Monitoring Programs using Rewriting

Klaus Havelund
Kestrel Technology
http://ase.arc.nasa.gov/havelund
NASA Ames Research Center
Moffett Field, CA, 94035

Grigore Roşu
Research Institute for Advanced Computer Science
http://ase.arc.nasa.gov/grosu
NASA Ames Research Center
Moffett Field, CA, 94035

Abstract

*We present a rewriting algorithm for efficiently testing future time Linear Temporal Logic (LTL) formulae on finite execution traces. The standard models of LTL are infinite traces, reflecting the behavior of reactive and concurrent systems which conceptually may be continuously alive. In most past applications of LTL, theorem provers and model checkers have been used to formally prove that down-scaled models satisfy such LTL specifications. Our goal is instead to use LTL for up-scaled testing of real software applications, corresponding to analyzing the conformance of finite traces against LTL formulae. We first describe what it means for a finite trace to satisfy an LTL formula and then suggest an optimized algorithm based on transforming LTL formulae. We use the Maude rewriting logic, which turns out to be a good notation and being supported by an efficient rewriting engine for performing these experiments. The work constitutes part of the Java PathExplorer (*JPAX*) project, the purpose of which is to develop a flexible tool for monitoring Java program executions.*

1. Introduction

Future time Linear Temporal Logic (future time LTL), introduced by Pnueli in 1977 [23], is a logic for specifying temporal properties about reactive and concurrent systems. Future time LTL provides temporal operators that refer to the future/remaining part of a trace relative to a current point of reference. We shall use the shorthand LTL when it is clear from the context that we mean future time LTL. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being an example. LTL has typically been used for specifying properties of concurrent and interactive down-scaled models of real systems, such that fully formal program proofs could subsequently be carried out, for example using theorem provers [15] or model checkers [10]. However, such formal proof techniques are usually not scalable to real sized systems without an extra effort to abstract the system to a model which is then analyzed. Several systems are currently being developed that apply model checking to software [4, 16, 3, 22, 6, 27], including our work [11, 28]. In this paper we restrict ourselves to investigate the use of LTL for testing whether single finite execution traces conform to LTL formulae. The merge of testing and temporal logic specification is an attempt to achieve the benefits of both approaches, while avoiding some of the pitfalls from ad hoc testing and the complexity in full-blown theorem proving and model checking.

An important question is how to efficiently test LTL formulae of finite trace models, and the main decision here is what data structure one should use to represent the formula such that it can be used to efficiently analyze the trace as it is traversed. We will present such a data structure. We will present and implement our logics and algorithms in Maude [1], a high-performance system supporting both membership equational logic [20] and rewriting logic [19]. The current version of Maude can do up to 3 million rewritings per second on 800MHz processors, and its compiled version is intended to support 15 million rewritings per second[1]. The decision to use Maude has made it very easy to experiment with logics and algorithms. Later realizations of the work can be done in a standard programming language such as Java or C++. In [14] we have for example described a data structure used to represent an LTL formula as a minimal finite state machine, based on a concept called *binary transition trees*. This structure can then be represented and interpreted within Java. In [25] we furthermore describe a dynamic programming algorithm for checking LTL formulae on execution traces. This algorithm evaluates a formula bottom-up for each point in the trace, going backwards from the final state, towards the initial state. In [9] we apply this dynamic algorithm to past time LTL, in which case the trace more naturally can be examined in a forward direction. In that paper it is in addition shown how future time and past time LTL formulae can be embedded as comments in code and get expanded into Java code fragments to get executed whenever reached. In [24, 21] various algorithms to generate testing automata from temporal logic formulae are described. Our colleague Dimitra Giannakopoulou has also implemented a Büchi automata inspired algorithm adapted to finite trace LTL. The Maude rewriting implementation of LTL described in this paper, besides its simplicity and elegance, however offers a greater flexibility in experimenting with temporal logics and is quite efficient for practical purposes.

The work constitutes part of the Java PathExplorer (JPAX) tool [13, 14] for monitoring Java program executions. JPAX facilitates automated instrumentation of Java byte code, which then emits relevant events to an observer during execution, see Figure 1. The observer can be running a Maude process as a special case, hence Maude's rewriting engine can be used to drive a temporal logic operational semantics with program execution events. The observer may run on a different computer, in which case the events are transmitted over a socket. When the observer receives the events it dispatches these to a set of observer rules, each rule performing a particular analysis that has been requested. Observer rules are written in Java, but can call programs written in other languages, such as in this case Maude. In addition to checking temporal logic requirements, rules can also be programmed to perform low level error pattern analysis of, for example, multi-threaded programs, identifying error-prone programming practices, such as unhealthy locking disciplines that may lead to data races and/or deadlocks. The process is driven by a *specification*, written in Java, which, as in [18], consists of an *instrumentation* part and a *verification* part. The verification specification defines the high level requirements (as String values), usually written in temporal logic, that events are to be checked against. The propositions referred to in these requirements are abstract boolean flags, and do hence not refer directly to entities in the concrete program. The instrumentation specification establishes this connection between the concrete boolean program predicates and the abstract propositions. Hence the instrumentation specification in particular defines what program variables will be monitored. The instrumentation is automatic and is performed using the bytecode engineering tool Jtrek [2].

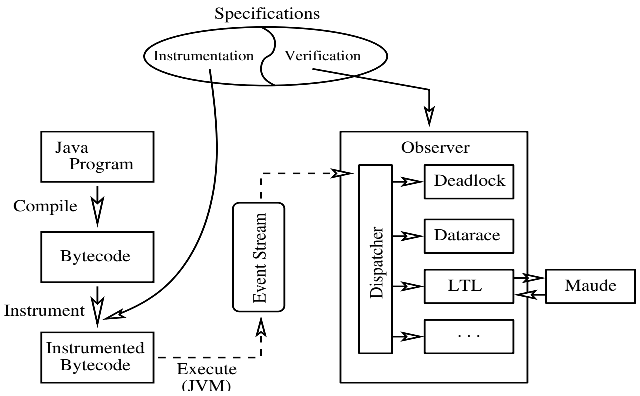

Figure 1. Overview of JPAX

The idea of using temporal logic in program testing is not new, and at our knowledge, has already been pursued in the commercial Temporal Rover tool (TR) [5], and in the MaC tool [18]. Both tools have greatly inspired our work. Our basic contribution in this paper is to show how a rewriting system, such as Maude, makes it possible to experiment with monitoring logics very fast and elegantly, and furthermore can be used as a practical program monitoring engine. This approach makes it possible to formalize ideas in a framework close to standard mathematics. The formula transforming approach suggested is a new and efficient way of testing LTL formulae. A previous version of the paper, published as a technical report [12], presents a simplified action based formalization of LTL rather than the state based more realistic framework presented here, which is the one currently implemented in JPAX. In [13] and [14] we describe a formalization of past time LTL (as well as future time LTL), again illustrating the succinctness of new logic definitions.

Section 2 contains preliminaries, including an introduction to Maude, propositional logic and the standard definition of propositional LTL with its infinite trace models. Section 3 presents a finite trace semantics for LTL and then its implementation in Maude. Although abstract and elegant, this implementation is not efficient, and Section 4 presents an efficient implementation using a formula transformation approach. Finally, Section 5 contains conclusions and a description of future work.

[1] Personal communication by José Meseguer.

2. Preliminaries

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

2.1. Maude and Logics for Program Monitoring

Maude [1] is a freely distributed high-performance system in the OBJ [8] algebraic specification family, supporting both rewriting logic [19] and membership equational logic [20]. Because of its efficient rewriting engine, able to execute 3 million rewriting steps per second on currently standard hardware configurations, and because of its metalanguage features, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, and even programming languages. We were delighted to notice how easily we could implement and efficiently validate our algorithms for testing LTL formulae on finite event traces in Maude, admittedly a tedious task in C++ or Java, and hence decided to use Maude at least for the prototyping stage of our runtime check algorithms.

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

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

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

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

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

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

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

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

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

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

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

The first block of declarations introduces the sort `Formula` which can be thought of as a generic sort for any well-

[2] Underscores are places for arguments.

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

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

2.2. Propositional Calculus

A rewriting decision procedure for propositional calculus due to Hsiang [17] is adapted and presented. It provides the usual connectives `_/\_` (and), `_++_` (exclusive or), `_\/_` (or), `!_` (negation), `_->_` (implication), and `_<->_`(equivalence). The procedure reduces tautology formulae to the constant `true` and all the others to some canonical form modulo associativity and commutativity. An unusual aspect of this procedure is that the canonical forms consist of exclusive or of conjunctions. Even if propositional calculus is very basic to almost any logical environment, we decided to keep it as a separate logic instead of being part of the logic infrastructure of JPAX. One reason for this decision is that its semantics could be in conflict with other logics, for example ones in which conjunctive normal forms are desired.

An OBJ3 code for this procedure appeared in [8]. Below we give its obvious translation to Maude together with its finite trace semantics, noticing that Hsiang [17] showed that this rewriting system modulo associativity and commutativity is Church-Rosser and terminates. The Maude team was probably also inspired by this procedure, since the builtin `BOOL` module is very similar.

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

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

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

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

2.3. Linear Temporal Logic

Classical LTL provides in addition to the propositional logic operators the temporal operators `[]_` (always), `<>_` (eventually), `_U_` (until), and `o_` (next). An LTL standard model is a function $t : \mathcal{N}^+ \rightarrow 2^{\mathcal{P}}$ for some set of atomic propositions $\mathcal{P}$, i.e., an infinite trace over the alphabet $2^{\mathcal{P}}$, which maps each time point (a natural number) into the set of propositions that hold at that point. The operators have the following interpretation on such an infinite trace. Assume formulae `X` and `Y`. The formula `[]X` holds if `X` holds in all time points, while `<>X` holds if `X` holds in some future time point. The formula `X U Y` (`X` until `Y`) holds if `Y` holds

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

in some future time point, and until then X holds (so we consider strict until). Finally, `o X` holds for a trace if X holds in the suffix trace starting in the next (the second) time point. The propositional operators have their obvious meaning. As an example illustrating the semantics, the formula `[](X -> <>Y)` is true if for any time point (`[]`) it holds that if X is true then eventually (<>) Y is true. Another similar property is `[](X -> o(Y U Z))`, which states that whenever X holds then from the next state Y holds until eventually Z holds. It's standard to define a core LTL using only atomic propositions, the propositional operators `!_` (not) and `_/\_` (and), and the temporal operators `o_` and `_U_`, and then define all other propositional and temporal operators as derived constructs. Standard equations are `<>X = true U X` and `[]X = !<>!X`.

3. Finite Trace Linear Temporal Logic

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

3.1. Finite Trace Semantics

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

$$
\begin{array}{lll}
t \models \mathtt{A} & \text{iff} & \mathtt{A} \in \mathit{head}(t) \\
t \models \mathtt{true} & \text{iff} & \mathit{true}, \\
t \models \mathtt{false} & \text{iff} & \mathit{false}, \\
t \models \mathtt{X\ /\backslash\ Y} & \text{iff} & t \models \mathtt{X} \text{ and } t \models \mathtt{Y}, \\
t \models \mathtt{X\ ++\ Y} & \text{iff} & t \models \mathtt{X} \text{ xor } t \models \mathtt{Y}, \\
t \models \mathtt{[]X} & \text{iff} & (\forall\, i \leq length(t))\ t_i \models \mathtt{X} \\
t \models \mathtt{<>X} & \text{iff} & (\exists\, i \leq length(t))\ t_i \models \mathtt{X} \\
t \models \mathtt{X\ U\ Y} & \text{iff} & (\exists\, i \leq length(t))\ (t_i \models \mathtt{Y} \text{ and } (\forall\, j < i)\ t_j \models \mathtt{X}) \\
t \models \mathtt{o\ X} & \text{iff} & (\text{if } \mathit{tail}(t) \text{ is defined then } tail(t) \models \mathtt{X} \text{ else } t \models \mathtt{X})
\end{array}
$$

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

3.2. Finite Trace Semantics in Maude

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

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

Notice that only the temporal operators needed declarations and semantics, the others being already defined in `PROP-CALC` and `LOGICS-BASIC`, and that the definitions that involved the functions *head* and *tail* were replaced by two alternative equations. One can now directly verify LTL properties on finite traces using Maude's rewriting engine, by commands such as

```
red a b, a, c a, a b, c b, a b, a, c a, a b, c b
   |= [] (a -> <> b) .
red a b, a, c a, a b, c b, a b, a, c a, a b, c b
   |= <> (! [](a -> <> b)) .
```

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

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

4. An Efficient Rewriting Algorithm

In this section we shall present a more efficient rewriting semantics for LTL, based on the idea of consuming the events in the trace, one by one, and updating a data structure (which is also a formula) corresponding to the effect of the event on the value of the formula. 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 the most natural. As it turns out, it also yields a more efficient rewriting system.

4.1. The Main Algorithm

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

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

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

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

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

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

procedure turns out to be still too slow, then one can statically generate all formulae in which a formula can transform and store them as the states of an automaton, the edges being the possible events. Then when a new event is generated by the monitored program, one could directly go to the "next" state of the automaton without any logical reasoning. We have implemented an improved version of such a procedure (in which only a minimal subset of atomic propositions are evaluated); details regarding this implementation will appear elsewhere, but an informal description can be found in [14].

Verification results are very encouraging and show that this optimized semantics is orders of magnitudes faster than the first semantics. Traces of less than 10,000 events are verified in milliseconds, while traces of 100,000 events never needed more than 3 seconds. This technique scales quite well; we were able to monitor even traces of hundreds of millions events. As a concrete example, we created an artificial trace by repeating 10 million times the 10 event trace `a b, a, c a, a b, c b, a b, a, c a, a b, c b`, and then checked it against the formula `[](a -> <> b)`. There were needed 4.9 billion rewriting steps for a total of about 1,500 seconds.

4.2. Correctness and Completeness

In this subsection we prove that the algorithm presented above is correct and complete with respect to the semantics of finite trace LTL presented in Section 3. The proof is done completely in Maude, but since Maude is not intended to be a theorem prover, we actually have to generate the proof obligations by hand. However, the proof obligations below could be automatically generated by a proof assistant like KUMO [7] or a theorem prover like PVS [26][6].

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

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

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

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

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

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

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

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

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

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

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

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

$\mathcal{P}$(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

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

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

5. Conclusions and Future Work

We have presented a finite trace semantics of LTL in the Maude logic together with a much more efficient version based on formula transforming state changes. The formula transformation approach can be regarded as a self contained result with interest to at least the rewriting and temporal logics communities. However, what perhaps makes it more interesting is that its integration into the general program monitoring framework JPAX seems to be quite efficient for practical purposes, allowing an elegant flexibility in the choice and design of requirement languages. This can be useful not only for research projects and educational purposes, but also for real-life projects, where requirement languages may be domain or application specific. In principle what Maude provides is a static parsing environment for defining syntax, combined with a rewrite-based dynamic execution environment for defining efficient semantics over the parse trees. It is our goal to examine the feasibility of this approach on a selection of NASA software systems.

A current research activity is, however, to find yet more efficient representations of future time LTL formula for the purpose of achieving an absolute optimal algorithm for testing their satisfaction on execution traces. This becomes especially crucial for an implementation in a standard programming language such as Java. In [14] we describe such a provably minimal finite state machine representation. An efficient dynamic programming algorithm is furthermore described in [25], although it examines the trace backwards, requiring the trace to be stored. As it turns out, this algorithm applies more naturally to the checking of past time LTL, since this can be done by a forward examination of the trace. Of future work can be mentioned that we will experiment with new logics in Maude, such as interval and real time logics and UML notations. We have already in [13, 14] described how past time LTL can be succinctly defined in Maude (note that this work is different from the dynamic programming algorithm for past time LTL just mentioned).

A general question is how such requirement monitoring can be tied into the different levels of the development cycle. With the composition of the specification into a verification part that refers to abstract propositions, and an instrumentation part that relate these to concrete program entities, the verification part can potentially be written before the program is developed. Hence, Maude can be used during the early phases of a project to write down requirements, which can then can be tested later in the implementation phase. One can further imagine that the requirements written in Maude themselves can be subject to various forms of formal analysis, also programmed in Maude, such as rapid prototyping, symbolic simulation, static analysis, theorem proving, and model checking. We have for example considered formulating state machines in Maude and use these for monitoring. Such state machines are obvious candidates for the above mentioned forms of analysis.

As described in [13, 14] JPAX provides in addition to specification based monitoring also a capability of checking error patterns in multi-threaded programs. Future work will try to develop new algorithms for detecting other kinds of concurrency errors than data races and deadlocks. This includes studying completely new functionalities of the system, such as guided execution via code instrumentation to explore more of the possible interleavings of a nondeterministic concurrent program during testing.

Last, but not least, program monitoring can not only be applied during program testing, but, perhaps more interestingly, during operation, and be used to influence the program behavior in case requirements get violated. Our future research will focus on this aspect.

References

[1] M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic, Mar. 1999. Maude System documentation at http://maude.csl.sri.com/papers.

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

[3] J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of*

the 22nd International Conference on Software Engineering, Limerich, Ireland, June 2000. ACM Press.

[4] C. Demartini, R. Iosif, and R. Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.

[5] D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[6] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, Jan. 1997.

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

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

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

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

[11] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.

[12] K. Havelund and G. Roşu. Testing Linear Temporal Logic Formulae on Finite Execution Traces. RIACS Technical report, `http://ase.arc.nasa.gov/pax`, November 2000.

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

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

[15] K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In M. C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer, 1996.

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

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

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

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

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

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

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

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

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

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

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

[27] S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer, 2000.

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