Automated Software Engineering, 12, 151–197, 2005

Rewriting-Based Techniques for Runtime Verification

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

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

Abstract. Techniques for efficiently evaluating future time Linear Temporal Logic (abbreviated LTL) formulae on finite execution traces are presented. 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. A finite trace variant of LTL is formally defined, together with an immediate executable semantics which turns out to be quite inefficient if used directly, via rewriting, as a monitoring procedure. Then three algorithms are investigated. First, a simple synthesis algorithm for monitors based on dynamic programming is presented; despite the efficiency of the generated monitors, they unfortunately need to analyze the trace backwards, thus making them unusable in most practical situations. To circumvent this problem, two rewriting-based practical algorithms are further investigated, one using rewriting directly as a means for online monitoring, and the other using rewriting to generate automata-like monitors, called binary transition tree finite state machines (and abbreviated BTT-FSMs). Both rewriting algorithms are implemented in Maude, an executable specification language based on a very efficient implementation of term rewriting. The first rewriting algorithm essentially consists of a set of equations establishing an executable semantics of LTL, using a simple formula transforming approach. This algorithm is further improved to build automata on-the-fly via caching and reuse of rewrites (called memoization), resulting in a very efficient and small Maude program that can be used to monitor program executions. The second rewriting algorithm builds on the first one and synthesizes provably minimal BTT-FSMs from LTL formulae, which can then be used to analyze execution traces online without the need for a rewriting system. 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 logics for program monitoring.

Keywords: runtime analysis, rewriting, verification

1. Introduction and motivation

Future time Linear Temporal Logic, abbreviated LTL, was introduced by Pnueli in 1977 (Pnueli, 1977) (see also Manna and Pnueli, 1992, 1995) 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

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

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++ (Ball et al., 2001; Demartini et al., 1999; Holzmann and Smith, 1999; Corbett et al., 2000; Park et al., 2000; Godefroid, 1997; Stoller, 2000; Havelund and Pressburger, 2000; Visser et al., 2000). 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. In the context of highly reliable and/or safety critical systems, one would actually want to *monitor* a program execution during operation and to determine whether it conforms to its specification. Any violation of the specification can then be used to guide the execution of the program into a safe state, either manually or automatically. In this paper we describe a collection of algorithms for monitoring program executions against LTL formulae. It is demonstrated how term rewriting, and in particular the Maude rewriting system (Clavel et al., 1999), can be used to implement some of these algorithms very efficiently and conveniently.

The work presented in this paper has been started as part of, and stimulated by, the PATHEXPLORER project at NASA Ames, and in particular the Java PATHEXPLORER (JPAX) tool (Havelund and Roşu, 2001a, 2001b) for monitoring Java programs. JPAX facilitates automated instrumentation of Java byte-code, currently using Compaq's JTREK which is not public anymore, but soon using BCEL (Dahm,). The instrumented code emits relevant events to an observer during execution (see figure 1). The observer can be running a Maude (Clavel et al., 1999) process as a special case, so 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 modules, each module performing a particular analysis that has been requested. In addition to checking temporal logic requirements, modules have also been programmed to perform error pattern analysis of multi-threaded programs, predicting deadlock and datarace potentials.

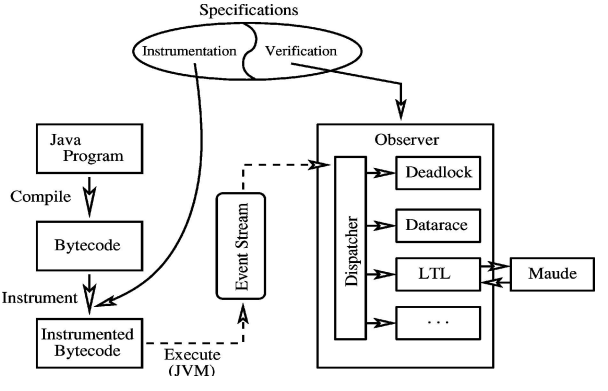

Figure 1. Overview of JPAX .

Using temporal logic in testing is an idea of broad practical and theoretical interest. One example is the commercial Temporal Rover and DBRover tools (Drusinsky, 2000, 2003), in which LTL properties are translated into code, which is then inserted at chosen positions in the program and executed whenever reached during program execution. The MaC tool (Lee et al., 1999; Kim et al., 2001) is another example of a runtime monitoring tool. Here, Java byte-code is automatically instrumented to generate events of interest during the execution. Of special interest is the temporal logic used in MaC, which can be classified as a past time interval logic convenient for expressing monitoring properties in a succinct way. All the systems above try to discharge the program execution events as soon as possible, in order to minimize the space requirements. In contrast, a technique is proposed in Kortenkamp et al. (2001) where the execution events are stored in an SQL database at runtime and then analyzed by means of queries after the program terminates. The PET tool, described in Gunter and Peled (2000, 2002), Gunter et al. (2003), uses a future time temporal logic formula to guide the execution of a program for debugging purposes. Java MultiPathExplorer (Sen et al., 2003) is a tool which checks a past time LTL safety formula against a partial order extracted online from an execution trace. POTA (Sen and Garg, 2003) is another partial order trace analyzer system. Java-MoP (Chen and Roşu, 2003) is a generic logic monitoring tool encouraging "monitoring-oriented programming" as a paradigm merging specification and implementation. Complexity results for testing a finite trace against temporal formulae expressed in different temporal logics are investigated in Markey and Schnoebelen (2003). Algorithms using alternating automata to monitor LTL properties are proposed in Finkbeiner and Sipma (2001), and a specialized LTL collecting statistics along the execution trace is described in Finkbeiner et al. (2002). Various algorithms to generate testing automata from temporal logic formulae are discussed in Richardson et al. (1992), O'Malley et al. (1996), and Giannakopoulou and Havelund (2001) presents a Büchi automata inspired algorithm adapted to finite trace LTL.

The major goal of this paper is to present rewriting-based algorithms for effectively and efficiently evaluating LTL formulae on finite execution traces *online*, that is, by processing each event as it arrives. An important contribution of 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 allows one to formalize ideas in a framework close to standard mathematics. The presented algorithms are considered in the context of JPAX, but they can be easily adapted and used within other monitoring frameworks. We claim that the techniques presented in this paper, even though applied to LTL, are in fact generic and can be easily applied to other logics for monitoring. For example, in Roşu and Viswanathan (2003), Sen and Roşu (2003) we applied the same generic, "formula transforming", techniques to obtain rewriting based algorithms for situations in which the logic for monitoring was replaced by extended regular expressions (regular expressions with complement).

A non-trivial application of the rewriting based techniques presented in this paper is X9, a test-case generation and monitoring environment for a software system that controls the planetary NASA rover K9. This collaborative effort is described in more detail in Artho et al. (2003) and it will be presented in full detail elsewhere soon. The rover controller, programmed in 35,000 lines of C++, essentially executes plans, where a plan is a tree-like

structure consisting of actions and sub-actions. The leaf actions control various hardware on the rover, such as for example the camera and the wheels. The execution of a plan must cause the actions to be executed in the right order and must satisfy various time constraints, also part of the plan. Actions can start and eventually either terminate successfully or fail. Plans can specify how failed sub-actions can propagate upwards.

Testing the rover controller consists of generating plans and then monitoring that the plan actions are executed in the right order and that failures are propagated correctly. X9 automatically generates plans from a "grammar" of the structure of plans, using the Java PathFinder model checker (Visser et al., 2000). For each plan, a set of temporal formulae that an execution of the plan must satisfy is also generated. For example, a plan may state that an action a should be executed by first executing a sub-action a_1 and then a sub-action a_2, and that the failure of any of the sub-actions should not propagate: action a should eventually succeed, regardless of whether a_1 or a_2 fails. The generated temporal formulae will state these requirements, such as for example `[](start(a) -> <>succeed(a))` saying that "it is always the case (`[]`) that when action `a` starts, then eventually (`<>`) it terminates successfully", and execution traces are monitored against them.

X9 is currently being turned into a mature system to be used by the developer. It is completely automated, generating a web-page containing all the warnings found. The top-level web-page identifies all the test-cases that have failed (by violating some of the temporal properties), each linked to a web-page containing specifics such as the plan, the execution trace, and the properties that are violated. X9 has itself been tested by seeding errors into the rover controller code. The automated monitoring relieves the programmer from manually analyzing printed execution traces. Extending the logic with real-time, as is planned in future work, is crucial for this application since plans are heavily annotated with time constraints.

In Section 2, based on our experience, we give a rough classification of monitoring and runtime analysis algorithms by considering three important criteria. A first criterion is whether the execution trace of the monitored or analyzed program needs to be stored or not. Storing a trace might be very useful for specific types of analysis because one could have random access to events, but storing an execution trace is an expensive operation in practice, so sometimes trace-storing algorithms may not be desirable. A second criterion regards the synchronicity of the monitor, more precisely whether the monitor is able to react as soon as the specification or the requirement has been violated. Synchronicity may often trigger running a validity checker for the logic under consideration, which is typically a very expensive task. Finally, monitoring and analysis algorithms can also be classified as "predictive" versus "exact", where the "exact" ones monitor the observed execution trace as a flat list of events, while the predictive algorithms try to guess potential erroneous behaviors of programs that can occur under different executions. All the algorithms in this paper are exact.

This paper requires a certain amount of mathematical notions and notations, which we introduce in Section 3 together with Maude (Clavel et al., 1999), a high-performance system supporting both membership equational logic (Meseguer, 1998) and rewriting logic (Meseguer, 1992). The current version of Maude can do more than 3 million rewritings per second on standard PCs, and its compiled version is intended to support more than 15 million rewritings per second[1], so it can quite well be used as an implementation language.

Section 4 defines the finite trace variant of linear temporal logic that we use in the rest of the paper. We found, by carefully analyzing several practical examples, that the most appropriate assumption to make at the end of the trace is that it is stationary in the last state. Then we define the semantics of the temporal operators using their usual meaning in infinite trace LTL, where the finite trace is infinitely extended by repeating the last state. Another option would be to consider that all atomic predicates are false or true in the state following the last one, but this would be problematic when inter-dependent predicates are involved, such as "gate-up" and "gate-down".

In previous work we described a technique which synthesizes efficient dynamic programming algorithms for checking LTL formulae on finite execution traces (Roşu and Havelund, 2001). Even though this algorithm is not dependent on rewriting (but it could be easily implemented in Maude by rewriting as we did with its dual variant for past time LTL (Havelund and Roşu, (to appear); Chen and Roşu, 2003)), for the sake of completeness we present it in some detail in Section 5. 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 because it is both asynchronous and trace-storing. in Havelund and Roşu, Havelund et al., Havelund and Roşu (2002, 2001, (to appear)) we dualize this technique and apply it to past time LTL, in which case the trace more naturally can be examined in a forwards direction synchronously.

Section 6 presents our first practical rewriting-based algorithm, which can directly monitor an LTL formula. This algorithm originates in Havelund and Roşu, Roşu and Havelund (2001, 2001) and it was partially presented at the Automated Software Engineering conference (Havelund and Roşu, 2001). The algorithm is expressed as a set of equations establishing an executable semantics of LTL using a simple formula transforming approach. The idea is to rewrite or transform an LTL monitoring requirement formula φ when an event e is received, to a formula $\varphi\{e\}$, which represents the new requirement that the monitored system should fulfill for the remaining part of the trace. This way, the LTL formula to monitor "evolves" into other LTL formulae by subsequent transformations. We show, however, that the size of the evolving formula is in the worst-case exponentially bounded by the size of the original LTL formula, and also that an exponential space explosion cannot be avoided in certain unfortunate cases. The efficiency of this rewriting algorithm can be improved by almost an order of magnitude by caching and reusing rewrites (a.k.a. "memoization"), which is supported by Maude. This algorithm is often synchronous, though there are situations in which it misses reporting a violation at the exact event when it occurs. The violation is, however, detected at a subsequent event. This algorithm can be relatively easily transformed into a synchronous one if one is willing to pay the price of running a validity checker, like the one presented in Section 7.3, after processing each event. The practical result of Section 6 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 in monitoring.

We finally present an alternative solution to monitoring LTL in Section 7, where a rewriting-based algorithm is used to *generate* an optimal special observer from an LTL formula. By optimality is meant everything one may expect, such as minimal number of states, forwards traversal of execution traces, synchronicity, efficiency, but also less standard

optimality features, such as transiting from one state to another with a minimum amount of computation. In order to effectively do this we introduce the notion of *binary transition tree* (BTT), as a generalization of binary decision diagrams (BDD) (Bryant, 1986), whose purpose is to provide an *optimal order* in which state predicates need to be evaluated to decide the next state. The motivation for this is that in practical applications evaluating a state predicate is a time consuming task, such as for example to check whether a vector is sorted. The associated finite state machines are called *binary transition tree finite state machines* (BTT-FSM). BTT-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 BTT-FSM generator, which includes a validity checker, is also implemented in Maude and has about 200 lines of code in total.

2. A taxonomy of runtime analysis techniques

A *runtime analysis technique* is regarded in a broad sense in this section; it can be a method or a concrete algorithm that analyzes the execution trace of a running program and concludes a certain property about that program. Runtime analysis algorithms can be arbitrarily complex, depending upon the kind of properties to be monitored or analyzed. Based on our experience with current procedures implemented in JPaX, in this section we make an attempt to classify runtime analysis techniques. The three criteria below are intended to be neither exhaustive nor always applicable, but we found them quite useful in practice. They are not specific to any particular logic or approach, so we present them before we introduce our logic and algorithms. In fact, this taxonomy will allow us to appropriately discuss the benefits and drawbacks of our algorithms presented in the rest of the paper.

2.1. Trace storing versus non-storing algorithms

As events are received from the monitored system, a runtime analysis algorithm typically maintains a state which allows it to reason about the monitored execution trace. Ideally, the amount of information needed to be stored by the monitor in its state depends only upon the property to be monitored and *not* upon the number of already processed events. This is desired because, due to the huge amount of events that can be generated during a monitoring session, one would want one's monitoring algorithms to work in linear time with the number of events processed.

There are, however, situations where it is not possible or practically feasible to use storage whose size is a function of only the monitoring requirement. One example is that of monitoring *extended regular expressions* (ERE), i.e., regular expressions enriched with a complement operator. As shown by the success of scripting languages like PERL or PYTHON, software developers tend to understand and like regular expressions and feel comfortable to describe patterns using those, so ERE is a good candidate formalism to specify monitoring requirements (we limit ourselves to only patterns described via temporal logics in this paper though).

It is however known that ERE to automata translation algorithms suffer from a non-elementary state explosion problem, because a complement operation requires

nondeterministic-to-deterministic automata conversions, which yield exponential blowups in the number of states. Since complement operations can be nested, generating automata from EREs may often not be feasible in practice. Fortunately, there are algorithms which avoid this state explosion problem, at the expense of having to store the execution trace and then, at the end of the monitoring session, to analyze it by traversing it forwards and backwards many times. The interested reader is referred to Hopcroft and Ullman (1979) for a $O(n^3m)$ dynamic programming algorithm (n is the length of the execution trace and m is the size of the ERE), and to Yamamoto (2000),Kupferman and Zuhovitzky (2002) for $O(n^2m)$ non-trivial algorithms.

Based on these observations, we propose a first criterion to classify monitoring algorithms, namely on whether they *store or do not store the execution trace*. In the case of EREs, trace storing algorithms are polynomial in the size of the trace and linear in the ERE requirement, while the non-storing ones are linear in the size of the trace and highly exponential in the size of the requirement. In this paper we show that trace storing algorithms for linear temporal logic can be linear in both the trace and the requirement (see Section 5), while trace non-storing ones are linear in the size of the trace but simply exponential in the size of the requirement.

Trace non-storing algorithms are apparently preferred, but, however, their size can be so big that it could make their use unamenable in certain important situations. One should carefully analyze the trade-offs in order to make the best choice in a particular situation.

2.2. *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 to 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 executes several more steps or after it is stopped and its entire execution trace is needed to perform the analysis, then we call it *asynchronous monitoring*.

The dynamic programming algorithm presented in Section 5 is *not* synchronous, because one can detect a violation only after the program is stopped and its execution trace is available for backwards traversal. The algorithm presented in Section 6 is also asynchronous in general because there are universally false formulae which are detected so only at the end of an execution trace or only after several other execution steps. 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, or the formula `ooA /\ oo!A`, which is also false but will be detected so only after two more events. However, the rewriting algorithm in Section 6 is synchronous in many practical situations. The algorithm in Section 7 is always synchronous, though one should be aware of the fact that its size may become a problem on large formulae.

In order for an LTL monitor to be synchronous, it needs to implement a validity checker for finite trace LTL, such as the one in Section 7.3 (figure 6), 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 PSPACE-complete, like for standard infinite trace LTL (Sistla and Clarke, 1985)). 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 formula transforming rewriting-based algorithm presented in Section 6. 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 occurred. A very interesting, practical and challenging problem is to find criteria that say when a formula can be synchronously monitored without the use of a validity checker.

2.3. Predictive versus exact analysis

An increasingly important class of runtime analysis algorithms are concerned with *predicting* anomalies in programs from *successful* observed executions. One such algorithm can be easily obtained by slightly modifying the *wait-for-graph* algorithm, which is typically used to *detect* when a system is in a deadlock state, to make it predict deadlocks. One way to do this is to *not* remove synchronization objects from the wait-for-graph when threads/processes release them. Then even though a system is not deadlock, a warning can be reported to users if a cycle is found in the wait-for-graph, because that represents a *potential* of a deadlock.

Another algorithm falling into the same category is Eraser (Savage et al., 1997), a datarace prediction procedure. For each shared memory region, Eraser maintains a set of *active locks* which protect it, which is intersected with the set of locks held by any accessing thread. If the set of active locks ever becomes empty then a warning is issued to the user, with the meaning that a potential unprotected access can take place. Both the deadlock and the datarace predictive algorithms are very successful in practice because they scale well and find many of the errors they are designed for. We have also implemented improved versions of these algorithms in Java PathExplorer.

We are currently also investigating predictive analysis of safety properties expressed using past time temporal logic, and a prototype system called Java MultiPathExplorer is being implemented (Sen et al., 2003). The main idea here is to *instrument* Java classes to emit events timestamped by vector clocks (Fidge, 1988), thus enabling the observer to extract a *partial order* reflecting the causal dependency on the memory accesses of the multithreaded program. If any linearization of that inferred partial order leads to a violation of the safety property then a warning is generated to the user, with the meaning that there can be executions of the multithreaded program, including the current one, which violate the requirements.

In this paper we restrict ourselves to only *exact* analysis of execution traces. That means that the events in the trace are supposed to have occurred exactly in the received order (this can be easily enforced by maintaining a logical clock, then timestamping each event with the current clock, and then delivering the messages in increasing order of timestamps), and that we only check whether that particular order violates the monitoring requirements or not. Techniques for predicting future time LTL violations will be investigated elsewhere soon.

Although the taxonomy discussed in this section is intended to only be applied to tools, the problem domain may also admit a similar taxonomy. While such a taxonomy seems to be hard to accomplish in general, it would certainly be very useful because it would allow one to choose the proper runtime analysis technique for a given system and set of properties. However, like this paper shows, it is often the case that one can choose among several types of runtime analysis techniques for a given problem domain.

3. Preliminaries

In this section we recall notions and notations that will be used in the paper, including membership equational logic, term rewriting, Maude notation, and (infinite trace) linear temporal logics.

3.1. Membership equational logic

Membership equational logic (MEL) extends many- and order-sorted equational logic by allowing memberships of terms to sorts in addition to the usual equational sentences. We only recall those MEL notions which are necessary for understanding this paper; the interested reader is referred to Meseguer (1998), Bouhoula et al. (2000) for a comprehensive exposition of MEL.

3.1.1. Basic definition. A *many-kinded algebraic signature* (K, Σ) consists of a set K and a $(K^\star \times K)$-indexed set $\Sigma = \{\Sigma_{k_1 k_2 \ldots k_n, k} \mid k_1, k_2, \ldots, k_n, k \in K\}$ of operations, where an operation $\sigma \in \Sigma_{k_1 k_2 \ldots k_n, k}$ is written $\sigma : k_1 k_2 \ldots k_n \to k$. A *membership signature* Ω is a triple (K, Σ, π) where K is a set of *kinds*, Σ is a K-kinded algebraic signature, and $\pi : S \to K$ is a function that assigns to each element in its domain, called a *sort*, a kind. Therefore, sorts are grouped according to kinds and operations are defined on kinds. For simplicity, we will call a "membership signature" just a "signature" whenever there is no confusion.

For a *many-kinded signature* (K, Σ), a Σ-algebra A consists of a K-indexed set $\{A_k \mid k \in K\}$ together with interpretations of operations $\sigma : k_1 k_2 \ldots k_n \to k$ into functions $A_\sigma : A_{k_1} \times A_{k_2} \times \cdots \times A_{k_n} \to A_k$. For any given signature $\Omega = (K, \Sigma, \pi)$, an Ω-*membership algebra* A is a Σ-algebra together with a set $A_s \subseteq A_{\pi(s)}$ for each sort $s \in S$. A particular algebra, called *term algebra*, is of special interest. Given a K-kinded signature Σ and a K-indexed set of *variables* X, let $T_\Sigma(X)$ be the algebra of Σ-*terms over variables in* X extending X iteratively as follows: if $\sigma : k_1 k_2 \ldots k_n \to k$ and $t_1 \in T_{\Sigma,k_1}(X)$, $t_2 \in T_{\Sigma,k_2}(X)$, $\ldots, t_n \in T_{\Sigma,k_n}(X)$, then $\sigma(t_1, t_2, \ldots, t_n) \in T_{\Sigma,k}(X)$.

Given a signature Ω and a K-indexed set of variables X, an *atomic* (Ω, X)-*equation* has the form $t = t'$, where $t, t' \in T_{\Sigma,k}(X)$, and an *atomic* (Ω, X)-*membership* has the form $t : s$, where s is a sort and $t \in T_{\Sigma,\pi(s)}(X)$. An Ω-*sentence* in MEL has the form $(\forall X)\ a$ `if` $a_1 \wedge \ldots \wedge a_n$, where $a, a_1, \ldots, a_n$ are atomic (Ω, X)-equations or (Ω, X)-memberships, and $\{a_1, \ldots, a_n\}$ is a set (no duplications). If $n = 0$, then the Ω-sentence is called *unconditional* and written $(\forall X)\ a$. Equations are called *rewriting rules* when they are used only from left to right, as it will happen in this paper.

Given an Ω-algebra A and a K-kinded map $\theta\colon X \to A$, then $A, \theta \models_\Omega t = t'$ iff $\theta(t) = \theta(t')$, and $A, \theta \models_\Omega t : s$ iff $\theta(t) \in A_s$. A *satisfies* $(\forall X)\ a$ `if` $a_1 \wedge \ldots \wedge a_n$, written $A \models_\Omega (\forall X)\ a$ `if` $a_1 \wedge \ldots \wedge a_n$, iff for each $\theta\colon X \to A$, if $A, \theta \models_\Omega a_1$ and ... and $A, \theta \models_\Omega a_n$, then $A, \theta \models_\Omega a$.

An Ω-*specification* (or Ω-*theory*) $T = (\Omega, E)$ in MEL consists of a signature Ω and a set E of Ω-sentences. An Ω-algebra A *satisfies* (or *is a model of*) $T = (\Omega, E)$, written $A \models T$, iff it satisfies each sentence in E.

3.1.2. Inference rules. MEL admits complete deduction (see (Meseguer, 1998), where the rule of congruence is stated in a somewhat different but equivalent way). In the congruence rule below, $\sigma \in \Sigma_{k_1 \ldots k_i, k}$, W is a set of variables $w_1 : k_1, \ldots, w_{i-1} : k_{i-1}, w_{i+1} : k_{i+1}, \ldots, w_n : k_n$, and $\sigma(W, t)$ is a shorthand for the term $\sigma(w_1, \ldots, w_{i-1}, t, w_{i+1}, \ldots, w_n)$):

$$\text{Reflexivity}: \frac{}{E \vdash_\Omega (\forall X)\ t = t} \tag{1}$$

$$\text{Symmetry}: \frac{E \vdash_\Omega (\forall X)\ t = t'}{E \vdash_\Omega (\forall X)\ t' = t} \tag{2}$$

$$\text{Transitivity}: \frac{E \vdash_\Omega (\forall X)\ t = t',\ E \vdash_\Omega (\forall X)\ t' = t''}{E \vdash_\Omega (\forall X)\ t = t''} \tag{3}$$

$$\text{Congruence}: \frac{E \vdash_\Omega (\forall X)\ t = t'}{E \vdash_\Omega (\forall X, W)\ \sigma(W, t) = \sigma(W, t'),\ \text{for each } \sigma \in \Sigma} \tag{4}$$

$$\text{Membership}: \frac{E \vdash_\Omega (\forall X)\ t = t',\ E \vdash_\Omega (\forall X)\ t : s}{E \vdash_\Omega (\forall X)\ t' : s} \tag{5}$$

$$\text{Modus Ponens}: \frac{\begin{array}{l}\text{Given a sentence in } E \\ \quad (\forall Y)\ t = t' \ \texttt{if}\ t_1 = t_1' \wedge \ldots \wedge t_n = t_n' \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m \\ (\text{resp. } (\forall Y)\ t : s \ \texttt{if}\ t_1 = t_1' \wedge \ldots \wedge t_n = t_n' \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m) \\ \text{and } \theta\colon Y \to T_\Sigma(X) \text{ s.t. for all } i \in \{1, \ldots, n\} \text{ and } j \in \{1, \ldots, m\} \\ \quad E \vdash_\Omega (\forall X)\ \theta(t_i) = \theta(t_i'),\ E \vdash_\Omega (\forall X)\ \theta(w_j) : s_j\end{array}}{E \vdash_\Omega (\forall X)\ \theta(t) = \theta(t') \quad (\text{resp. } E \vdash_\Omega (\forall X)\ \theta(t) : s)} \tag{6}$$

The rules above can therefore prove any unconditional equation or membership that is true in all membership algebras satisfying E. In order to derive conditional statements, we will therefore consider the standard technique adapting the "deduction theorem" to equational logics, namely deriving the conclusion of the sentence after adding the condition as an axiom; in order for this procedure to be correct, the variables used in the conclusion

need to be first transformed into constants. All variables can be transformed into constants, so we only consider the following simplified rules:

$$\text{Theorem of Constants: } \frac{E \vdash_{\Omega \cup X} (\forall \emptyset)\ a \ \texttt{if}\ a_1 \wedge \ldots \wedge a_n}{E \vdash_{\Omega} (\forall X)\ a \ \texttt{if}\ a_1 \wedge \ldots \wedge a_n} \tag{7}$$

$$\text{Implication Elimination: } \frac{E \cup \{a_1, \ldots, a_n\} \vdash_{\Omega} (\forall \emptyset)\ a}{E \vdash_{\Omega} (\forall \emptyset)\ a \ \texttt{if}\ a_1 \wedge \ldots \wedge a_n} \tag{8}$$

Theorem 1. (from Meseguer (1998)) *With the notation above,* $E \models_{\Omega} (\forall X)\ a \ \texttt{if}\ a_1 \wedge \ldots \wedge a_n$ *if and only if* $E \vdash_{\Omega} (\forall X)\ a \ \texttt{if}\ a_1 \wedge \ldots \wedge a_n$. *Moreover, any statement can be proved by first applying rule* (7), *then* (8), *and then a series of rules* (1) *to* (6).

This theorem is used within the correctness proof of the monitoring algorithm in Section 6.

3.1.3. Initial semantics and induction. MEL specifications are often intended to allow only a restricted class of models (or algebras). For example, a specification of natural numbers defined using the Peano equational axioms would have many "undesired" models, such as models in which the addition operation is not commutative, or models in which, for example, $10 = 0$. We restrict the class of models of a MEL specification only to those which are *initial*, that is, those which obey the *no junk no confusion* principle; therefore, our specifications have *initial semantics* (Goguen et al., 1977) in this paper. Intuitively, that means that only those models are allowed in which all elements can be "constructed" from smaller elements and in which no terms which cannot be proved equal are interpreted to the same elements.

By reducing the class of models, one can enlarge the class of sound inference rules. A major benefit one gets under initial semantics is that *proofs by induction become valid*. Since the proof of correctness for the main algorithm in this paper is done by induction on the structure of the temporal formula to monitor, it is important for the reader to be aware that the specifications presented from now on have initial semantics.

3.1.4. Syntactic sugar conventions. To make specifications easier to read, the following syntactic sugar conventions are widely accepted:

Subsorts. Given sorts s, s' with $\pi(s) = \pi(s') = k$, the declaration $s < s'$ is syntactic sugar for the conditional membership $(\forall x : k)\ x : s' \ \texttt{if}\ x : s$.

Operations. If $\sigma \in \Omega_{k_1 \ldots k_n, k}$ and $s_1, \ldots, s_n, s \in S$ with $\pi(s_1) = k_1, \ldots, \pi(s_n) = k_n$, $\pi(s) = k$, then the declaration $\sigma : s_1 \cdots s_n \to s$ is syntactic sugar for $(\forall x_1 : k_1, \ldots, x_n : k_n)\ \sigma(x_1, \ldots, x_n) : s \ \texttt{if}\ x_1 : s_1 \wedge \ldots \wedge x_n : s_n$.

Variables. $(\forall x : s, X)\ a \ \texttt{if}\ a_1 \wedge \ldots \wedge a_n$ is syntactic sugar for the Ω-sentence $(\forall x : \pi(s), X)\ a \ \texttt{if}\ a_1 \wedge \ldots \wedge a_n \wedge x : s$. With this, the operation declaration $\sigma : s_1 \cdots s_n \to s$ above is equivalent to $(\forall x_1 : s_1, \ldots, x_n : s_n)\ \sigma(x_1, \ldots, x_n) : s$.

3.2. Maude

Maude (Clavel et al., 1999) is a freely distributed high-performance system in the OBJ (Goguen et al., 2000) algebraic specification family, supporting both rewriting logic (Meseguer, 1992) and membership equational logic (Meseguer, 1998). 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 informally describe some of Maude's features via examples in this section, referring the interested reader to its manual (Clavel et al., 1999) for more details. The examples discussed in this subsection are not random. On the one hand they show all the major features of Maude that we need, while on the other hand they are part of our current JPaX implementation; several references to them will be made later in the paper. Maude supports modularization in the OBJ style. There are various kinds of modules, but we use only functional modules which follow the pattern "`fmod <name> is <body> endfm`", and which have initial semantics. The body of a functional module consists of a collection of declarations, of which we are using importation, sorts, subsorts, operations, variables and equations, usually in this order.

3.2.1. Defining logics for monitoring. In the following we 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 first 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 concept in 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` (more details regarding the communication between the running program and Maude will be given later):

```
fmod TRACE is
  protecting ATOM.
  sorts Event Event* Trace.
```

```
  subsorts Atom < Event < Event* < Trace.
  op empty : -> Event.
  op  : Event Event -> Event [assoc comm id: empty prec 23].
  var A : Atom.
  eq A A = A.
  op _* : Event -> Event*.
  op _,_ : Event Trace -> Trace [prec 25].
endfm
```

The statement `protecting ATOM` imports the module `ATOM` without changing its initial semantics. 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 *set* of atoms. The `subsorts` declaration declares `Atom` to be a subsort of `Event`, which in turn is a subsort of `Event*` which is a subsort 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 one event. Likewise, an atomic proposition can occur as an event, containing only this atomic proposition.

Operations can have attributes, such as associativity (A), commutativity (C), identity (I) as well as precedences, which are written between square brackets. When a binary operation is declared using the attributes A, C, and/or I, Maude uses built-in efficient specialized algorithms for matching and rewriting. However, semantically speaking, the A, C, and/or I attributes can be replaced by there corresponding equations. The attribute `prec` gives a precedence to an operator,[3] thus eliminating the need for most parentheses. Notice the special sort `Event*` which stays 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.

An event is defined as a set of atoms which should in fact be thought of as the set of all those atoms which "hold" in the new state of the event emitting program. Note the idempotency equation "`eq A A = A`", which ensures that an event is indeed a set. On the other hand, a trace is a an ordered list of events which can potentially have repetitions of events. For example, the event "$x = 5$" can occur several times during the execution of a program. Note that there is no need and consequently no definition of an empty trace.

Syntax and semantics are basic requirements to any logic. The following module introduces what we believe are the basic ingredients of monitoring logics, i.e., logics used for specifying monitoring requirements:

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

```
  eq [false] = false.

  var 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{A E} = true.
  eq A{E} = false [owise].
  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
```

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 in any monitoring logic. The sort `Bool` is built-in in Maude together with two constants `true` and `false`, which are different from those of sort `Formula`, and a generic operator `if_then_else_fi`. The "interpretation" operator `[_]` maps a formula to a Boolean value. Each logic implemented on top of `LOGICS-BASIC` is free to define it appropriately; here we only give the obvious mappings of `true` and `false` of `Formula` into `true` and `false` of `Bool`.

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 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. Each logic will further complete the definition of this operator. Note that the equation "`eq A{A E} = true`" speculates Maude's capability of performing matching modulo associativity, commutativity and identity (the attributes of the *set* concatenation on events); it basically says that `A{E}` is `true` if `E` contains the atom `A`. The next equation contains the attribute `[owise]`, stating that it should be applied only if any other equation fails to apply at a particular position.

Finally, the satisfaction relation is defined. Two obvious equations deal with the formulae `true` and `false`. The last two equations state that a trace satisfies an atomic proposition `A` if evaluating that atomic proposition `A` on the first event in the trace yields true. The remaining elements in the trace do not matter because `A` is a simple atom, so it refers to only the current state.

3.2.2. Defining propositional calculus. A rewriting decision procedure for propositional calculus due to Hsiang (1985) is adapted and presented. It provides the usual connectives _/_ (and), _++_ (exclusive or), _\/_ (or), !_ (negation), _->_ (implication), and _<->_(equivalence). The procedure reduces tautological 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 a canonical form consists of an exclusive or of conjunctions. In fact, this choice of basic operators corresponds to regarding propositional calculus as a Boolean ring rather than as a Boolean algebra. A major advantage of this choice is that normal forms are unique modulo associativity and commutativity. 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 operational 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 Goguen et al. (2000). Below we give its obvious translation to Maude together with its finite trace semantics, noticing that Hsiang (1985) 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
```

The statement "`extending LOGICS-BASIC`" imports the module `LOGICS-BASIC` with the reserve that its initial semantics can be extended. The operators "`and`" and "`xor`" come from the Maude's built-in module `BOOL` which is automatically imported by any other module.

Operators are declared with special attributes, such as `assoc` and `comm`, which enable Maude to use its specialized efficient internal rewriting algorithms. Once the module above is loaded[4] in Maude, reductions can be done as follows:

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

Notice that one should first declare the constants `a`, `b` and `c`. The last six equations in the module `PROP-CALC` are related to the semantics of propositional calculus. The default evaluation strategy for `[_]` is eager, so `[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 stated by Theorem 3.

4. Finite trace future time linear temporal logic

Classical (infinite trace) linear temporal logic (LTL) (Pnueli, 1977), (Manna and Pnueli 1992, 1995) 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 $t : \mathcal{N}^+ \to 2^{\mathcal{P}}$ 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. For example, the formula `[](X -> <>Y)` is true if for any time point (`[]`) it holds that if `X` is true then eventually (`<>`) `Y` is true. It is 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`.

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.

4.1. Finite trace semantics

As mentioned in Section 3.2.1, a trace is viewed as a non-empty finite sequence of program states, each state denoting the set of propositions that hold at that state. We shall first outline the finite trace LTL semantics using standard mathematical notation rather than Maude notation. The debatable issue here is what happens at the end of the trace. The choice to validate or invalidate all the atomic propositions does not work in practice, because there might be propositions whose values are always opposite to each other, such as, for example, "gate up" and "gate down". Driven by experiments, we found that a more reasonable assumption is to regard a finite trace as an infinite stationary trace in which the last event is repeated infinitely.

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 $length(e) = 1$ 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:

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

The semantics of the "next" operator reflects perhaps best the stationarity assumption of last events in finite traces.

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)` for any atomic proposition 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 does not.

4.2. Finite trace semantics in Maude

Now it can be relatively easily seen that the following Maude specification correctly "defines" 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 |= o X = E |= X.
  eq E,T |= o X = T |= X.
  eq E |= <> X = E |= X.
  eq E,T |= <> X = E,T |= X or T |= <> X.
  eq E |= [] X = E |= X.
  eq E,T |= [] X = E,T |= X and 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.
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:

```
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 30 ms (74 k rewrite steps) to reduce the first formula above and less than 1 s (254 k rewrite steps) to reduce the second on traces of 100 events (10 times larger than the above). Unfortunately, this algorithm does not 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 could not 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 cannot 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 worse-case 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 $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$, it follows that $T(n, m) > \binom{m}{n}$, that is, $T(n, m) = \Omega(n^m)$, which is of course unacceptable.

5. A backwards, asynchronous, but efficient algorithm

The satisfaction relation above for finite trace LTL can hence 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 first show how such an algorithm looks for a particular formula, and then present the main algorithm generator. The work in this section appeared as a technical report (Roşu and Havelund, 2001), but for a slightly different finite trace LTL, namely one in which all the atomic propositions were considered *false* at the and of the trace. As explained previously in the paper, we are now in the favor of a semantics where traces are considered stationary in their last event. The generated dynamic programming algorithms are as efficient as they can be and one can hope: linear in both the trace and the LTL formula. Unfortunately, they need to traverse the execution trace backwards, so they are trace storing and asynchronous. However, a similar but dual technique applies to past time LTL, producing very efficient forwards and synchronous algorithms (Havelund and Roşu, 2002; Havelund and Roşu, (to appear)).

5.1. An example

The formula we choose below is artificial (and will not be used later in the paper), but contains all four temporal operators. We believe that this example would practically be sufficient for the reader to foresee the general algorithm presented in the remaining of the section. Let `[]((p U q) -> <>(q -> or))` be an LTL formula and let $\varphi_1, \varphi_2, \ldots, \varphi_{10}$ be its subformulae, in breadth-first order:

$$
\begin{aligned}
\varphi_1 &= \texttt{[]((}p\ \texttt{U}\ q\texttt{) -> <>(}q\ \texttt{->}\ \texttt{or))},\\
\varphi_2 &= \texttt{(}p\ \texttt{U}\ q\texttt{) -> <>(}q\ \texttt{->}\ \texttt{or)},\\
\varphi_3 &= p\ \texttt{U}\ q,\\
\varphi_4 &= \texttt{<>(}q\ \texttt{->}\ \texttt{or)},\\
\varphi_5 &= p,\\
\varphi_6 &= q,\\
\varphi_7 &= q\ \texttt{->}\ \texttt{or},\\
\varphi_8 &= q,\\
\varphi_9 &= \texttt{or},\\
\varphi_{10} &= r.
\end{aligned}
$$

Given any finite trace $t = e_1 e_2 \ldots e_n$ of n events, one can recursively define a matrix $s[1..n, 1..10]$ of Boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$ as follows:

$$
\begin{aligned}
s[i, 10] &= (r \in e_i)\\
s[i, 9] &= s[i + 1, 10]\\
s[i, 8] &= (q \in e_i)\\
s[i, 7] &= s[i, 8]\ \textit{implies}\ s[i, 9]\\
s[i, 6] &= (q \in e_i)\\
s[i, 5] &= (p \in e_i)\\
s[i, 4] &= s[i, 7]\ \textit{or}\ s[i + 1, 4]\\
s[i, 3] &= s[i, 6]\ \textit{or}\ (s[i, 5]\ \textit{and}\ s[i + 1, 3])\\
s[i, 2] &= s[i, 3]\ \textit{implies}\ s[i, 4]\\
s[i, 1] &= s[i, 2]\ \textit{and}\ s[i + 1, 1],
\end{aligned}
$$

for all $i < n$, where *and*, *or*, *implies* are ordinary Boolean operations and $==$ is the equality predicate, where $s[n, 1..10]$ are defined as below:

$$
\begin{aligned}
s[n, 10] &= (r \in e_n)\\
s[n, 9] &= s[n, 10]\\
s[n, 8] &= (q \in e_n)
\end{aligned}
$$

$$
\begin{aligned}
s[n,7] &= s[n,8] \textit{ implies } s[n,9]\\
s[n,6] &= (q \in e_n)\\
s[n,5] &= (p \in e_n)\\
s[n,4] &= s[n,7]\\
s[n,3] &= s[n,6]\\
s[n,2] &= s[n,3] \textit{ implies } s[n,4]\\
s[n,1] &= s[n,2].
\end{aligned}
$$

Note again that the trace needs to be *traversed backwards*, and that the row n of s is filled according to the stationary view of finite traces in their last event. An important observation is that, like in many other dynamic programming algorithms, one does not have to store all the table $s[1..n, 1..10]$, which would be quite large in practice; in this case, one needs only two rows, $s[i, 1..10]$ and $s[i+1, 1..10]$, which we shall write *now* and *next* from now on, respectively. It is now only a simple exercise to write up the following algorithm:

$$
\begin{aligned}
&\text{INPUT:trace } t = e_1 e_2 \ldots e_n\\
&\quad next[10] \leftarrow (r \in e_n);\\
&\quad next[9] \leftarrow \widehat{\text{next}}[10];\\
&\quad next[8] \leftarrow (q \in e_n);\\
&\quad next[7] \leftarrow next[8] \textit{ implies } next[9];\\
&\quad next[6] \leftarrow (q \in e_n);\\
&\quad next[5] \leftarrow (p \in e_n);\\
&\quad next[4] \leftarrow next[7];\\
&\quad next[3] \leftarrow next[6];\\
&\quad next[2] \leftarrow next[3] \textit{ implies } next[4];\\
&\quad next[1] \leftarrow next[2];\\
&\quad \textbf{for} i = n-1 \textbf{downto} 1 \textbf{do}\{\\
&\quad now[10] \leftarrow (r \in e_i);\\
&\quad now[9] \leftarrow next[10];\\
&\quad now[8] \leftarrow (q \in e_i);\\
&\quad now[7] \leftarrow now[8] \textit{ implies } now[9];\\
&\quad now[6] \leftarrow (q \in e_i);\\
&\quad now[5] \leftarrow (p \in e_i);\\
&\quad now[4] \leftarrow now[7] \textit{ or } next[4];\\
&\quad now[3] \leftarrow now[6] \textit{ or } (now[5] \textit{ and } next[3]);\\
&\quad now[2] \leftarrow now[3] \textit{ implies } now[4];\\
&\quad now[1] \leftarrow now[2] \textit{ and } next[1];
\end{aligned}
$$

next ← *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 Havelund and Roşu, Havelund and Roşu (2002, (to appear)). 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.

5.2. *Generating dynamic programming algorithms*

We now formally describe our algorithm that synthesizes dynamic programming algorithms like the one above from LTL formulae. Our synthesizer is generic, the potential user being expected to adapt it to his/her desired target language. The algorithm consists of three main steps:

Breadth First Search. The LTL formula should be first visited in breadth-first search (BFS) order to assign increasing numbers to subformulae as they are visited. Let $\varphi_1, \varphi_2, \ldots, \varphi_m$ be the list of all subformulae in BFS order. Because of the semantics of finite trace LTL, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i+1} \models \varphi_{j'}$ for all $j \leq j' \leq m$. This recurrence gives the order in which one should generate the code.

Loop Initialization. Before we generate the "for" loop, we should first initialize the vector *next*[1..*m*], which basically gives the truth values of the subformulae on the empty trace. According to the semantics of LTL, one should fill the vector *next* backwards. For a given $m \geq j \geq 1$, *next*[j] is calculated as follows:

- If φ_j is a variable then *next*[j] = ($\varphi_j \in e_n$). In a more complex setting, where φ_j was a state predicate, one would have to evaluate φ_j in the final state in the execution trace;
- If φ_j is $!\varphi_{j'}$ for some $j < j' \leq m$, then *next*[j] = *not next*[j'], where *not* is the negation operation on Booleans (bits);
- If φ_j is $\varphi_{j_1}\ Op\ \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then *next*[j] = *next*[j_1] *op next*[j_2], where Op is any propositional operation and *op* is its corresponding Boolean operation;
- If φ_j is $\circ\varphi_{j'}$, $[]\,\varphi_{j'}$, or $<>\varphi_{j'}$, then clearly *next*[j] = *next*[j'] according to the stationary semantics of our finite trace LTL;
- If φ_j is $\varphi_{j_1}\ \mathtt{U}\ \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then *next*[j] = *next*[j_2] for the same reason as above.

Loop Generation. Because of the dependences in the recursive definition of finite trace LTL satisfaction relation, one is expected to visit the remaining of the trace backwards, so the loop index will vary from $n-1$ down to 1. The loop body will update/calculate the vector *now* and in the end will move it into the vector *next* to serve as basis for the next iteration. At a certain iteration i, the vector *now* is updated also backwards as follows:

- If φ_j is a variable then $now[j] = (\varphi_j \in e_i)$.
- If φ_j is $!\varphi_{j'}$ for some $j < j' \leq m$, then $now[j] = not\ now[j']$;
- If φ_j is $\varphi_{j_1}\ Op\ \varphi_{j_2}$ for $j < j_1, j_2 \leq m$, then $now[j] = now[j_1]\ op\ now[j_2]$, where Op is any propositional operation and op is its corresponding Boolean operation;
- If φ_j is $\circ\varphi_{j'}$ then $now[j] = next[j']$ since φ_j holds now if and only if $\varphi_{j'}$ held at the previous step (which processed the next event, the $i+1$-th);
- If φ_j is $[]\varphi_{j'}$ then $now[j] = now[j']\ and\ next[j]$ because φ_j holds now if and only if $\varphi_{j'}$ holds now and φ_j held at the previous iteration;
- If φ_j is $<>\varphi_{j'}$ then $now[j] = now[j']\ or\ next[j]$ because of similar reasons as above;
- If φ_j is $\varphi_{j_1}\ \mathtt{U}\ \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $now[j] = now[j_2]\ or\ (now[j_1]\ and\ next[j])$.

After each iteration, $next[1]$ says whether the initial LTL formula is validated by the trace $e_i e_{i+1} \ldots e_n$. Therefore, the desired output is $next[1]$ after the last iteration. Putting all the above together, one can now write up the generic pseudocode presented below which can be implemented very efficiently on any current platform. Since the BFS procedure is linear, the algorithm synthesizes a dynamic programming algorithm from an LTL formula in linear time and of linear size with the size of the formula.

The following generic program implements the discussed technique. It takes as input an LTL formula and generates a "for" loop which traverses the trace of events backwards, thus validating or invalidating the formula.

```
INPUT: LTL formula φ
output("INPUT: trace t = e_1 e_2 ... e_n");
let φ_1, φ_2, ..., φ_m be all the subformulae of φ is BFS order
for j = m downto 1 do {
      output("next[", j, "] ← ");
       if φ_j is a variable then
output((φ_j,"∈ e_n);");
       if φ_j = !φ_j' then output("not next[",j', "];");
       if φ_j = φ_j1 Op φ_j2 then output("next[",j_1, "] op next[", j_2, "];");
       if φ_j = ○φ_j' then output("next[",j', "];");
       if φ_j = [] φ_j' then output("next[",j', "];");
       if φ_j = <>φ_j' then output("next[",j', "];");
       if φ_j = φ_j1 U φ_j2 then output("next[",j_2, "];"); }
output("for i = n − 1 downto 1 do {");
for j = m downto 1 do {
      output("       now[", j, "] ← ");
       if φ_j is a variable then output((φ_j,"∈ e_i);");
       if φ_j = !φ_j' then output("not now[",j', "];");
       if φ_j = φ_j1 Op φ_j2 then output("now[",j_1, "] op now[", j_2, "];");
       if φ_j = ○φ_j' then output("next[", j', "];");
       if φ_j = [] φ_j' then output("now[", j', "] and next[", j, "];" );
       if φ_j = <>φ_j' then output("now[", j', "] or next[", j, "];" );
       if φ_j = φ_j1 U φ_j2 then output("now[", j_2, "] or (now[",j_1, "]
```

```
    and next[", j, "]);"); }
output("      next ← now; }");
output("output next[1];");
```

where Op is any propositional connective and op is its corresponding Boolean operator.

The Boolean operations used above are usually very efficiently implemented on any microprocessor and the vectors of bits *next* 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.

The dynamic programming technique presented in this section 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 synchronously.

6. A forwards and often synchronous 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.

6.1. An event consuming algorithm

We will implement this rewriting based algorithm by extending the definition of the event consuming operation `_{_} : Formula Event* -> Formula` to temporal operators, with the following intuition. Assuming a trace `E,T` consisting of event `E` followed by trace `T`, 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 (o X){E} = X .
  eq (o X){E *} = X{E *} .
  eq (<> X){E} = X{E} \/ <> X .
  eq (<> X){E *} = X{E *} .
  eq ([] X){E}   = X{E} /\ [] X .
  eq ([] 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 .
  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 on the rest of the trace in order for `X` to hold now.

As an example, consider again the traffic light controller safety formula `[](green -> !red U yellow)`, which is first rewritten to `[](true ++ green ++ green /\ (true ++ red) U yellow)` by the equations in module `PROP-CALC`. This formula modified by an event `green yellow` (notice that two lights can be lit at the same time) yields the rewriting sequence

```
([](true ++ green ++ green /\ (true ++ red) U yellow)){green yellow}  =*=>
(true ++ green{green yellow}
   ++ green{green yellow} /\ ((true ++ red) U yellow){green yellow})
     /\ [](true ++ green ++ green /\ (true ++ red) U yellow)          =*=>
((true ++ red) U yellow){green yellow})
     /\ [](true ++ green ++ green /\ (true ++ red) U yellow)          =*=>
(yellow{green yellow} \/ ((true ++ red{green yellow}) /\ (true ++ red) U yellow)
     /\ [](true ++ green ++ green /\ (true ++ red) U yellow)          =*=>
[](true ++ green ++ green /\ (true ++ red) U yellow)
```

which is exactly the original formula, while the same formula transformed by just the event `green` yields

```
([](true ++ green ++ green /\ (true ++ red) U yellow)){green}         =*=>
(true ++ green{green}
   ++ green{green} /\ ((true ++ red) U yellow){green})
     /\ [](true ++ green ++ green /\ (true ++ red) U yellow)          =*=>
((true ++ red) U yellow){green})
     /\ [](true ++ green ++ green /\ (true ++ red) U yellow)          =*=>
(yellow{green} \/ ((true ++ red{green}) /\ (true ++ red) U yellow)
     /\ [](true ++ green ++ green /\ (true ++ red) U yellow)          =*=>
(true ++ red) U yellow /\ [](true ++ green ++ green /\ (true ++ red) U yellow)
```

which further modified by an event `red` yields

```
(yellow{red} \/ (true ++ red{red}) /\ (true ++ red) U yellow)
     /\ ([](true ++ green ++ green /\ (true ++ red) U yellow)){red}   =*=>
false /\ ([](true ++ green ++ green /\ (true ++ red) U yellow)){red}  =*=>
false
```

When the current formula becomes `false`, as it happened above, we say that the original formula has been violated. Indeed, the current formula will remain `false` for any subsequent trace of events, so the result of the monitoring session will be `false`.

Note that the rewriting system described so far obviously terminates, because what it does is to propagate the current event to the atomic subformulae, replace those by either true or false, and eventually canonize the newly obtained formula.

Some operators could be defined 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 in practice though, because more rewrites are needed. This happens regardless of whether one enables memoization (explained in detail in Section 6.3) or not, because memoization brings a real benefit only when previously processed terms are to be reduced again.

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 at each step is always kept small by being reduced to normal form via the equations in the `PROP-CALC` module in Section 3.2.2.

Our current JPAX implementation of the rewriting algorithm above executes the last two rules of the module `LTL-REVISED` outside the main rewriting engine. More precisely, Maude is started in its *loop mode* (Clavel et al., 2002), which provides the capability of enabling rewriting rules in a reactive system style: a "state" term is stored, which can then be modified via rewriting rules that are activated by ASCII text events that are provided via the standard I/O. JPAX starts a Maude process and assigns the formula to be monitored as its loop mode state. Then, as events are received from the monitored program, they are filtered and forwarded to the Maude module, which then enables rewriting on the term `X{E}`, where `X` is the current formula and `E` is the newly received event; the normal form of this reduction, a formula, is stored as the new loop mode state term. The process continues until the last event is received. JPAX tags the last event, asking Maude to reduce a term `X{E *}`; the result will be either `true` or `false`, which is reported to the user at the end of the monitoring session[6].

A natural question here is how big the stored formula can grow during a monitoring session. Such a formula will consist of Boolean combinations of sub-formulae of the initial formula, kept in a minimal canonical form. This can grow exponentially in the size of the initial formula in the worst-case (see (Roşu and Viswanathan, 2003) for a related result for extended regular expressions).

Theorem 2. *For any formula* `X` *of size m and any sequence of events to be monitored* `E1, E2, ..., En`, *the formula* `X{E1}{E2} ... {En}` *needs* $O(2^m)$ *space to be stored. Moreover,*

the exponential space cannot be avoided: *any synchronous or asynchronous forwards monitoring algorithm for LTL requires space* $\Omega(2^{c\sqrt{m}})$, *where c is some fixed constant.*

Proof: Due to the Boolean ring simplification rules in `PROP-CALC`, any LTL formula is kept in a canonical form, which is an exclusive disjunction of conjunctions, where conjuncts have temporal operators at top. Moreover, after processing any number of events, in our case `E1`, `E2`, ..., `En`, the conjuncts in the normal form of `X`$\{$`E1`$\}\{$`E2`$\}\ldots\{$`En`$\}$ are subterms of the initial formula `X`, each having a temporal operator at its top. Since there are at most m such subformulae of `X`, it follows that there are at most 2^m possibilities to combine them in a conjunction. Therefore, one needs space $O(2^m)$ to store any exclusive disjunction of such conjunctions. This reasoning only applies on "idealistic" rewriting engines, which carefully optimize space needs during rewriting. It is not clear to us whether Maude is able to attain this space upper bound in all situations.

For the space lower bound of any finite trace LTL monitoring algorithm, consider a simplified framework with only two atomic predicate and therefore only four possible states. For simplicity, we encode these four states by 0, 1, # and \$. Consider also some natural number k and the language

$$L_k = \{\sigma\#w\#\sigma'\$w \mid w \in \{0,1\}^k \text{ and } \sigma, \sigma' \in \{0,1,\#\}^*\}.$$

This language was previously used in several works (Kupferman and Vardi, 1998; Kupferman and Vardi, 1999; Roşu and Viswanathan, 2003) to prove lower bounds. The language can be shown to contain exactly those finite traces satisfying the following LTL formula (Kupferman and Vardi, 1999) of size $\Theta(k^2)$:

$$\phi_k = [(!\$)\ \mathtt{U}\ (\$\ /\backslash\ \mathtt{o}[](!\$))]\ /\backslash\ \langle\rangle\Big[\#\ /\backslash\ \mathtt{o}^{n+1}\#\ /\backslash\ \bigwedge_{i=1}^{n}((\mathtt{o}^i 0\ /\backslash\ [](\$\text{->}\mathtt{o}^i 0))\backslash/ (\mathtt{o}^i 1\ /\backslash\ [](\$\text{->}\mathtt{o}^i 1)))\Big]. \tag{9}$$

Let us define an equivalence relation on finite traces in $(0+1+\#)^\star$. For a $\sigma \in (0+1+\#)^\star$, define $S(\sigma) = \{w \in (0+1)^k \mid \exists\lambda_1, \lambda_2.\ \lambda_1\#w\#\lambda_2 = \sigma\}$. We say that $\sigma_1 \equiv_k \sigma_2$ if and only if $S(\sigma_1) = S(\sigma_2)$. Now observe that the number of equivalence classes of $\equiv_k$ is 2^{2^k}; this is because for any $S \subseteq (0+1)^k$, there is a σ such that $S(\sigma) = S$.

Since $|\phi_k| = \Theta(k^2)$, it follows that there is some constant c' such that $|\phi_k| \leq c'k^2$ for all large enough k. Let c be the constant $1/\sqrt{c'}$. We will prove this lower bound result by contradiction. Suppose $\mathcal{A}$ is an LTL forwards monitoring algorithm that uses less that $2^{c\sqrt{m}}$ space for any LTL formulae of large enough size m. We will look at the behavior of the algorithm $\mathcal{A}$ on inputs of the form ϕ_k. So $m = |\phi_k| \leq c'k^2$, and $\mathcal{A}$ uses less than 2^k space. Since the number of equivalence classes of $\equiv_k$ is 2^{2^k}, by the pigeon hole principle there must be two strings $\sigma_1 \not\equiv_k \sigma_2$ such that the memory of $\mathcal{A}$ on ϕ_k after reading $\sigma_1\$$ is the same as the memory after reading $\sigma_2\$$. In other words, $\mathcal{A}$ running on ϕ_k will give the same answer on all traces of the form $\sigma_1\$w$ and $\sigma_2\$w$. Now since $\sigma_1 \not\equiv_k \sigma_2$, it follows that $(S(\sigma_1) \setminus S(\sigma_2)) \cup (S(\sigma_2) \setminus S(\sigma_1)) \neq \emptyset$. Take $w \in (S(\sigma_1) \setminus S(\sigma_2)) \cup (S(\sigma_2) \setminus S(\sigma_1))$. Then clearly, exactly one out of $\sigma_1\$w$ and $\sigma_2\$w$ is in L_k, and so $\mathcal{A}$ running on ϕ_k gives the wrong answer on one of these inputs. Therefore, $\mathcal{A}$ is not a correct. □

It seems, however, that this worst-case exponential complexity in the size of the LTL formula is more of theoretical importance than practical, since in general the size of the formula rarely grew more than twice in our experiments. Verification results are very encouraging and show that this optimized semantics is orders of magnitude 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 4.2, 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 Section 6.3 we will see how this algorithm can be made even more efficient, using memoization.

6.2. 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 4. 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. In other words, the proof that follows was *not* generated automatically. However, it could have been mechanized by using proof assistants and/or theorem provers like KUMO (Goguen et al., 2000), PVS (Shankar et al., 1993), or Maude-ITP (Clavel, 2001). We have already done it in PVS, but we prefer to use only plain Maude in this paper.

Theorem 3. *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 these lemmas 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 hypotheses are added to the following specification via equations. Notice that we merged the two proofs to save space. A proof assistant like KUMO, PVS or Maude-ITP 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}.
  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 could not be proved equal; they can still be equal.

```
reduce (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:

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

where $\mathcal{P}$(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
reduce e   |= x == e   |- x .
reduce 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. □

6.3. Further optimization by memoization

Even though the formula transforming algorithm in Section 6.1 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 Section 6.1, 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 these 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, because the operational semantics of all the operators can be defined recursively, so formulae repeat often during the monitoring process. However, there might be monitoring logics where memoization could be less efficient. Such a logic would probably be an extension of LTL with time, allowing formulae of the form "`<5> X`" with the meaning "eventually in 5 units of time `X`", because of a potentially very large number of terms to be memoized: `<5> X`, `<4> X`, etc. Experimentation is certainly needed if one designs a new logic for monitoring and wants to use memoization.

7. Generating forwards, synchronous and 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 rewriting engine modulo associativity and commutativity, 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 give a technique, based on ideas presented previously in the paper, to generate automata-based optimal monitors for future time LTL formulae. By optimality we here mean everything one may expect, such as minimal number of states, forwards traversal of execution traces, synchronicity, efficiency, but also less standard optimality features, such as transiting from one state to another with a minimum amount of computation. In order to effectively do this we introduce the notion of *binary transition tree* (BTT), as a generalization of binary decision diagrams (BDD) (Bryant, 1986), whose purpose is to provide an *optimal order* in which state predicates need to be evaluated to decide the next state. The motivation for this is that in practical applications evaluating a state predicate is a time consuming task, such as for example to check whether a vector is sorted. The associated finite state machines are called *binary transition tree finite state machines* (BTT-FSM).

The drawback of generating an optimal BTT-FSM statically, i.e., before monitoring, is the worst-case double exponential time/space required at startup. Therefore, the algorithm presented in this section is recommended for situations where the LTL formulae to monitor are relatively small in size but the runtime overhead is desired to be minimal. It is worth noting that the BTT-FSM generation process can potentially take place on a machine different from the one performing the monitoring. In particular, one can think of a WWW fast server offering LTL-to-BTT-FSM services via the Internet, which can also maintain a database of already generated BTT-FSMs to avoid regenerating the same monitors.

7.1. Multi-transition and binary transition tree finite state machines

To keep the runtime overhead of monitors low, it is crucial to do as little computation as possible in order to proceed to the next state. In the sequel we assume that finite state monitors are desired to efficiently change their state (when a new event is received) to one of possible states $s_1, s_2, \ldots, s_n$, under the knowledge that a transition to each such state is enabled deterministically by some Boolean formula, $p_1, p_2, \ldots, p_n$, respectively, on atomic state predicates.

Definition 4. Let S be a set whose elements are called *states*, and let A be another set, whose elements are called *atomic predicates*. Let $\{s_1, s_2, \ldots, s_n\} \subseteq S$ and let $p_1, p_2, \ldots, p_n$ be propositions over atoms in A (using the usual Boolean combinators presented in Section 3.2.2), with the property that exactly one of them is true at any moment, that is, $p_1 \vee p_2 \vee \ldots \vee p_n$ holds and for any distinct p_i, p_j, it is the case that $p_i \rightarrow \neg p_j$ holds. Then we call the n-tuple $[p_1?s_1, p_2?s_2, \ldots, p_n?s_n]$ a *multi-transition* (MT) over states S and atomic predicates (or simply atoms) A. Let $MT(S, E)$ be the set of multi-transitions over states S and atoms A.

Since the rest of the paper is concerned with rather theoretical results, from now on we use mathematical symbols for the Boolean operators instead of ASCII symbols like in Section 3.2.2. The intuition underlying multi-transitions is straightforward: depending on which of the propositions $p_1, p_2, \ldots, p_n$ is true at a given moment, a corresponding transition to exactly one of the states $s_1, s_2, \ldots, s_n$ will take place. Formally,

Definition 5. Maps $\theta : A \rightarrow \{true, false\}$ are called *events* from now on. With the notation above, given an event θ, we define a map $\theta_{MT} : MT(S, A) \rightarrow S$ as $\theta_{MT}([p_1?s_1, p_2?s_2, \ldots, p_n?s_n]) = s_i$, where $\theta(p_i) = true$; notice that $\theta(p_j) = false$ for any $1 \leq j \neq i \leq n$.

Definition 6. Given an event $\theta : A \rightarrow \{true, false\}$, let e_θ denote the list of atomic predicates a with $\theta(a) = true$. There is an obvious correspondence between events as maps $A \rightarrow \{true, false\}$ and events as lists of atomic predicates, which justifies our implementation of events in Section 3.2.1. We take the liberty to use either the map or the list notation for events from now on in the paper. We let $\mathcal{E}$ denote the set of all events and call lists, or words, $e_{\theta_1} \ldots e_{\theta_n} \in \mathcal{E}^\star$ (finite) traces; this is also consistent with our mechanical Maude ASCII notation in Section 3.2.1, except that we prefer not to separate events by commas in traces from now on, to avoid mathematical notational conflicts.

We next define binary transition trees.

Definition 7. Under the same notations as in the previous definition, a *binary transition tree* (BTT) over states S and atoms A is a term over syntax

$$BTT ::= S \mid (A \ ? \ BTT : BTT).$$

We let $BTT(S, A)$ denote the set of binary transition trees over states S and atoms A. Given an event $\theta : A \to \{true, false\}$, we define a map $\theta_{BTT} : BTT(S, A) \to S$ inductively as follows:

$$\begin{aligned} \theta_{BTT}(s) &= s \ \text{ for any } s \in S, \\ \theta_{BTT}(a \ ? \ b_1 : b_2) &= \theta_{BTT}(b_1) \ \text{ if } \theta(a) = true, \text{ and} \\ \theta_{BTT}(a \ ? \ b_1 : b_2) &= \theta_{BTT}(b_2) \ \text{ if } \theta(a) = false. \end{aligned}$$

A binary transition tree b in $BTT(S, A)$ is said to *implement* a multi-transition t in $MT(S, A)$ if and only if $\theta_{BTT}(b) = \theta_{MT}(t)$ for any map $\theta : A \to \{true, false\}$.

BTTs generalize BDDs (Bryant, 1986), which can be obtained by taking $S = \{true, false\}$. As an example of BTT, $a1 \ ? \ a2 \ ? \ s1 : a3 \ ? \ false : s2 : a3 \ ? \ s2 : true$ says "eval a_1; if a_1 then (eval a_2; if a_2 then go to state $s1$ else (eval a_3; if a_3 then go to state *false* else go to state $s2$)) else (eval a_3; if a_3 then go to state $s2$ else go to state *true*)". Note that *true* and *false* are just some special states. Depending on the application they can have different meanings, but in our applications *true* typically means that the monitoring requirement has been fulfilled, while *false* means that it has been violated. It is often convenient to represent BTTs graphically, such as the one in figure 2.

Definition 8. A *multi-transition finite state machine* (MT-FSM) is a triple (S, A, μ), where S is a set of states, A is a set of atomic predicates, and μ is a map from $S - \{true, false\}$ to $MT(S, A)$. If S contains *true* and/or *false*, then $\mu(true) = [true?true]$ and/or $\mu(false) =$

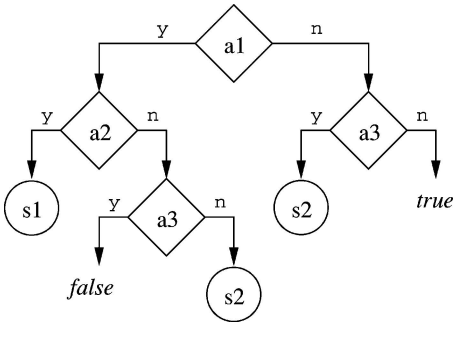

Figure 2. Graphical representation for the BTT $a1 \ ? \ a2 \ ? \ s1 : a3 \ ? \ false : s2 : a3 \ ? \ s2 : true$.

[*true*?*false*], respectively. A *binary transition tree finite state machine* (BTT-FSM) is a triple (S, A, β), where S and A are like before and β is a map from $S - \{true, false\}$ to $BTT(S, A)$. If S contains *true* and/or *false*, then it is the case that $\beta(true) = [true]$ and/or $\beta(false) = [false]$, respectively. For an event $\theta : A \to \{true, false\}$ in any of these machines, we let $s \xrightarrow{\theta} s'$ denote the fact that $\theta_{MT}(\mu(s)) = s'$ or $\theta_{BTT}(\beta(s)) = s'$, respectively. We take the liberty to call $s \xrightarrow{\theta} s'$ a *transition*. Also, we may write $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \cdots s_n \xrightarrow{\theta_n} s_{n+1}$ and call it a *sequence of transitions* whenever $s_1 \xrightarrow{\theta_1} s_2, \ldots, s_n \xrightarrow{\theta_n} s_{n+1}$ are transitions.

Note that S is not required to contain the special states *true* and *false*, but if it contains them, these states transit only to themselves. The finite state machine notions above are intended to abstract the intuitive concept of a monitor. The states in S are monitor states, and some of them can trigger side effects in practice, such as messages sent or reported to users, actions taken, feedback sent to the monitored program for guidance purposes, etc.

Definition 9. If $true \in S$ then a sequence of transitions $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \cdots s_n \xrightarrow{\theta_n} true$ is called a *validating sequence* (*for* s_1); if $false \in S$ then $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \cdots s_n \xrightarrow{\theta_n} false$ is called an *invalidating sequence* (*for* s_1). These notions naturally extend to corresponding finite traces of events $e_{\theta_1} \ldots e_{\theta_n}$.

BTT-FSMs can and should be thought of as efficient "implementations" of MT-FSMs, in the sense that they implement multi-transitions as binary transition trees. These allow one to reduce the amount of computation in a monitor implementing a BTT-FSM to only evaluate at most all the atomic predicates in a given state in order to decide to which state to go next; however, it will only very rarely be the case that *all* the atomic predicates need to be evaluated.

A natural question is why one should bother at all then with defining and investigating MT-FSMs. Indeed, what one really looks for in the context of FSM monitoring is efficient BTT-FSMs. However, MT-FSMs are nevertheless an important intermediate concept as it will become clearer later in the paper. This is because they have the nice property of *state mergeability*, which allows one to elegantly generate MT-FSMs from logical formulae. By state mergeability we mean the following. Suppose that during a monitor generation process, such as that for LTL that will be presented in Section 7.3, one proves that states s and s' are "logically equivalent" (for now, equivalence can be interpreted intuitively: they have the same behavior), and that s and s' have the multi-transitions $[p_1?s_1, p_2?s_2, \ldots, p_n?s_n]$ and $[p'_1?s'_1, p'_2?s'_2, \ldots, p'_n?s'_{n'}]$. Then we can merge s and s' into one state whose multi-transition is $\text{MERGE}([p_1?s_1, p_2?s_2, \ldots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \ldots, p'_n?s'_{n'}])$, defined as follows:

$\text{MERGE}([p_1?s_1, p_2?s_2, \ldots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \ldots, p'_n?s'_{n'}])$
contains all choices $p?s''$, where s'' is a state in $\{s_1, s_2, \ldots, s_n\} \cup \{s'_1, s'_2, \ldots, s'_{n'}\}$ and

- p is p_i when $s'' = s_i$ for some $1 \leq i \leq n$ and $s'' \neq s'_{i'}$ for all $1 \leq i' \leq n'$, or
- p is $p'_{i'}$ when $s'' = s'_{i'}$ for some $1 \leq i' \leq n'$ and $s'' \neq s_i$ for all $1 \leq i \leq n$, or
- p is $p_i \vee p'_{i'}$ when $s'' = s_i$ for some $1 \leq i \leq n$ and $s'' = s'_{i'}$ for some $1 \leq i' \leq n'$.

It is easy to see that this multi-transition merging operation is well defined, that is,

Proposition 10. MERGE($[p_1?s_1, p_2?s_2, \ldots, p_n?s_n]$, $[p'_1?s'_1, p'_2?s'_2, \ldots, p'_n?s'_{n'}]$) *is a well-formed multi-transition whenever both* $[p_1?s_1, p_2?s_2, \ldots, p_n?s_n]$ *and* $[p'_1?s'_1, p'_2?s'_2, \ldots, p'_n?s'_{n'}]$ *are well-formed multi-transitions. Therefore,* MERGE *can be seen as a function* $\mathcal{P}_f(MT(S, A)) \to MT(S, A)$*, where* $\mathcal{P}_f$ *is the finite powerset operator.*

There are situations when a definite answer, *true* or *false* is desired at the end of the monitoring session, as it will be in the case of LTL. As explained in Section 4, the intuition for the last event in an execution trace is that the trace is infinite and stationary in that last event. This seems to be the best and simplest assumption about future when a monitoring session is ended. For such situations, we enrich our definitions of MT-FSM and BTT-FSM with support for terminal events:

Definition 11. A *terminating multi-transition finite state machine* (abbreviated MT-FSM*) is a tuple $(S, A, \mu, \mu^\star)$, where (S, A, μ) is an MT-FSM and $\mu^\star$ is a map from $S - \{true, false\}$ to $MT(\{true, false\}, A)$. A *terminating binary transition tree finite state machine* (BTT-FSM*) is a tuple $(S, A, \beta, \beta^\star)$, where (S, A, β) is a BTT-FSM and $\beta^\star$ is a map from $S - \{true, false\}$ to $BTT(\{true, false\}, A)$. For a given event $\theta : A \to \{true, false\}$ in any of these finite state machines, we let $s \xrightarrow{\theta^\star} true$ (or *false*) denote the fact that $\theta_{MT}(\mu^\star(s)) = true$ (or *false*) or $\theta_{BTT}(\beta^\star(s)) = true$ (or *false*), respectively. We call $s \xrightarrow{\theta^\star} true$ (or *false*) a *terminating transition*. A sequence $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \cdots s_n \xrightarrow{\theta_n^\star} true$ is called an *accepting sequence (for* s_1*)* and a sequence $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \cdots s_n \xrightarrow{\theta_n^\star} false$ is called a *rejecting sequence (for* s_1*)*. These notions also naturally extend to corresponding finite traces of events $e_{\theta_1} \ldots e_{\theta_n}$.

Languages can be associated to states in MT-FSM*s or BTT-FSM*s as finite words of events.

Definition 12. Given a state $s \in S$ in an MT-FSM* or in a BTT-FSM* M, we let $\mathcal{L}_M(s)$ denote the set of finite traces $e_{\theta_1} \ldots e_{\theta_n}$ with the property that $s \xrightarrow{\theta_1} \cdots \xrightarrow{\theta_n^\star} true$ is an accepting sequence in M. If a state $s_0 \in S$ is desired to be *initial*, then we write it at the end of the tuple, such as $(S, A, \mu, \mu*, s_0)$ or $(S, A, \beta, \beta*, s_0)$, and let $\mathcal{L}_M$ denote $\mathcal{L}_M(s_0)$. If $s_0 \xrightarrow{\theta_1} s_1 \cdots \xrightarrow{\theta_n} s_n$ is a sequence of transitions from the initial state, then $e_{\theta_1} \ldots e_{\theta_n}$ is called a *valid prefix* if and only if $e_{\theta_1} \ldots e_{\theta_n} t \in \mathcal{L}_M$ for any (empty or not) trace t, and it is called an *invalid prefix* if and only if $e_{\theta_1} \ldots e_{\theta_n} t \notin \mathcal{L}_M$ for any trace t.

The following is immediate.

Proposition 13. *If* $true \in S$ *then* $\mathcal{L}_M(true) = \mathcal{E}^\star$*, and if* $false \in S$ *then* $\mathcal{L}_M(false) = \emptyset$*. If* $s \xrightarrow{\theta} s'$ *in* M *then* $\mathcal{L}_M(s') = \{t \mid e_\theta t \in \mathcal{L}_M(s)\}$*; more generally, if* $s \xrightarrow{\theta_1} s_1 \cdots \xrightarrow{\theta_n} s_n$ *is a sequence of transitions in* M *then* $\mathcal{L}_M(s_n) = \{t \mid e_{\theta_1} \ldots e_{\theta_n} t \in \mathcal{L}_M(s)\}$*. In particular, if* $s = s_0$ *then* $e_{\theta_1} \ldots e_{\theta_n}$ *is a valid prefix if and only if* $\mathcal{L}_M(s_n) = \mathcal{E}^\star$*, and it is an invalid prefix if and only if* $\mathcal{L}_M(s_n) = \emptyset$*.*

7.2. From MT-FSMs to BTT-FSMs

Supposing that one has encoded a logical requirement into an MT-FSM (we shall see how to do it for LTL in the next subsection), the next important step is to generate an efficient equivalent BTT-FSM. 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, so they are preferred to MT-FSMs. What one needs to do is to develop a procedure that takes a multi-transition and returns a BTT. More BTTs can encode the same multi-transition, so one needs to develop some criteria to select the better ones. A natural selection 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 the expensive 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 $a_1, \ldots, a_k$ that hold with probabilities $\pi_1, \ldots, \pi_k$ and have costs $c_1, \ldots, c_k$, respectively, and a multi-transition $p_1?s_1, \ldots, p_n?s_n$ where $p_1, \ldots, p_n$ are Boolean formulae over $a_1, \ldots, a_k$.
Output: A BTT implementing the multi-transition that probabilistically minimizes the amount of computation to decide the next state.

The probabilities and the costs in the problem above can be estimated either by static analysis of the source code of the program, or dynamically by first running and measuring the program several times, or by combinations of those. 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 14. If $\pi_1 = \ldots = \pi_k$ and $c_1 = \ldots = c_k$ then the solution to the problem above is the BTT of minimal size.

Our current multi-transition to BTT algorithm is exponential in the number of atomic predicates; it simply enumerates all possible BTTs recursively and then selects the one of minimal size. Generating the BTTs takes significantly less time than generating the MT-FSM, so we do not regard it as a practical problem yet. However, it seems to be a challenging theoretical problem.

Example 15. Consider again the traffic light controller safety property stating that "after green comes yellow", which was written as `[](green -> (!red U yellow))` using LTL notation. Since more than one light can be lit at any moment, one should be very careful when expressing this safety property as an MT-FSM or a BTT-FSM.

Let us first express it as an MT-FSM. We need two states, one for the case in which green has not triggered yet the `!red U yellow` part and another for the case when it has. The condition to stay in state 1 is then `yellow \/ !green` and the condition to move to state 2 is `!yellow /\ green /\ !red`. If both a `green` and a `red` are seen then the machine should move to state *false*. The condition to move from state 2 back to state 1 is `yellow`,

while the condition to stay in state 2 is `!yellow /\ !red`; `!yellow /\ red` should also move the machine in its *false* state. If the event is terminal then a `yellow` would make the reported answer *true*, i.e., the observed trace is an accepting sequence; if `yellow` is not true, then in state 1 the answer should be the opposite of `green`, while in state 2 the answer should be simply *false*. This MT-FSM is shown in figure 3.

The interesting aspect of our FSMs is that not all the atomic predicates need to always be evaluated. For example, in state 2 of this MT-FSM the predicate `green` is not needed. A BTT-FSM further reduces the amount of predicates to be evaluated, by enforcing an "evaluate-by-need" policy. Figure 4 shows a BTT-FSM implementing the MT-FSM in figure 3.

7.3. From LTL Formulae to BTT-FSMs

Informally, our algorithm to generate optimal BTT-FSMs from LTL formulae consists of two steps. First, it generates an MT-FSM with a minimum number of states. Then, using the technique presented in the previous subsection, it generates an optimal BTT from each MT.

State	MT for non-terminal events	MT for terminal events
1	[`yellow \/ !green` ? 1, `!yellow /\ green /\ !red` ? 2, `!yellow /\ green /\ red` ? *false*]	[`yellow \/ !green` ? *true*, `!yellow /\ green` ? *false*]
2	[`yellow` ? 1, `!yellow /\ !red` ? 2, `!yellow /\ red` ? *false*]	[`yellow` ? *true*, `!yellow` ? *false*]

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

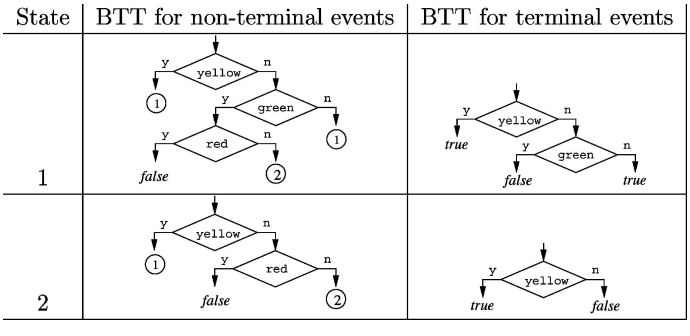

Figure 4. A BTT-FSM for the formula `[](green -> !red U yellow)`.

```
1. let S be φ
2. procedure LTL2MT-FSM(φ)
3.     let μ*(φ) be ∅
4.     let μ(φ) be ∅
5.     foreach θ : A → {true, false} do
6.       let e_θ be the list of atoms a with θ(a) = true
7.       let p_θ be the proposition ⋀{a | θ(a) = true} ∧ ⋀{¬a | θ(a) = false}
8.       let μ*(φ) be MERGE([p_θ ? φ{e*_θ}], μ*(φ))
9.       let φ_θ be φ{e_θ}
10.      if there is φ' ∈ S with VALID(φ_θ ↔ φ')
11.      then let μ(φ) be MERGE([p_θ ? φ'], μ(φ))
12.      else let S be S ∪ {φ_θ}
13.           let μ(φ) be MERGE([p_θ ? φ_θ], μ(φ))
14.           LTL2MT-FSM(φ_θ)
15.    endfor
16.    if μ(φ) = [true ? φ] and μ*(φ) = [true ? b] then replace φ by b everywhere
17. endprocedure
```

Figure 5. Algorithm to generate a minimal MT-FSM* $(S, A, \mu, \mu^\star, \varphi)$ from an LTL formula φ.

To generate a minimal MT-FSM, our algorithm uses the rewriting based procedure presented in Section 6 on all possible events, until the set of formulae to which the original LTL formula can "evolve" stabilizes. The procedure LTL2MT-FSM shown in figure 5 builds an MT-FSM whose states are formulae, with the help of a validity checker. Initially, the set S of states contains only the original LTL formula. LTL2MT-FSM is called on the original LTL formula, and then recursively in a depth-first manner on all the formulae to which the initial formula can ever evolve via the event-consuming operator $\_\{\_\}$ introduced in Section 6.

For each LTL state formula φ in S, multi-transitions $\mu(\varphi)$ and $\mu^\star(\varphi)$ are maintained. For each possible event θ, one first updates $\mu^\star(\varphi)$ by considering the case in which θ is the last event in a trace (step 8), and then the current formula φ evolves into the corresponding formula φ_θ (step 9). If some equivalent formula φ' to φ_θ has already been discovered then one only needs to modify the multi-transition set of φ accordingly in order to point to φ' (step 11). Notice that equivalence of LTL formulae is checked by using a validity procedure (step 10), which is given in figure 6. If there is no formula in S equivalent to φ_θ, then the new formula φ_θ is added to S, multi-transition $\mu(\varphi)$ is updated accordingly, and then the MT-FSM generation procedure is called recursively on φ_θ. This way, one eventually generates all possible LTL formulae into which the initial formula can ever evolve during a monitoring session; this happens, of course, modulo finite trace LTL semantic equivalence, implemented elegantly using the validity checker described below. By Theorem 2, this recursion will eventually terminate, leading to an MT-FSM*.

The procedure VALID used in LTL2MT-FSM above is defined in figure 6. It essentially follows the same idea of generating all possible formulae to which the original LTL formula tested for validity can evolve via the event consumption operator defined by rewriting in Section 6, but for each newly obtained formula φ and for each event θ, it also checks whether an execution trace stopping at that event would be a rejecting sequence. The intuition for this check is that a formula is valid under finite trace LTL semantics if and only if any (finite)

```
1. let S be φ
2. function VALID(φ)
3.    foreach θ : A → {true, false} do
4.       let e_θ be the list of atoms a with θ(a) = true
5.       if φ{e*_θ} = false then return false
6.       let φ_θ be φ{e_θ}
7.       if φ_θ ∉ S
8.       then let S be S ∪ {φ_θ}
9.            if VALID(φ_θ) = false then return false
10.   endfor
11.   return true
12. end function
```

Figure 6. Validity checker for an LTL formula φ.

sequence of events satisfies that formula; since any generated formula corresponds to one into which the initial formula can evolve, we need to make sure that each of these formulae becomes *true* under any possible last monitored event. This is done by rewriting the term $\varphi\{e^\star_\theta\}$ with the rules in Section 6 to its normal form (step 5.), i.e., *true* or *false*. The formula is valid if and only if there is no rejecting sequence; the entire space of evolving formulae is again explored by depth-first search. Notice that VALID does not test for equivalence of formulae, so it can potentially generate a larger number of formulae than LTL2MT-FSM. However, by Theorem 2, this procedure will also eventually terminate.

Theorem 16. *Given an LTL formula φ of size m, the following hold:*

1. *The procedure* VALID *is correct, that is*, VALID(φ) *returns true if and only if φ is satisfied, as defined in Section 4, by any finite trace*;
2. *The space and time complexity of* VALID(φ) *is* $2^{O(2^m)}$;

Additionally, letting M denote the MT-FSM $(S, A, \mu, \mu^\star, \varphi)$ generated by* LTL2MT-FSM(φ), *the following hold*:

3. LTL2MT-FSM(φ) *is correct; more precisely, M has the property that for any events $\theta_1, \ldots, \theta_n, \theta$, it is the case that $\varphi \xrightarrow{\theta_1} \varphi_1 \cdots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta^\star}$ true if and only if the finite trace $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ satisfies φ, and $\varphi \xrightarrow{\theta_1} \varphi_1 \cdots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta}$ false if and only if $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ does not satisfy φ;*
4. *M is synchronous; more precisely, for any events $\theta_1, \ldots, \theta_n$, it is the case that $\varphi \xrightarrow{\theta_1} \varphi_1 \cdots \xrightarrow{\theta_n} \varphi_n \xrightarrow{e_\theta}$ true if and only if $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ is a valid prefix of φ, and $\varphi \xrightarrow{\theta_1} \varphi_1 \cdots \xrightarrow{\theta_n} \varphi_n \xrightarrow{e_\theta}$ false if and only if $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ is a bad prefix of φ;*
5. *The space and time complexity of* LTL2MT-FSM(φ) *is* $2^{O(2^m)}$;
6. *M is the smallest MT-FSM* which is correct and synchronous (as defined in* 3 *and* 4 *above)*;
7. *Monitoring against a BTT-FSM* corresponding to M, as shown in Section* 7.2*, needs $O(2^m)$ space and time.*

Proof:

1. Using a depth-first search strategy, the procedure VALID visits all possible formulae into which the original formula φ can evolve via any possible sequence of events. These formulae are different modulo the rewriting rules in Section 6 which are used to simplify LTL formulae and to remove the derivatives, but those rules were shown to be sound, so VALID indeed explores all possible LTL formulae into which φ can *semantically* evolve. Moreover, for each such formula, say φ', and each event, VALID checks at Step 5 whether a trace terminating with that event in φ' would lead to rejection. VALID(φ) returns true if and only if there is no such rejection, so it is indeed correct: if it had been some finite trace that did not satisfy φ then VALID would have found it during its exhaustive search.
2. The space required by VALID(φ) is clearly dominated by the size of S. By Theorem 2, each formula φ' into which φ can evolve needs $O(2^m)$ space to be stored. That means that there can be at most $2^{O(2^m)}$ such formulae. So the total space needed to store S in the worst case is $2^{O(2^m)}$. An amortized analysis of its running time, tells us that VALID runs its "for each event" loop one per formula in S. Since the number of events is much less than 2^m and since reducing the derivative of a formula by an event takes time proportional with the size of the formula, we deduce that the total running time of the VALID procedure is $2^{O(2^m)}$.
3. By Theorem 3 and the event consuming procedure described in Section 6.1, it follows that $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ satisfies φ if and only if $\varphi\{e_{\theta_1}\}\{\cdots\}\{e_{\theta_n}\}\{e_\theta^\star\}$ reduces to *true*, and that $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ does not satisfy φ if and only if $\varphi\{e_{\theta_1}\}\{\cdots\}\{e_{\theta_n}\}\{e_\theta^\star\}$ reduces to *false*. It is easy to see that VALID($\psi \leftrightarrow \psi'$) returns true if and only if ψ and ψ' are formulae equivalent under the finite trace LTL semantics. That means the MT-FSM* generated by LTL2MT-FSM(φ) contains a formula-state φ_1 which is equivalent to $\varphi\{e_{\theta_1}\}$; moreover, $\varphi \xrightarrow{\theta_1} \varphi_1$ in this MT-FSM*. Inductively, one can show that there is a series of formulae-states $\varphi_1, \ldots, \varphi_n$ such that $\varphi \xrightarrow{\theta_1} \varphi_1 \cdots \xrightarrow{\theta_n} \varphi_n$ is a sequence of transitions in the generated MT-FSM and $\varphi\{e_{\theta_1}\}\{\cdots\}\{e_{\theta_n}\}$ is equivalent to φ_n. The rest follows by noting that $\varphi_n\{e_\theta^\star\}$ reduces to true if and only if $\varphi_n \xrightarrow{\theta^\star} \mathit{true}$ in the generated MT-FSM, and that $\varphi_n\{e_\theta^\star\}$ reduces to false if and only if $\varphi_n \xrightarrow{\theta^\star} \mathit{false}$ in the MT-FSM.
4. As in *3* above, one can inductively show that there is a series of formulae $\varphi_1, \ldots, \varphi_n, \varphi'$ such that $\varphi \xrightarrow{\theta_1} \varphi_1 \cdots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta} \varphi'$ is a sequence of transitions in the generated MT-FSM and $\varphi\{e_{\theta_1}\}\{\cdots\}\{e_{\theta_n}\}\{e_\theta\}$ is equivalent to φ'. By Proposition 13, due to the use of the validity checker in step 10 of LTL2MT-FSM it follows that $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ is a valid prefix of φ if and only if φ' is equivalent to *true*, and that $e_{\theta_1} \ldots e_{\theta_n} e_\theta$ is a bad prefix of φ if and only if φ' is equivalent to *false*. The rest follows now by Step 16 in the algorithm in figure 5, which, due to the validity checker, provides a necessary and sufficient condition for a processed formula to be semantically equivalent to *true* or *false*, respectively.
5. The space required by LTL2MT-FSM(φ) is again dominated by the size of S. Like in the analysis of VALID in *2* above, by Theorem 2 we get that there can be at most $2^{O(2^m)}$ formulae generated in S, so the total space needed to store S in the worst case is also $2^{O(2^m)}$. For each newly added formula in S and for each event, Step 10 calls the procedure VALID potentially once for each already existing formula in S. It is important to notice that the formulae on which VALID is called are exclusive disjunctions of conjunctions

of sub-formulae rooted in temporal operators of the original formula φ, so its space and time complexity will also be $2^{O(2^m)}$ each time it is called by LTL2MT-FSM. One can easily see now that the space and time requirements of LTL2MT-FSM(φ) are also $2^{O(2^m)}$ (the constant in the exponent $O(2^m)$ can be appropriately enlarged).

6. For any MT-FSM* machine $M' = (S', A, \mu', \mu'^{\star}, q_0)$, one can associate a formula, more precisely a state in M, to each state in S' as follows. φ is associated to the initial state q_0. Then for each transition $q_1 \xrightarrow{\theta} q_2$ in M' such that q_1 has a formula φ_1 associated and q_2 has no formula associated, then one associates that φ_2 to q_2 with the property that $\varphi_1 \xrightarrow{\theta} \varphi_2$ is a transition in M. For a state q in S' let φ_q be the formula in S associated to it. Since M' is correct and synchronous for φ, it follows that $\mathcal{L}_{M'}(q) = \mathcal{L}_M(\varphi_q)$ for any state q in S'. We can now show that the map associating a formula in S to any state in S' is surjective, which shows that M has therefore a minimal number of states. Let φ' be a formula in S and let $\varphi \xrightarrow{\theta_1} \varphi_1 \cdots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta} \varphi'$ be a sequence of transitions in M leading to φ'; note that all formulae in S are reachable via transitions in M from φ. Let q' be the state in S' such that $q_0 \xrightarrow{\theta_1} q_1 \cdots \xrightarrow{\theta_n} q_n \xrightarrow{\theta} q'$. Then $\mathcal{L}_{M'}(q') = \mathcal{L}_M(\varphi_{q'})$. Since by Proposition 7.1, $\mathcal{L}_{M'}(q') = \{t \mid e_{\theta_1} \ldots e_{\theta_n} e_\theta t \in \mathcal{L}_{M'}(q_0)\}$ and $\mathcal{L}_M(\varphi') = \{t \mid e_{\theta_1} \ldots e_{\theta_n} e_\theta t \in \mathcal{L}_M(\varphi)\}$, it follows that $\mathcal{L}_M(\varphi_{q'}) = \mathcal{L}_M(\varphi')$, that is, that $\varphi_{q'}$ and φ' are equivalent. Since Step 10 of the LTL2MT-FSM eventually uses the validity checker on any pairs of formulae in S, it follows that $\varphi_{q'} = \varphi'$.
7. In order to distinguish N pieces of data, $\log(N)$ bits are needed to encode each datum. Therefore, one needs $O(2^m)$ bits to encode a state of M, which is the main memory needed by the monitoring algorithm. Like in Theorem 2.1, we assume "idealistic" rewriting engines able to optimize the space requirements; we are not aware whether Maude is able to attain this performance or not. To make a transition from one state to another, a BTT-FSM* associated to the MT-FSM* generated by LTL2MT-FSM(φ) needs to only evaluate at most all the atomic predicates occurring in the formula, which, assuming that evaluating atom predicates takes unit time, is clearly $O(2^m)$. When the entire BTT is evaluated, it finally has to store the newly obtained state of the BTT-FSM*, which can reuse the space of the previous one, $O(2^m)$, and which also takes time $O(2^m)$.

□

Once the procedure LTL2MT-FSM terminates, the formulae φ, φ', etc., are not needed anymore, so one can and should replace them by unique labels in order to reduce the amount of storage needed to encode the MT-FSM* and/or the corresponding BTT-FSM*. This algorithm can be relatively easily implemented in any programming language. We have, however, found Maude again a very elegant system for this task, implementing the entire LTL formula to BTT-FSM algorithm in about 200 lines of code.

7.4. Examples

The BTT-FSM generation algorithm presented in this section, despite its overall worst-case high startup time, can be very useful when formulae are relatively short, as it is most

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

often the case in practice. For the traffic light controller requirement formula discussed previously in the paper, `[](green -> (!red) U yellow)`, this algorithm generates in about 0.2 seconds the optimal BTT-FSM* in figure 7, also shown in figure 4 in flowchart notation; Figure 3 shows its optimal MT-FSM*. For simplicity, the states *true* and *false* do not appear in figure 7. 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 Maude implementation of this algorithm 1.3 seconds to generate the minimal 3-state (*true* and *false* states are not counted) BTT-FSM* for the formula `a U (b U (c U d))` and 13.2 seconds to generate the 7-state minimal BTT-FSM* for the formula `((a U b) U c) U d`. It never took our current implementation more than a few seconds to generate the BTT-FSM* of any LTL formula of interest for our applications, i.e., non-artificial. Figure 8 shows the generated BTT-FSM of some artificial LTL formulae, taking together less than 15 seconds to be generated. To keep the figure small, the states *true* and *false* together with their self-transitions are not shown in figure 8, and they are replaced by `t` and `f` in BTTs.

The generated BTT-FSM*s are monitored most efficiently on RAM machines, due to the fact that conditional statements are implemented via jumps in memory. Monitoring BTT-FSM*s using rewriting does not seem appropriate because it would require linear time, as a function of number of states, to extract the BTT associated to a state in a BTT-FSM*.

Formula	State	Monitoring BTT	Terminating BTT
$[]\langle\rangle\, a$	1	1	a ? t : f
$\langle\rangle([]a \vee []\neg a)$	–	–	–
$[](a \texttt{ -> } \langle\rangle b)$	1	a ? (b ? 1 : 2) : 1	a ? (b ? t : f) : t
	2	b ? 1 : 2	b ? t : f
a U (b U c)	1	c ? t : (a ? 1 : (b ? 2 : f))	c ? t : f
	2	c ? t : (b ? 2 : f)	c ? t : f
a U (b U (c U d))	1	d ? t : a ? 1 : b ? 2 : c ? 3 : f	d ? t : f
	2	d ? t : b ? 2 : c ? 3 : f	d ? t : f
	3	d ? t : c ? 3 : f	d ? t : f
((a U b) U c) U d	1	d ? t : c ? 1 : b ? 4 : a ? 5 : f	d ? t : f
	2	b ? c ? t : 7 : a ? c ? 6 : 2 : f	c ? b ? t : f : f
	3	b ? d ? t : c ? 1 : 4 : a ? d ? 6 : c ? 3 : 5 : f	d ? b ? t : f : f
	4	c ? d ? t : 1 : b ? d ? 7 : 4 : a ? d ? 2 : 5 : f	d ? c ? t : f : f
	5	b ? d ? c ? t : 7 : c ? 1 : 4 : a ? d ? c ? 6 : 2 : c ? 3 : 5 : f	d ? c ? b ? t : f : f : f
	6	b ? t : a ? 6 : f	b ? t : f
	7	c ? t : b ? 7 : a ? 2 : f	c ? t : f

Figure 8. Six BTT-FSM*s generated in less than 15 seconds.

However, we believe that the algorithm presented in Section 6 is satisfactory in practice if one is willing to use a rewriting engine for monitoring.

8. Conclusions

This paper presented a foundational study in using rewriting in runtime verification and monitoring of systems. After a short discussion on types of monitoring and mathematical and technological preliminaries, a finite trace linear temporal logic was defined, together with an immediate but inefficient implementation of a monitor following directly its semantics. Then an efficient but ineffective implementation based on dynamic programming was presented, which traverses the execution trace backwards. The first effective and relatively efficient rewriting algorithm was further introduced, based on the idea of transforming the monitoring requirements as events are received from the monitored program. A non-trivial improvement of this algorithm based on hashing rewriting results, thereby reducing the number of rewritings performed during trace analysis, was also proposed. 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. Since in many cases one would want to generate an observer finite state machine (or automaton) a priori, for example when a rewriting system cannot be used for monitoring, or when minimal runtime overhead is needed, a specialized data-structure called a binary transition tree (BTT) and corresponding finite state machines were introduced, and an algorithm for generating minimal such monitors from temporal formulae was discussed.

All algorithms were implemented 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 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. Some of the discussed results and algorithms have been already used in two NASA applications, JPAX and X9, but an extensive experimental assesment of these techniques is left as future work.

Acknowledgment

The authors warmly thank the anonymous reviewers for their very detailed and useful comments and suggestions on how to improve this paper.

Notes

1. Personal communication by José Meseguer.
2. Underscores are places for arguments.

3. The lower the precedence number, the tighter the binding.
4. Either by typing it or using the command "`in <filename>`".
5. On a 1.7 GHz, 1 Gb memory PC.
6. In fact, JPAX reports a similar message also when the current monitoring requirement becomes `true` or `false` at any time during the monitoring process.

References

Artho, C., Drusinsky, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Roşu, G., and Visser, W. 2003. Experiments with test case generation and runtime analysis. In *Proc. of ASM'03: Abstract State Machines*, Vol. 2589 of *Lecture Notes in Computer Science*, Taormina, Italy, Springer, pp. 87–107.

Ball, T., Podelski, A., and Rajamani, S. 2001. Boolean and cartesian abstractions for model checking c programs. In *Proc. of TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Genova, Italy.

Bouhoula, A., Jouannaud, J.-P., and Meseguer, J. 2000. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132.

Bryant, R.E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

Chen, F. and Roşu, G. 2003. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, Vol. 89 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Boulder, Colorado, USA, pp. 106–125.

Clavel, M. 2001. The ITP tool. In *Logic, Language and Information. Proc. of the First Workshop on Logic and Language*, Kronos, pp. 55–62.

Clavel, M., Durán, F.J., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada., J.F. 1999. Maude: Specification and programming in rewriting logic, Maude System documentation at `http://maude.csl.sri.com/papers`.

Clavel, M., Durán, F.J., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada., J.F. 2002. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243.

Corbett, J., Dwyer, M.B., Hatcliff, J., Pasareanu, C.S., Robby, Laubach S., and Zheng., H. 2000. Bandera: Extracting finite-state models from java source code. In *Proc. of ICSE'00: International Conference on Software Engineering*, Limerich, Ireland, ACM Press.

Dahm., M. BCEL. `+http://jakarta.apache.org/bcel+.`

Demartini, C., Iosif, R., and Sisto., R. 1999. A deadlock detection tool for concurrent java programs. *Software Practice and Experience*, 29(7):577–603.

Drusinsky, D. 2000. The temporal rover and the atg rover. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, Vol. 1885 of *Lecture Notes in Computer Science*, Springer Stanford, California, USA, pp. 323–330.

Drusinsky, D. 2003. Monitoring temporal rules combined with time series. In *Proc. of CAV'03: Computer Aided Verification*, Vol. 2725 of *Lecture Notes in Computer Science*, Springer-Verlag, Boulder, Colorado, USA, pp. 114–118.

Fidge, C.J. 1988. Partial orders for parallel debugging. In *Proc. of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*, ACM, pp. 183–194.

Finkbeiner, B., Sankaranarayanan, S., and Sipma, H. 2002. Collecting statistics over runtime executions. In *Proc. of RV'02: The Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, Elsevier, Paris, France.

Finkbeiner, B. and Sipma, H. 2001. Checking finite traces using alternating automata. In *Proc. of RV'01: The First International Workshop on Runtime Verification*, Vol. 55(2) of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Paris, France.

Giannakopoulou, D. and Havelund, K. 2001. Automata-based verification of temporal properties on running programs. In *Proc. of ASE'01: International Conference on Automated Software Engineering*, Institute of Electrical and Electronics Engineers, Coronado Island, California, pp. 412–416.

Godefroid, P. 1997. Model checking for programming languages using verisoft. In *Proc. of POPL'97: the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, pp. 174–186.

Goguen, J., Lin, K., Roşu, G., Mori, A., and Warinschi, B. 2000. An overview of the Tatami project. In *Cafe: An Industrial-Strength Algebraic Formal Method*, Elsevier, pp. 61–78.

Goguen, J., Thatcher, J., Wagner, E., and Wright, J. 1977. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95.

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

Gunter, E. and Peled, D. 2002. Tracing the executions of concurrent programs. In *Proc. of RV'02: Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, Elsevier, Copenhagen, Denmark.

Gunter, E.L., Kurshan, R.P., and Peled, D. 2003. PET: An interactive software testing tool. In *Proc. of CAV'00: Computer Aided Verification*, Vol. 1885 of *Lecture Notes in Computer Science*, Springer-Verlag, Chicago, Illinois, USA, pp. 552–556.

Gunter, E.L. and Peled, D. 2000. Using functional languages in formal methods: The PET system. In *Parallel and Distributed Processing Techniques and Applications*, CSREA, pp. 2981–2986.

Havelund, K., Johnson, S., and Roşu, G. 2001. Specification and error pattern based program monitoring. In *Proc. of the European Space Agency workshop on On-Board Autonomy*, Noordwijk, The Netherlands.

Havelund, K., Lowry, M.R., and Penix, J. 2001. Formal analysis of a space craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, An earlier version occurred in the Proc. of SPIN'98: the fourth SPIN workshop, Paris, France, 1998.

Havelund, K. and Pressburger, T. 2000. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Special issue containing selected submissions to SPIN'98: the fourth SPIN workshop, Paris, France, 1998.

Havelund, K. and Roşu, G. 2001. Java PathExplorer—A Runtime verification tool. In *Proc. of i-SAIRAS'01: the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Montreal, Canada.

Havelund, K. and Roşu, G. 2001. Monitoring Java programs with java pathexplorer. In *Proc. of RV'01: the First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, Paris, France, Elsevier Science, pp. 97–114.

Havelund, K. and Roşu, G. 2001. Monitoring programs using rewriting. In *Proc. of ASE'01: International Conference on Automated Software Engineering*, Institute of Electrical and Electronics Engineers, Coronado Island, California, USA, pp. 135–143.

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

Havelund, K. and Roşu, G. to appear Efficient monitoring of safety properties. *Software Tools and Technology Transfer*.

Havelund, K. and Roşu, G. 2002. Synthesizing monitors for safety properties. In *Proc. of TACAS'02: Tools and Algorithms for Construction and Analysis of Systems*, Vol. 2280 of *Lecture Notes in Computer Science*, Springer, Grenoble, France, pp. 342–356.

Havelund, K. and Shankar, N. 1996. Experiments in theorem proving and model checking for protocol verification. In *Proc. of FME'96: Industrial Benefit and Advances in Formal Methods*, Vol. 1051 of *Lecture Notes in Computer Science*, Springer, Oxford, England, pp. 662–681

Holzmann, G.J. and Smith, M.H. 1999. A practical method for verifying event-driven software. In *Proc. of ICSE'99: International Conference on Software Engineering*, IEEE/ACM, Los Angeles, California, USA.

Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusetts: Addison-Wesley.

Hsiang, J. 1985. Refutational theorem proving using term rewriting systems. *Artificial Intelligence*, 25:255–300.

Kim, M., Kannan, S., Lee, I., and Sokolsky, O. 2001. Java-MaC: A Run-time assurance tool for Java. In *Proc. of RV'01: First International Workshop on Runtime Verification*, Vol. 55 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Paris, France.

Kortenkamp, D., Milam, T., Simmons, R., and Fernandez, J. 2001. Collecting and analyzing data from distributed control programs. In *Proc. of RV'01: First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Paris, France.

Kupferman, O. and Vardi, M.Y. 1998. Freedom, weakness, and determinism: From linear-time to branching-Time. In *Proc. of the IEEE Symposium on Logic in Computer Science*, pp. 81–92.

Kupferman, O. and Vardi, M.Y. 1999. Model checking of safety properties. In *Proc. of CAV'99: Conference on Computer-Aided Verification*, Trento, Italy.

Kupferman, O. and Zuhovitzky, S. 2002. An improved algorithm for the membership problem for extended regular expressions. In *Proc. of the International Symposium on Mathematical Foundations of Computer Science*, Vol. 2420 of *Lecture Notes in Computer Science*.

Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M. 1999. Runtime assurance based on formal specifications. In *Proc. of PDPTA'99: International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA.

Manna, Z. and Pnueli, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. New York: Springer.

Manna, Z. and Pnueli, 1995. *Temporal Verification of Reactive Systems: Safety*. New York: Springer.

Markey, N. and Schnoebelen, P. 2003. Model checking a path (Preliminary Report). In *Proc. of CONCUR'03: International Conference on Concurrency Theory*, Vol. 2761 of *Lecture Notes in Computer Science*, Springer, Marseille, France, pp. 251–265.

Meseguer, J. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 73–155.

Meseguer, J. 1998. Membership algebra as a logical framework for equational specification. In *Proc. of WADT'97: Workshop on Algebraic Development Techniques*, Vol. 1376 of *Lecture Notes in Computer Science*, Springer, Tarquinia, Italy, pp. 18–61.

O'Malley, T., Richardson, D., and Dillon, L. 1996. Efficient specification-based oracles for critical systems. In *Proc. of the California Software Symposium*.

Park, D.Y., Stern, U., and Dill, D.L. 2000. Java model checking. In *Proc. of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland.

Pnueli, A. 1977. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, pp. 46–77.

Richardson, D.J., Aha, S.L., and O'Malley, T.O. 1992. Specification-based test oracles for reactive systems. In *Proc. of ICSE'92: International Conference on Software Engineering*, Melbourne, Australia, pp. 105–118.

Roşu, G. and Havelund, K. 2001. Synthesizing dynamic programming algorithms from linear temporal logic formulae. RIACS Technical report TR 01-08.

Roşu, G. and Viswanathan, M. 2003. Testing extended regular language membership incrementally by rewriting. In *Proc. of RTA'03: Rewriting Techniques and Applications*, Vol. 2706 of *Lecture Notes in Computer Science*, Valencia, Spain, Springer-Verlag, pp. 499–514.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411.

Sen, A. and Garg, V.K. 2003. Partial order trace analyzer (POTA) for distrubted programs. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, Vol. 89 of *Electronic Notes in Theoretical Computer Science*, Boulder, Elsevier Science, Colorado, USA.

Sen, K. and Roşu, G. 2003. Generating optimal monitors for extended regular expressions. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Boulder, Colorado, USA, pp. 162–181.

Sen, K., Roşu, G., and Agha, G. 2003. Runtime safety analysis of multithreaded programs. In *Proc. of ESEC/FSE'03: European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, Helsinki, Finland.

Shankar, N., Owre, S., and Rushby, J.M. 1993. *PVS Tutorial*. Computer science laboratory, SRI International, Menlo Park, CA, 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, Odense, Denmark, pp. 357–406.

Sistla, A.P. and Clarke, E.M. 1985. The Complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749.

Stoller, S.D. 2000. Model-checking multi-threaded distributed java programs. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, Vol. 1885 of *Lecture Notes in Computer Science*, , Springer, Stanford, California, USA, pp. 224–244.

Visser, W., Havelund, K., Brat, G., and Park, S. 2000. Model checking programs. In *Proc. of ASE'00: International Conference on Automated Software Engineering*, IEEE CS Press, Grenoble, France.

Yamamoto, H. 2000. An automata-based recognition algorithm for semi-extended regular expressions. In *Proc. of the International Symposium on Mathematical Foundations of Computer Science*, Vol. 1893 of *Lecture Notes in Computer Science*, pp. 699–708.