5 4 3 9 5 6

Synthesizing Monitors for Safety Properties

Klaus Havelund[1] and Grigore Roşu[2]

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

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

1 Introduction

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

Linear Temporal Logic (LTL) [18, 16] is a logic for specifying properties of reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being a typical example. LTL has been mainly used to specify properties of concurrent and interactive down-scaled models of real systems, so that fully formal correctness proofs could subsequently be carried out, for example using theorem provers or model checkers (see for example [11, 6]). However, such formal proof techniques are usually not scalable to real sized systems without a substantial effort to abstract the system more or less manually to a model which can be analyzed. Model checking of programs has

2 The PathExplorer Architecture

PathExplorer, PAX, is a flexible environment for monitoring and analyzing program executions. A program (or a set of programs) to be monitored, is supposed to be instrumented to emit execution events to an observer, which then examines the events and checks that they satisfy certain user-given constraints. The constraints can be of different kinds and defined in different languages. Each kind of constraint is represented by a rule. Such a rule in principle implements a particular logic or program analysis algorithm. Currently there are rules for checking deadlock potentials, datarace potentials, and for checking temporal logic formulae in different logics. Amongst the latter, several rules have been implemented for checking future time temporal logic, and the work presented in this paper is the basis for a rule for checking past time logic formulae. In general, the user can program new rules and in this way extend PAX in an easy way.

The system is defined in a component-based way, based on a dataflow view, where components are put together using a "pipeline" operator. The dataflow between any two components is a stream of events in simple text format, without any apriori assumptions about the format of the events; the receiving component just ignores events it cannot recognize. This simplifies composition and allows for components to be written in different languages and in particular to define observers of arbitrary systems, programmed in a variety of programming languages. This latter fact is important at NASA since several systems are written in a mixture of C, C++ and Java.

The central component of the PAX system is a so-called *dispatcher*. The dispatcher receives events from the executing program or system and then retransmits the event stream to each of the rules. Each rule is running in its own process with one input pipe, only dealing with events that are relevant to the rule. For this purpose each rule is equipped with an event parser. The dispatcher takes as input a configuration script, which specifies from where to read the program execution events, and then a list of commands - a command for each rule that starts the rule in a process.

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

3 Finite Trace Linear Temporal Logic

We briefly remind the reader the basic notions of finite trace linear past time temporal logic, and also establish some conventions and introduce some operanever true since the last time F_1 was observed to be true, including the state when F_1 was true; the interval operator, like the "since" operator, has both a strong and a weak version. For example, if START and DOWN are predicates on the state of a web server to be monitored, say for the last 24 hours, then $[\text{START}, \text{DOWN})_s$ is a property stating that the server *was* rebooted recently and since then it was not down, while $[\text{START}, \text{DOWN})_w$ says that the server was not unexpectedly down recently, meaning that it was either not down at all recently or it was rebooted and since then it was not down.

What makes past time temporal logic such a good candidate for dynamic programming is its recursive nature: the satisfaction relation for a formula can be calculated along the execution trace looking only one step backwards:

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

We call the past time temporal logic presented above *ptLTL*. There is a tendency among logicians to minimize the number of operators in a given logic. For example, it is known that two operators are sufficient in propositional calculus, and two more ("next" and "until") are needed for future time temporal logics. There are also various ways to minimize *ptLTL*. Let $ptLTL\upharpoonright_{Ops}$ be the restriction of *ptLTL* to propositional operators plus the operations in *Ops*. Then

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

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

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

[2] The first two are known in the literature [16].

problem. An important observation is, however, that, like in many other dynamic programming algorithms, one doesn't have to store all the table $s[1..n, 0..8]$, which would be quite large in practice; in this case, one needs only $s[i, 0..8]$ and $s[i-1, 0..8]$, which we'll write $now[0..8]$ and $pre[0..8]$ from now on, respectively. It is now only a relatively simple exercise to write up the following algorithm for checking the above formula on a finite trace:

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

In the following we explain the generated program.

Declarations Initially a state is declared. This will be updated as the input event list is processed. Next, the two arrays *pre* and *now* are declared. The *pre* array will contain values of all subformulae in the previous state, while *now* will contain the value of all subformulae in the current state. The trace of events is then input. Such an event list can be read from a file generated from a program execution, or alternatively the events can be input on-the-fly one by one when generated, without storing them in a file first. The latter solution is in fact the one implemented in PAX, where the observer runs in parallel with the executing program.

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

where *op* is any propositional connective. Since we have already given a detailed explanation of the example in the previous section, we shall only give a very brief description of the algorithm.

The formula should be first visited top down to assign increasing numbers to subformulae as they are visited. Let $\varphi_0, \varphi_1, ..., \varphi_m$ be the list of all subformulae. Because of the recursive nature of *ptLTL*, this step insures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i-1} \models \varphi_{j'}$ for all $j \leq j' \leq m$.

class constructor takes as parameter a reference to the object that represents the state such that any updates to the states by the monitor based on received events can be seen by the `evaluate()` method. The generated `Formulae` class for the above specification looks as follows:

```
class Formulae{
  abstract class Formula{
    protected String name;     protected State state;
    protected boolean[] pre;  protected boolean[] now;

    public Formula(String name,State state){
      this.name = name;  this.state = state;
    }
    public String getName(){return name;}
    public abstract boolean evaluate();
  }
  private List formulae = new ArrayList();
  public void evaluate(){
    Iterator it = formulae.iterator();
    while(it.hasNext()){
      Formula formula = (Formula)it.next();
      if(!formula.evaluate()){
        System.out.println("Property " + formula.getName() + " violated");
  }}}
  class Formula_P extends Formula{
    public boolean evaluate(){
      now[8] = state.holds("s");
      now[7] = state.holds("r");
      now[6] = now[7] || now[8];
      now[5] = !now[6] && pre[6];
      now[4] = state.holds("q");
      now[3] = (pre[3] || now[4]) && !now[5];
      now[2] = state.holds("p");
      now[1] = now[2] && !pre[2];
      now[0] = !now[1] || now[3];
      System.arraycopy(now,0,pre,0,9);
      return now[0];
    }
    public Formula_P(State state){
      super("P",state);
      pre = new boolean[9];  now = new boolean[9];
      pre[8] = state.holds("s");
      pre[7] = state.holds("r");
      pre[6] = pre[7] || pre[8];
      pre[5] = false;
      pre[4] = state.holds("q");
      pre[3] = pre[4] && !pre[5];
      pre[2] = state.holds("p");
      pre[1] = false;
      pre[0] = !pre[1] || pre[3];
    }
  }
  public Formulae(State state){
    formulae.add(new Formula_P(state));
  }
}
```

The class contains an inner abstract[3] class `Formula` and, in the general case, an inner class `Formula_X` extending the `Formula` class for each formula in the specification, where `X` is the formula's name. In our case there is one such `Formula_P` class. The abstract `Formula` class declares the `pre` and `now` arrays, without giving

[3] An abstract class is a class where some methods are abstract, by having no body. Implementations for these methods will be provided in extending subclasses.

A first observation is that not all the bits in *pre* are needed, but only those which are used at the next iteration, namely 2, 3, and 6. Therefore, only a bit per temporal operator is needed, thereby reducing significantly the memory required by the generated algorithm. Then the body of the generated "for" loop becomes after (blind) substitution (we don't consider the initialization code here):

```
$state \leftarrow update(state, e_i)$
$now[3] \leftarrow r(state)\ or\ s(state)$
$now[2] \leftarrow (pre[2]\ or\ q(state))\ and\ not\ (not\ now[3]\ and\ pre[3])$
$now[1] \leftarrow p(state)$
if $((not\ (now[1]\ and\ not\ pre[1])\ or\ now[2]) = 0)$
    then output(``property violated'');
```

which can be further optimized by boolean simplifications:

```
$state \leftarrow update(state, e_j)$
$now[3] \leftarrow r(state)\ or\ s(state)$
$now[2] \leftarrow (pre[2]\ or\ q(state))\ and\ (now[3]\ or\ not\ pre[3])$
$now[1] \leftarrow p(state)$
if $(now[1]\ and\ not\ pre[1]\ and\ not\ now[2])$
    then output(``property violated'');
```

The most expensive part of the code above is clearly the function calls, namely $p(state)$, $q(state)$, $r(state)$, and $s(state)$. Depending upon the runtime requirements, the execution time of these functions may vary significantly. However, since one of the major concerns of monitoring is to affect the normal execution of the monitored program as little as possible, especially in the inline monitoring approach, one would of course want to evaluate the atomic predicates on states only if really needed, or rather to evaluate only those that, probabilistically, add a minimum cost. Since we don't want to count on an optimizing compiler, we prefer to store the boolean formula as some kind of binary decision diagram, more precisely, as a term over the operation $\_?\_:\_$, for example, $pre[3]\ ?\ pre[2]\ ?\ now[3]\ :\ q(state)\ :\ pre[2]\ ?\ 1\ :\ q(state)$ (see [9] for a formal definition). Therefore, one is faced with the following optimum problem:

Given a boolean formula φ using propositions $a_1, a_2, \ldots, a_n$ of costs $c_1, c_2, \ldots, c_n$, respectively, find a $(\_?\_:\_)$-expression that optimally implements φ.

We have implemented a procedure in Maude [1], on top of a propositional calculus module, which generates all correct $(\_?\_:\_)$-expressions for φ, admittedly a potentially exponential number in the number of distinct atomic propositions in φ, and then chooses the shortest in size, ignoring the costs. Applied on the code above, it yields:

```
$state \leftarrow update(state, e_j)$
$now[3] \leftarrow r(state)\ ?\ 1\ :\ s(state)$
$now[2] \leftarrow pre[3]\ ?\ pre[2]\ ?\ now[3]\ :\ q(state)\ :\ pre[2]\ ?\ 1\ :\ q(state)$
$now[1] \leftarrow p(state)$
```

5. Klaus Havelund, Scott Johnson, and Grigore Roşu. Specification and Error Pattern Based Program Monitoring. In *European Space Agency Workshop on On-Board Autonomy*, Noordwijk, The Netherlands, 2001.
6. Klaus Havelund, Michael Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, August 2001.
7. Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
8. Klaus Havelund and Grigore Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.
9. Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. In Klaus Havelund and Grigore Roşu, editors, *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
10. Klaus Havelund and Grigore Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. San Diego, California.
11. Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer, 1996.
12. Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE'99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
13. JavaCC. Web page. `http://www.webgain.com/products/java_cc`.
14. JTrek. Web page. `http://www.compaq.com/java/download`.
15. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
16. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
17. David Y.W. Park, Ulrich Stern, and David L. Dill. Java Model Checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.
18. Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
19. Grigore Roşu and Klaus Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. Technical Report TR 01-08, NASA - RIACS, May 2001.
20. Scott D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer, 2000.
21. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.