Testing Linear Temporal Logic Formulae on Finite Execution Traces

Klaus Havelund
Grigore Roșu

RIACS Technical Report 01.08
May 2001

Testing Linear Temporal Logic Formulae on Finite Execution Traces[1]

Klaus Havelund, QSS / Recom Technologies
Grigore Roşu, RIACS

RIACS Technical Report 01.08
May 2001

We present a rewriting algorithm for efficiently testing 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 property and then suggest an optimized algorithm based on transforming LTL formulae. We use Maude, which turns out to be a good notation and an efficient rewriting engine for performing these experiments.

[1] This work was supported in part by the National Aeronautics and Space Administration (NASA) under Cooperative Agreement MCC 2-1006 with the Universities Space Research Association (USRA).

Testing Linear Temporal Logic Formulae on Finite Execution Traces

Klaus Havelund[1] and Grigore Roşu[2]

[1] QSS / Recom Technologies
[2] Research Institute for Advanced Computer Science
http://ase.arc.nasa.gov/{havelund,grosu}
NASA Ames Research Center
Moffett Field, CA, 94035

Abstract. We present a rewriting algorithm for efficiently testing 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 property and then suggest an optimized algorithm based on transforming LTL formulae. We use Maude, which turns out to be a good notation and an efficient rewriting engine for performing these experiments.

1 Introduction

Linear Temporal Logic (LTL), introduced by Pnueli in 1977 [21], is a logic for specifying temporal properties about reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being an example. LTL has since then 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 [13] or model checkers [11]. However, such formal proof techniques are usually not scalable to real sized systems without an extra effort to abstract the system manually to a model which is then analyzed. Model checking of programs has received an increased attention from the formal methods community within the last couple of years, and several systems have emerged that can model check source code, such as Java and C, directly [12, 25, 3, 15, 2, 20]. Stateless model checkers [6, 24] try to avoid the abstraction process by not storing states. We believe these techniques will show useful for targeted verification. Although these systems provide very high confidence in the results they provide, they scale less well. One also needs techniques that can be applied instantly and in a completely automated fashion. In this paper we investigate the use of LTL for testing whether finite execution traces conform to LTL formulae.

Our main objective is eventually to develop a practical temporal logic testing environment for NASA software developers. Testing scales well, and is by far the most used technique in practice to validate software systems. The merge of testing and temporal logic specification is an attempt to achieve the benefits of both approaches, while avoiding some of the pitfalls from ad hoc testing and the complexity in full-blown theorem proving and model checking. Of course there is a price to pay in order to obtain a scalable technique: the loss of coverage. That is, the suggested framework can only be used to examine single execution traces, and can therefore not be used to prove a system correct. The advantage is that it is completely automated and scales to programs of any size.

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 [19] and rewriting logic [18]. Some of Maude's features will be gradually introduced as needed, but it is worth mentioning at this stage that the current version of Maude can do up to 3 million rewritings per second on 800Mhz processors, and that 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 may be done in a standard programming language such as Java or C++, although the speed of Maude is very promising at this point.

The idea of using LTL in program testing is not new, and has already been pursued in the commercial Temporal Rover tool (TR) [4], which admittedly has inspired us in a major way to do this work. In TR one states LTL properties as comments in the code where statements can occur. These statements are then translated into statements in the programming language that are executed whenever reached during the execution. The implementation details of TR are not public. The work presented in this paper is motivated by the following observations. First, we find the ideas behind TR attractive due to the scalability and automatedness of the approach, and we therefore find it worthwhile to continue a practical investigation. In order to do that we need a software base for experimentation. Second, the use of a rewriting system, such as Maude, makes it possible do these experiments very fast and elegantly. This also makes it possible to formalize the ideas in a framework close to standard mathematics. Third, we believe that the formula transforming approach suggested in this paper is a new and efficient way of testing LTL formulae, the main result of the paper. Finally, whereas TR is based on annotating the code with formulae which are then expanded into the code, our approach is event-based where a program is seen as emitting events to an observer process, which then examines the events. In this respect our framework is similar to the MaC system [17] which, however, does

[1] Personal communication by José Meseguer.

not support the standard LTL. Such an event-based framework is well suited for program tracing in general, and has for example been used in detection of race conditions and deadlocks in the Visual Threads tool [9, 22], and in the Java PathFinder tool [10].

Section 2 contains preliminaries, including an introduction to Maude 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.

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

Maude [1] is a freely distributed high-performance system in the OBJ [8] algebraic specification family, supporting both rewriting logic [18] and membership equational logic [19]. Because of its efficient rewriting engine, able to execute up to 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 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.

2.2 Propositional Calculus

A decision procedure for propositional calculus due to Hsiang [16] is presented. It provides the usual truth constants (`true` and `false`) together propositional variables and 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 OBJ3 code for this procedure appeared in [8]. Below we give its obvious translation to Maude, noticing that Hsiang [16] 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 PROPOSITIONAL-CALCULUS is protecting QID .
  sort Formula .  subsort Qid < Formula .
    *** Constructors ***
  ops true false : -> Formula .
  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 .
endfm
```

`QID` is a builtin module providing a potentially infinite collection of constants called "quoted identifiers", starting with a quote character, such as `'a`, `'b`, `'id`, etc. These are used as labels for propositional variables. Operators were declared in *mix-fix* notation[2] and have attributes between squared brackets, such as `assoc`, `comm` and `prec <number>`. The attribute `prec` gives a precedence to an operator[3], thus eliminating the need for most parentheses. Once the module above is loaded[4] in Maude, reductions can be done as follows:

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

Since decidability of propositional calculus is NP-complete, one should not expect the code above to run very fast for all large problems. However, it seems to be very suitable for our purpose, so we next define LTL on top of it.

2.3 Linear Temporal Logic

Classical LTL provides in addition to the propositional logic operators the temporal operators defined by the following Maude specification:

```
fmod LINEAR-TEMPORAL-LOGIC is extending PROPOSITIONAL-CALCULUS .
  ops ([]_) (<>_) : Formula -> Formula [prec 11] .
  op _U_ : Formula Formula -> Formula [prec 14] .
  op o_  : Formula -> Formula [prec 11] .
endfm
```

[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>`.

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` (always `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. Finally, `o X` (next `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_` (next) and `_U_` (until), 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. First we present a semantics of LTL on finite traces 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

In this subsection we present a semantics of LTL on finite traces. We will regard a trace as a finite sequence of events emitted from the program that we want to observe. Such events could for example indicate when variables are written to. For example, the event *write(x,v)* would mean that "x is assigned the value v". Note that this view is slightly different from the traditional view where a trace is a sequence of program states, each state denoting the set of propositions that hold at that state. Our view is consistent with our goal to define an LTL observer as a process that is detached from the program to be observed, receiving only observed events. We shall abstract away from the concrete contents of events and just define events as a set of distinguishable identifiers. The following Maude module formalizes this idea:

```
fmod EVENT is protecting QID .
  sort Event .
  subsort Qid < Event .
endfm
```

It introduces the sort `Event` and states that the sort `Qid` of distinguishable identifiers is a subsort of `Event`. A trace is now a finite list of events. This is modeled by the following Maude specification:

```
fmod TRACE is extending EVENT .
  sort Trace .
  op end : -> Trace .
  op __ : Event Trace -> Trace [prec 25] .
endfm
```

It introduces the sort `Trace` and the constructors `end` for the empty trace, and juxtaposition of an event "e" and a trace "t", as in "$e\ t$", for creating a new trace. We shall outline the finite trace LTL semantics using standard mathematical notation rather than Maude notation. Assume two partial functions defined for nonempty traces $head$: `Trace` $\rightarrow$ `Event` and $tail$: `Trace` $\rightarrow$ `Trace` for taking the head and tail respectively of a trace, and a total function $length$ returning the length of a finite trace. That is, $head(e\ t) = e$, $tail(e\ t) = t$, and $length(\texttt{end}) = 0$ and $length(e\ t) = 1 + length(t)$. Assume further for any trace t that t_i for some natural number 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 `P` is any quoted identifier and `X` and `Y` are any formulae:

$$
\begin{array}{lll}
t \models \texttt{P} & \text{iff } t \neq \texttt{end} \text{ and } head(t) = \texttt{P}, \\
t \models \texttt{true} & \text{iff } true, \\
t \models \texttt{false} & \text{iff } false, \\
t \models \texttt{X /\textbackslash\ Y} & \text{iff } t \models \texttt{X} \text{ and } t \models \texttt{Y}, \\
t \models \texttt{X ++ Y} & \text{iff } t \models \texttt{X} \text{ xor } t \models \texttt{Y}, \\
t \models \texttt{[]X} & \text{iff } (\forall\ i \leq length(t))\ t_i \models \texttt{X} \\
t \models \texttt{<>X} & \text{iff } (\exists\ i \leq length(t))\ t_i \models \texttt{X} \\
t \models \texttt{X U Y} & \text{iff } (\exists\ i \leq length(t))\ (t_i \models \texttt{Y} \text{ and } (\forall\ j < i)\ t_j \models \texttt{X}) \\
t \models \texttt{o X} & \text{iff } t \neq \texttt{end} \text{ and } tail(t) \models \texttt{X}
\end{array}
$$

Recall the standard equations `<>X = true U X` and `[]X = !<>!X` from Subsection 2.3 on page 7. While the first equation holds also in this new finite setting, the second equation does not. To see this, observe that the formula `<>!X` always holds for a finite trace since `!X` holds on the empty trace `end`, and that therefore `[]X` would be false on all traces. We can, however, use a different set of equations: `R = o true` and `[]X = X U !R`. The operator `R` stands for "Running" and is true for non-empty traces: in the next state `true` must hold, meaning that there must be a current event and a suffix trace following it (potentially the empty one). The equation for `[]X` then states that eventually the trace ends and until then `X` holds.

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 FINITE-TRACE-SEMANTICS is protecting LINEAR-TEMPORAL-LOGIC .
  protecting TRACE .
  op _ |= _ : Trace Formula -> Bool [prec 30] .
  var P Q : Qid . var E : Event . var T : Trace . var X Y : Formula .
  eq end |= P      = false .
  eq P T |= Q      = P == Q .
  eq   T |= true   = true .
  eq   T |= false  = false .
  eq   T |= X /\ Y = T |= X and T |= Y .
  eq   T |= X ++ Y = T |= X xor T |= Y .
  eq end |= [] X   = true .
  eq E T |= [] X   = E T |= X and T |= [] X .
  eq end |= <> X   = end |= X .
  eq E T |= <> X   = E T |= X or T |= <> X .
  eq end |= X U Y  = end |= Y .
  eq E T |= X U Y  = E T |= Y or E T |= X and T |= X U Y .
  eq end |= o X    = false .
  eq E T |= o X    = T |= X .
endfm
```

Notice that the definitions that involved the partial 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 'b 'a 'c 'a 'b 'g 'f 'c 'a end |= [] ('b -> <> 'c) .
red 'a 'b 'a 'b 'a 'c 'a 'b 'g 'f 'c 'b end |= [] ('b -> <> 'c) .
```

which should return the expected answers, i.e., `true` and `false`, respectively, or by first creating a new module, say `TEST`, introducing a few traces[5] and formulae:

```
fmod TEST is extending FINITE-TRACE-SEMANTICS .
  ops trace1 trace2 trace3 : -> Trace .
  ops formula1 formula2 formula3 : -> Formula .
  eq trace1 = 'a 'b 'a 'b 'a 'c 'a 'a 'b 'g 'f 'h 'c 'b 'a end .
  eq trace2 = 'a 'b 'a 'b 'a 'c 'a 'a 'b 'g 'f 'h 'c 'b 'c end .
  eq trace3 = 'a 'b 'a 'b 'a 'c 'a 'a 'b 'g 'f 'h 'c 'b 'a
                                          . . .
              'a 'b 'a 'b 'a 'c 'a 'a 'b 'g 'f 'h 'c 'b 'a end .
  eq formula1 = [] ('b -> <> 'c) .
  eq formula2 = <> (! formula1) .
  eq formula3 = [] ((('a /\ o'b) \/ ('b /\ o'a)) U ('a /\ o'c)) .
endfm
```

and then trying various combinations of reductions:

```
red trace1 |= formula1 .    ***> should be: false
red trace1 |= formula2 .    ***> should be: true
red trace2 |= formula1 .    ***> should be: false
red trace2 |= formula2 .    ***> should be: true
red trace3 |= formula1 .    ***> should be: false
red trace3 |= formula3 .    ***> should be: false
red trace3 |= formula2 .    ***> should be: true
```

The algorithm above does nothing but blindly follow the mathematical definition of satisfaction and even runs reasonably fast for relatively small traces. For example, it takes less than 10,000 rewriting steps (a few milliseconds) to reduce the first 4 goals involving only traces of 15 events. Unfortunately, this algorithm doesn't seem to be tractable for large event traces, even if run on very performant platforms. That's because the number of atoms of the form `T |= X` keeps

[5] The three dots in `trace3` stand for 100 repetitions of the previous sequence of events. The three dots are not a Maude feature.

growing exponentially. As a concrete example, it took Maude 8 million rewriting steps to reduce the fifth expression above, 53 million steps for the sixth, and it couldn't finish the last one in 10 hours. Since the event traces generated by an executing program can easily be larger than 5,000 events, the trivial algorithm above can not be used in practice.

4 An Efficient Rewriting Algorithm

In this section we shall present a more efficient rewriting semantics. First we shall motivate the design choice. Then follows the algorithm, and finally we prove that the new semantics is equivalent to the one given in the previous section.

4.1 Motivation

The operational Maude semantics of LTL that was presented in the previous section is not efficient due to the fact that the traces are carried around in several subexpressions. For example, the semantics of the until operator is given as follows: `eq E T |= X U Y = E T |= Y or E T |= X and T |= X U Y`. We can see that the trace `T` occurs in three subexpressions. A more efficient algorithm is presented below, which is based on the idea of consuming the events in the trace, one by one, and updating a data structure, say of type D, corresponding to the effect of the event on the value of the formula. Hence, we should define a function *transform* : `Event` $\times D \to D$. Our decision to write an operational Maude 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.

We have considered two approaches: an *automata* approach and a *formula* approach. In the automata approach one could translate the formula into an automaton, and then take the synchronized product of the automaton and the execution trace. This is for example how Büchi automata are used in explicit-state model checkers for representing formulae [14, 5]. A Büchi automaton is a special automaton which accepts infinite traces (words): certain states are designated as *acceptance* states, and an infinite trace is in the language of the automaton if it brings the automaton through an acceptance state infinitely often. A model checker can detect such infinite traces by hashing states and detect cycles that include acceptance states.

We have decided not to use Büchi automata for a number of reasons. First, the translation of LTL formulae to Büchi automata is quite involved, at least of one strives for small automata, causing this approach to become inflexible the moment we want to experiment with the logic, for example by adding past time temporal operators, or real time temporal operators that refer to time units. It should be noted that other similar systems like Temporal Rover [4] and MaC [17] do not use Büchi automata, and in the Temporal Rover case exactly for the reason stated above[6]. Second, at a semantic level, Büchi automata are

[6] According to personal communication with Doron Drusinsky.

interpreted over infinite traces, and the question would be how to interpret them on finite traces. Consider for example a property such as `[](P -> <>Q)`, the automaton A generated from the formula, and a finite *error free* trace t that according to the semantics satisfies the formula. The naive suggestion would be to drive the automaton A by t until the end of the trace, and then observe whether the automaton is in an acceptance state or not. This will, however, generally not work. In experiments made using the LTL-to-Büchi automata translator in the SPIN system [14][7] such as trace may bring the automaton to a state that is not an acceptance state. Hence, one can generally not conclude anything from the resulting state. A potential solution would be to pretend that an infinite sequence of stuttering transitions is appended to the trace, where a stuttering transition does not satisfy any propositions. One could then examine whether such a stuttering sequence would bring the automaton from the state(s) resulting from the finite trace, through an acceptance state infinitely often. Hence, the stuttering should be shown to "finish off" the automaton correctly. However, even though such an interpretation is possible, a different issue is that our finite trace semantics of the *always* operator `[]_` is different from the infinite trace semantics implied by Büchi automata.

Hence, a Büchi automata approach could be possible, and will be investigated, but we are not convinced that it is worthwhile the effort. In the *formula* approach that we choose to follow instead, we let the formula to be checked be represented by itself in some normal form, and let it evolve as the execution trace is traversed, reducing it to its normal form after each event, using rewriting. This turns out to be a very efficient solution in a testing context such as the one presented here.

4.2 Consuming Events

We define the formula transforming function in the following Maude module. Given an event `E` and a formula `X`, then `X{E}` denotes a new formula. The intuition behind this formula transformer is as follows. 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`.

```
fmod CONSUME-EVENT is protecting LINEAR-TEMPORAL-LOGIC .
  protecting EVENT .
  op _{_} : Formula Event -> Formula [prec 10] .
  var E : Event .  vars X Y : Formula .  vars P Q : Qid .
  eq        P {Q} = if P == Q then true else false fi .
  eq     true {E} = true .
  eq    false {E} = false .
  eq (X /\ Y) {E} = X {E} /\ Y {E} .
  eq (X ++ Y) {E} = X {E} ++ Y {E} .
  eq   ([] X) {E} = [] X /\ X {E} .
  eq   (<> X) {E} = <> X \/ X {E} .
  eq  (X U Y) {E} = Y {E} \/ X {E} /\ X U Y .
  eq    (o X) {E} = X .
endfm
```

[7] The formula can be translated by calling SPIN as follows: `spin -f "[](p -> <>q)"`.

A propositional identifier is transformed to `true` if the event equals that proposition, otherwise `false`. 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 `[]<>P` where `P` is a propositional identifier. This formula applied to the distinct proposition `Q` yields the following rewritings: `([]<>P){Q} => []<>P /\ (<>P){Q} => []<>P /\ (<>P \/ P{Q}) => []<>P /\ (<>P \/ false) => []<>P /\ <>P`. As we can see, the property `<>P` has been spawned off as a consequence of the `Q` event, in addition to the original formula that still has to hold due to the "`[]`" operator.

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. Interestingly enough this turns out to be less efficient, a result that we had not quite expected since propositional logic rewriting seems to benefit from rewriting into normal forms as demonstrated with the module `PROPOSITIONAL-CALCULUS` described in Subsection 2.2.

4.3 Revised Semantics

Before we complete the definition of our fast algorithm to evaluate formulae on finite traces, we need to introduce a new operation, `eval`, which basically "evaluates" to either `true` or `false` a formula *as it is*, that is, without using any information about the trace. This operation is needed when all the events in the trace are consumed.

```
fmod EVAL is protecting LINEAR-TEMPORAL-LOGIC .
  op eval : Formula -> Bool .
  var P : Qid .  vars X Y : Formula .
  eq eval(P)      = false .
  eq eval(true)   = true .
  eq eval(false)  = false .
  eq eval(X /\ Y) = eval(X) and eval(Y) .
  eq eval(X ++ Y) = eval(X) xor eval(Y) .
  eq eval([] X)   = true .
  eq eval(<> X)   = eval(X) .
  eq eval(X U Y)  = eval(Y) .
  eq eval(o X)    = false .
endfm
```

The intuition here is that at the end of a trace, no proposition holds. The module also gives the semantics of the temporal operators on the empty trace. Now, the revised semantics of finite trace LTL:

```
fmod FINITE-TRACE-SEMANTICS-REVISED is protecting CONSUME-EVENT .
  protecting TRACE .
  protecting EVAL .
  op _ |- _ : Trace Formula -> Bool [prec 30] .
  var E : Event .  var T : Trace .  var X : Formula .
  eq end |- X = eval(X) .
  eq E T |- X = T |- X {E} .
endfm
```

This module defines a new relation `|-` between traces and formulae. The term `T |- X` is evaluated now by a recursive 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 `PROPOSITIONAL-CALCULUS` module in Section 2.2.

Verification results show that the optimized semantics is orders of magnitudes faster than the first semantics. A rigorous mathematical analysis of the algorithm above seems to be hard and perhaps not worth the effort at this stage, so we prefer to only report the results of our experiments which are very encouraging. If one writes a new test module, say:

```
fmod TEST-REVISED is protecting TEST .
  extending FINITE-TRACE-SEMANTICS-REVISED .
endfm
```

and then evaluates the same combinations as in Subsection 3.2,

```
red trace1 |- formula1 .    ***> should be: false
red trace1 |- formula2 .    ***> should be: true
red trace2 |- formula1 .    ***> should be: true
red trace2 |- formula2 .    ***> should be: false
red trace3 |- formula1 .    ***> should be: false
red trace3 |- formula3 .    ***> should be: false
red trace3 |- formula2 .    ***> should be: true
```

then one will immediately notice that the number of reductions and implicitly the reduction times are significantly reduced. For example, the 6th reduction, which took 53 million steps and 2 minutes under the standard semantics, needs about 4,000 rewriting steps and takes less than 10 milliseconds, while the 7th, which couldn't terminate under the standard semantics in 10 hours, needs about 155,000 rewriting steps and about 400 milliseconds on our platform.

4.4 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 [23][8].

Theorem 1. *For any trace* `T` *and any formula* `X`*,* `T |= X` *iff* `T |- X`.

Proof. The proof of this theorem is not trivial; we do it 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 `FINITE-TRACE-SEMANTICS` and `FINITE-TRACE-SEMANTICS-REVISED`:

(∀ `X : Formula`) `end |= X` = `end |- X`,

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

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

We prove them by structural induction on the formula `X`. A constant `x` is 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 FINITE-TRACE-SEMANTICS .
  extending FINITE-TRACE-SEMANTICS-REVISED .
  op e : -> Event .  op t : -> Trace .
  ops p q : -> Qid .  ops y z : -> Formula .
  eq end |= y = end |- y .
  eq end |= z = end |- z .
  eq e t |= y = t |= y {e} .
  eq e t |= z = t |= z {e} .
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. In particular, notice that an event can only be a specialized identifier since there are no other operations generating events. 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  (end |= p      == end |- p)           and (end |= true   == end |- true)
 and (end |= false  == end |- false)       and (end |= y /\ z == end |- y /\ z)
 and (end |= y ++ z == end |- y ++ z)      and (end |= [] y   == end |- [] y)
 and (end |= <> y   == end |- <> y)        and (end |= y U z  == end |- y U z)
 and (end |= o y    == end |- o y)
 and (p t |= p      ==   t |= p {p})       and (q t |= p      ==   t |= p {q})
 and (e t |= true   ==   t |= true {e})    and (e t |= false  ==   t |= false {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}) .
***> should be: true
```

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

```
fmod LEMMAS is
  protecting FINITE-TRACE-SEMANTICS .
  protecting FINITE-TRACE-SEMANTICS-REVISED .
  var E : Event .  var T : Trace .  var X : Formula .
  eq end |= X = end |- X .
  eq E T |= X = T |= X {E} .
endfm
```

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

$\mathcal{P}$(`end`), 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 end |= x == end |- x .    ***> should be: true
red e t |= x == e t |- x .    ***> should be: true
```

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 rewriting system Maude together with a much more efficient version based on formula transforming events. This exercise can be regarded as a self contained result with interest to at least the rewriting and temporal logics communities. What perhaps makes it even more interesting is that this rewriting framework likely can be applied in testing real software applications, where events are extracted from a running program and stored in a finite trace, which then subsequently is examined using a variant of the presented Maude specification. We intend to carry out this experiment on a planetary Rover software platform developed at NASA Ames.

A future research activity is to find a yet more efficient representation of an LTL formula for the purpose of achieving an optimal algorithm for testing its satisfaction on an execution trace. This becomes especially crucial for an implementation in a standard programming language such as C++ or Java. For example one can study how LTL formulae can be translated into automata similar to Büchi automata. It should be emphasized that our ultimate goal is to implement a practical tool for testing execution traces, and that a conventional programming language therefore may be the most optimal choice. Since the Maude modules are so simple and elegant, it is quite easy to experiment with different logics. We intend for example to extend the current framework to deal with past time temporal operators, as well as interval logics. Finally, we plan to extend with real time operators that refer to time stamps in the events of the execution trace. We expect such experiments to be very easy to make, but very useful in a design phase of developing a practical tool.

In related work we are developing a tool for performing *runtime analysis* on execution traces. Runtime analysis is based on the idea of extracting information from a single execution trace in order to guess properties about other execution traces. Hence this is a way to obtain a high degree of coverage although only one execution trace is examined. The technique consists of searching for *error patterns*, that is, patterns in the execution trace that may indicate potential problems. Examples of problems that can be identified this way are data races and deadlocks. For example, a deadlock potential can be discovered from a single trace that has no deadlocks if it can be observed that lock acquisitions do not follow a partial order. This means that other execution traces may then have deadlocks. The violation of the partial order between locks can be called an error pattern. We intend to integrate this environment with a temporal logic testing environment. In particular, we intend to investigate whether error patterns can be specified in a some variant of temporal logic, and hence reduce some of the effort in programming runtime analysis algorithms for detecting these error patterns.

References

1. Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic, March 1999. Maude System documentation at `http://maude.csl.sri.com/papers`.
2. James Corbett, Matthew B. Dwyer, John Hatcliff, Corina S. Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
3. Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
4. Doron Drusinsky. The Temporal Rover and the ATG Rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
5. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15th Workshop on Protocol Specification, Testing, and Verification*. North-Holland, 1995.
6. Patrice 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, January 1997.
7. Joseph Goguen, Kai Lin, Grigore Roşu, Akira Mori, and Bogdan Warinschi. An overview of the tatami project. In Kokichi Futatsugi, Tetsuo Tamai, and Ataru Nakagawa, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, to appear, 2000.
8. Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.
9. Jerry Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2000.

10. Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2000.
11. Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop*, Paris, France, November 1998. To appear in IEEE Transactions of Software Engineering.
12. Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
13. Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer, 1996.
14. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
15. Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE'99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
16. Jieh Hsiang. *Refutational Theorem Proving using Term Rewriting Systems.* PhD thesis, University of Illinois at Champaign-Urbana, 1981.
17. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
18. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, pages 73–155, 1992.
19. José 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.
20. David Y.W. Park, Urlich Stern, and David L. Dill. Java Model Checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.
21. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
22. Stefan Savage, Michael Burrows, Greg Nelson, Patrik Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
23. Natarajan Shankar, Sam Owre, and John M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.
24. Scott D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer, 2000.
25. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering.* IEEE CS Press, September 2000.