also minimize the amount of work needed to decide to which of $\varphi_1, ..., \varphi_j$ to go from φ. To achieve this, we have also implemented a method that takes a *generalized transition* $p_1?\varphi_1, ..., p_j?\varphi_j$ and generates a *binary transition tree (BTT)*, that is, a binary tree whose nodes are atomic predicates and whose leaves are formulae/states (Figure 2 shows some BTTs in flowchart notation). BTTs encode the transitions in our FSMs, reason for which we call our FSMs *binary transition tree finite state machines (BTT-FSMs)*. We actually generate *optimal* BTTs, where our optimality criterion at this stage is the size of the BTT.

Once the steps above terminate, the formulae φ, φ_1, etc., are not needed anymore, so we replace them by unique labels in order to reduce the amount of storage needed to encode the BTT-FSM. This algorithm can be relatively easily implemented in any programming language. We have, however, found Maude again a very elegant system, implementing this whole algorithm in about 200 lines of Maude code. We next describe our implementation in more detail.

5.2 The Validity Checker

Our implementation for the validity checker is simple and exponential. It follows closely the implementation of the main algorithm described in the previous subsection, except, of course, the minimization step that uses the validity checker; a boolean validity checker is used instead to keep the number of generated formulae small. For a given LTL formula, we therefore obtain a FSM also given by terms of the form $\varphi[p_1?\varphi_1, ..., p_j?\varphi_j][p_t?true]$; this FSM will almost certainly have more states than the minimal BTT-FSM described above, but it can be generated faster (because the expensive validity checks for formulae $\varphi_{i_1} \leftrightarrow \varphi_{i_2}$ are not needed anymore). Then we just check that for any state in this FSM it is the case that $p_t = true$. The intuition for this check is that a formula is valid under finite trace LTL semantics if and

only if any (finite) monitoring sequence satisfies that formula; since any generated formula/state in the FSM corresponds to a valid formula to which the initial formula can evolve, we need to make sure that each of these formulae become *true* under any possible last monitored event.

5.3 A Minimal FSM

As mentioned earlier, minimality of the monitor is not crucial for obtaining a small runtime overhead, in the same way in which a smaller program is not necessarily faster than a larger program. However, it seems to be an interdependence between the size of a FSM monitor and the size of the BTTs associated to its states, namely the smaller the FSM in number of states the smaller the BTTs, so the lower the monitoring overhead. A rigorous proof of this interdependence seems to be hard; we only show that our FSMs are minimal with respect to the number of states.

Theorem: The special FSM presented in this section has minimal number of states in the class of deterministic (total) FSMs in which transitions are activated by any boolean combination of atomic state predicates.

Proof: It follows by standard results in the theory of deterministic finite state automata (DFA), noticing that our rewriting decision procedure for propositional calculus gives a canonical form for any transition (i.e., boolean formula) and that the usual language equivalence in DFA's becomes trace equivalence in our special FSMs, which is nothing but semantical equivalence of formulae in our finite trace LTL setting. The rest follows by the fact that φ and ψ are semantically equivalent if and only if $\varphi \leftrightarrow \psi$ is valid.

5.4 Binary Transition Trees

In order to keep the runtime overhead of our FSMs minimal, it is crucial to do as little computation as possible in order to proceed to the next state. When in a state $\varphi[p_1?\varphi_1, ..., p_j?\varphi_j][...]$, a naive implementation would start to sequentially evaluate p_1, ..., p_j on the new event until a first one, say p_i, holds and then move to state φ_i. Fortunately, one can do much better by making use of what we call *binary transition trees (BTTs)*, which are binary trees associated to each state of a FSM having atomic propositions as nodes and states in the FSM as leaves. We use the notation $\langle condition \rangle$? $\langle BTT \rangle$: $\langle BTT \rangle$ borrowed from C and Java to denote BTTs. For example, the BTT a_1 ? a_2 ? 1 : a_3 ? 2 : 1 : a_3 ? 3 : 2 says "eval a_1; if a_1 then (eval a_2; if a_2 then go to state 1 else (eval a_3; if a_3 then go to state 2 else go to state 1)) else (eval a_3; if a_3 then go to state 3 else go to state 2". When the transitions of a FSM are all encoded as BTTs (one associated to each state), then we call it a *binary transition tree finite state machine (BTT-FSM)*. Notice that, in the worst possible case, one just has to evaluate all the atomic predicates in order to proceed to the next state of a BTT-FSM. We have implemented a procedure that takes a generalized transition and returns a BTT. More BTTs can encode the same generalized transition, so one needs to develop some criteria to select the better ones. A most natural criterion would be to minimize the average amount of computation. For example, if all atomic predicates are equally probable to hold and if an atomic predicate is very expensive to evaluate, then one would select that BTT that places that predicate as deeply as possible, so its evaluation is delayed as much as possible. Based on the above, we believe that the following is an important theoretical problem in runtime monitoring:

> **Problem:** Optimal BTT
>
> **Input:** A set of atomic predicates $a_1, ..., a_k$ that hold with probabilities $\pi_1, ..., \pi_k$ and have costs $c_1, ..., c_k$, respectively, and a generalized transition $p_1?\varphi_1, ..., p_j?\varphi_j$ where $p_1, ..., p_j$ are boolean formulae over $a_1, ..., a_k$.

Output: A BTT encoding the generalized transition that probabilistically minimizes the amount of computation to decide the next state.

We do not know how to solve this interesting problem yet, but we conjecture the following result that we currently use in our implementation:

Conjecture: If $\pi_1 = ... = \pi_k$ and $c_1 = ... = c_k$ then the solution to the problem above is the BTT of minimal size.

Our current generalized transition to BTT algorithm is exponential in the number of atomic predicates; it simply enumerates all possible BTTs recursively and then selects the minimal one (in size). Generating the BTTs takes significantly less then generating the FSM, so we do not regard it as a problem yet.

5.5 Evaluation

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

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

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

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

The LTL formulae on which our algorithm has the worst performance are those containing many nested temporal operators (which are not frequently used in specifications anyway, because of the high risk of getting them wrong). For example, it takes our algorithm 2.5 seconds to generate the minimal 3-state (*true* and *false* states are not counted) BTT-FSM for the formula `a U (b U (c U d))` and 25.8 seconds to generate the 7-state minimal BTT-FSM for the formula `((a U b) U c) U d`. The generated BTT-FSMs are monitored most efficiently on RAM machines, due to the fact that case statements are usually implemented via jumps in memory. Monitoring BTT-FSMs using rewriting does not seem appropriate because it would require linear time (as a function of the number of states) to extract the BTT associated to a state in a BTT-FSM. An interesting experiment would be use memoization to avoid multiple searches for the same BTT. However, we believe that the algorithm presented in Section 4 is satisfactory in practice if one is willing to use a rewriting engine for monitoring.

6 Conclusions

We presented a finite trace semantics of LTL in the Maude logic together with a much more efficient version based on formula transforming state changes. We then went a step further and improved the efficient version by hashing rewriting results, thereby reducing the number of rewritings performed during trace analysis. The hashing corresponds to building an observer automaton on-the-fly, having the advantage that only the

part of the automaton that is needed for analyzing a given trace is generated. The resulting algorithm is very efficient. However, in certain cases one would want to generate an observer finite state machine (or automaton) apriori, for example if a rewriting system cannot be used for monitoring, or if minimal runtime overhead is needed by any means. For this case, we presented an algorithm for generating minimal automata from LTL formulae.

All algorithms are written in surprisingly few lines of Maude code, illustrating the strength of rewriting for this particular domain. In spite of the reduced size of the code, the implementations seem to be efficient for practical purposes. As a consequence, we have demonstrated how rewriting can be used not only to experiment with runtime monitoring logics, but also as an implementation language. As an example of future work, that we anticipate to be very easy, is the extension of LTL with real-time constraints. Since Maude by itself provides a high-level specification language, one can argue that Maude in its entirety can be used for writing requirements. Further work will show whether this avenue is fruitful.

References

[1] 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`.

[2] S. Cohen. Jtrek. Compaq, `http://www.compaq.com/java/download/jtrek`.

[3] 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*, Limerich, Ireland, June 2000. ACM Press.

[4] 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.

[5] 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.

[6] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.

[7] 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.

[8] J. Goguen, K. Lin, G. Roşu, A. Mori, and B. Warinschi. An Overview of the Tatami Project. In K. Futatsugi, T. Tamai, and A. Nakagawa, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, to appear, 2000.

[9] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

[10] K. Havelund, S. Johnson, and G. Roşu. Specification and Error Pattern Based Program Monitoring. In *Proceedings of the European Space Agency workshop on On-Board Autonomy*, Noordwijk, The Netherlands, Oct. 2001.

[11] K. Havelund, M. R. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, Aug. 2001. An earlier version occurred in the Proceedings of the 4th SPIN workshop, 1998, Paris, France.

[12] 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.

[13] K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS'01)*, Montreal, Canada, June 2001.

[14] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.

[15] 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.

[16] K. Havelund and G. Roşu. Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, RIACS, May 2001. Written 20 December 2000.

[17] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. EASST best paper award at ETAPS'02.

[18] 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.

[19] G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE'99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.

[20] J. Hsiang. *Refutational Theorem Proving using Term Rewriting Systems*. PhD thesis, University of Illinois at Champaign-Urbana, 1981.

[21] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[22] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.

[23] J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *Proceedings, WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[24] T. O'Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *In Proceedings of the California Software Symposium*, 1996.

[25] D. Y. Park, U. Stern, and D. L. Dill. Java Model Checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.

[26] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[27] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, 1992.

[28] G. Roşu and K. Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. RIACS Technical report, `http://ase.arc.nasa.gov/pax`, January 2001.

[29] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[30] 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.

[31] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, Sept. 2000.