

Efficient Monitoring of Safety Properties

Klaus Havelund¹, Grigore Roşu^{2*}

¹ Kestrel Technology NASA Ames Research Center, USA. e-mail: havelund@email.arc.nasa.gov

² Department of Computer Science, University of Illinois at Urbana-Champaign, USA. e-mail: grosu@cs.uiuc.edu

Received: date / Revised version: date

Abstract. The problem of testing whether a finite execution trace of events generated by an executing program violates a linear temporal logic (LTL) formula occurs naturally in runtime analysis of software. Two efficient algorithms for this problem are presented in this paper, both for checking safety formulae of the form “always P ”, where P is a past time LTL formula. The first algorithm is implemented by rewriting and the second synthesizes efficient code from formulae. Further optimizations of the second algorithm are suggested, reducing space and time consumption. Special operators suitable for writing succinct specifications are discussed and shown equivalent to the standard past time operators. This work is part of NASA’s PathExplorer project, the objective of which is to construct a flexible framework for efficient monitoring and analysis of program executions.

1 Introduction

The work presented in this paper is part of a project at NASA Ames Research Center, called PathExplorer [19, 18, 14, 17, 34], that aims at developing a practical testing environment for NASA software developers. The basic idea of the project is to analyze the execution trace of a running program to detect errors. The errors that we are considering at this stage are multi-threading errors such as deadlocks and data races, and non-conformance with linear temporal logic specifications, which is the main focus of this paper.

Linear Temporal Logic (LTL) [33, 27, 28] 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 [21, 20, 15]). However, 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 received an increased attention from the formal methods community within the last couple of years, and several systems have emerged that can directly model check source code, such as Java and C [16, 37, 9, 22, 8, 3, 32]. Stateless model checkers [12, 36] try to avoid the abstraction process by not storing states. Although these systems provide high confidence, they scale less well because most of their internal algorithms are exponential or worse.

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 of ad hoc testing and the complexity of 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. The suggested framework can only be used to examine single execution traces, and can therefore not be used to prove a system correct. Our work is based on the belief that software engineers are willing to trade coverage for scalability, so our goal is to provide tools that are completely automatic, implement very efficient algorithms and find *many* errors in programs. A longer term goal is to explore the use of conformance with a formal specification to achieve fault tolerance. The idea is that the failure may trigger a recovery action in the monitored program.

The idea of using LTL in program testing is not new. It has already been pursued in commercial tools such as

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

Temporal Rover (TR) [10], which stimulated our work in a major way. In TR, future and past time LTL properties are stated as formal comments within the program at chosen program points, like assertions. These formal comments are then, by a pre-processor, translated into code, which is inserted at the position of the comments, and executed whenever reached during program execution¹. The MaC tool [26] is another example of a runtime monitoring tool that has inspired this work. Here Java bytecode is automatically instrumented to generate events of interest during the execution. Of special interest is the temporal logic used in MaC, which can be classified as a past time interval logic convenient for expressing monitoring properties in a succinct way. A theoretical contribution in this paper is Theorem 1, which shows that the MaC temporal logic, together with 10 others, is equivalent to the standard past time temporal logic. The path exploration tool described in [13] uses a future time temporal logic formula to guide the execution of a program for debugging purposes. Hence, the role of a temporal logic formula is turned around from monitoring a trace to generation of a trace.

Past time LTL has been shown to have the same expressiveness as future time LTL [11]. However, past time LTL is exponentially more succinct than future time LTL [29]. For example, a property like *"every response should be preceded by a request"* can be easily stated in past time logic (reflecting directly the previous sentence), but the corresponding future time representation becomes *"it's not the case that (there is no request until (there is a response and no request))"*. Hence, past time LTL is more convenient for specifying certain properties, and is the focus of this paper.

We present two efficient monitoring algorithms for checking safety formulae of the form "always P ", where P is a past time LTL formula, one based on formula rewriting and one based on synthesizing efficient monitoring code from a formula. The rewriting-based algorithm illustrates how rewriting can be used to easily and elegantly define new logics for monitoring. This may be of interest when experimenting with logics, or if logics are domain specific and change with the application, or if one simply wants a small and elegant implementation. The synthesis-based algorithm, on the other hand, generates a very effective monitor for the particular past time logic, and focuses on efficiency. It is also better suited for generating code that can be inserted in the monitored program, in contrast to the rewriting approach, where a rewriting engine must be called by an external call.

The first algorithm is implemented by rewriting using Maude [5–7], an executable specification language whose main operational engine is based on term rewriting. Since flexibility with respect to defining/modifying monitoring logics is a very important factor at this stage in the development of PathExplorer, we have actually

developed a general framework using Maude which allows one to easily and effectively define new logics for runtime analysis and to monitor execution traces against formulae in these logics. The rewriting algorithm presented in this paper instantiates that framework to our logic of interest, past time LTL. The second algorithm presented in this paper is designed to be as efficient and specialized as possible, thus adding the minimum possible amount of runtime overhead. It essentially synthesizes a special purpose, efficient monitoring code from formulae, which is further compiled into an executable monitor. Further optimizations of the second algorithm are suggested, making each monitoring step typically run in time lower than the size of the monitored formula. Both algorithms are based on the fact that the semantics of past time LTL can be defined recursively in such a way that one only needs to look one step, or event, backwards in order to compute the new truth value of a formula and of its subformulae, thus allowing one to process and then discard the events as they are received from the instrumented program. Several special operators suitable for writing succinct monitoring safety specifications are introduced and shown semantically equivalent to the standard past time operators.

Section 2 gives a short description of the PathExplorer architecture, putting the presented work in context. Section 3 recalls past time LTL and introduces several monitoring operators together with their semantics, then discusses several past time logics and finally shows their equivalences. Section 4 first presents our rewriting-based framework for defining and executing new monitoring logics, and then shows how past time LTL fits into this framework. Section 5 finally explains our monitor-synthesis algorithm, together with optimizations and two ways to implement it. Section 6 concludes the paper.

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 whether they satisfy certain user-given constraints. We first give an overview of the *observer* that monitors the event stream. Then we discuss how a program is instrumented for monitoring of temporal logic properties. The instrumentation presented is specialized to Java, but the principles carry over to any programming language.

2.1 The Observer

The constraints to be monitored can be of different kinds and defined in different languages. Each kind of constraint is represented by a module. Such a constraint

¹ The implementation details of TR are not public.

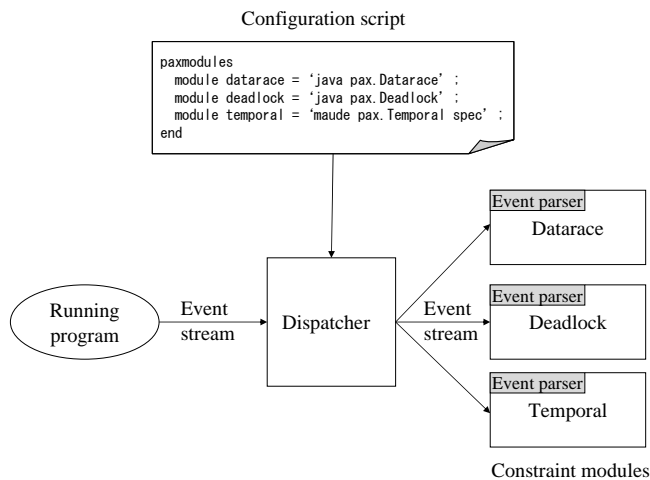


Fig. 1. Overview of the PaX observer.

module in principle implements a particular logic or program analysis algorithm. Currently there are modules for checking deadlock potentials, data race potentials, and for checking temporal logic formulae in different logics. Amongst the latter, several modules have been implemented for checking future time temporal logic, and the work presented in this paper is the basis for a module for checking past time logic formulae. In general, the user can program new constraint modules and in this manner 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, see Figure 1. The dataflow between any two components is a stream of events in simple text format, without any a-priori 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 constraint modules. Each module is running in its own process with one input pipe, only dealing with events that are relevant to the module. For this purpose each module is equipped with an event parser. The dispatcher takes as input a configuration script, which specifies a list of commands - a command for each module that starts the module in a process. The dispatcher may read its input event stream from a file, or alternatively from a socket, to which the instrumented running program must write the event stream. In the latter case, monitoring can happen on-the-fly as the

event stream is produced, and potentially on a different computer than the observed system.

2.2 Code Instrumentation

The program or system to be observed must be instrumented to emit execution events to the dispatcher (writing them to a file or to a socket as discussed above). We have currently implemented an automated instrumentation package for Java bytecode using the Java bytecode engineering tool JTrek [25]. The instrumentation package together with PathExplorer is called Java PathExplorer (JPaX). Given information about what kind of events to be emitted, the instrumentation package instruments the bytecode by inserting extra code for emitting events. For deadlock analysis, for example, events are generated that inform about lock acquisitions and releases. For temporal logic monitoring, one specifies the variables to be observed, and what predicates over these variables one wants to refer to in the temporal properties to be monitored. Imagine for example that the observer monitors the formula: “*always p*”, involving the predicate p , and that p is intended to be defined as $p \equiv x > y$, where x and y are static variables defined in a class C . In this case all updates to these variables must be instrumented, such that an update to any of them causes the predicate to be evaluated, and a toggle p to be emitted to the observer in case it has changed. The instrumentation script is written in Java (using reflection), but in essence can be represented as follows:

```

monitor C.x, C.y;
proposition p is C.x > C.y;
  
```

The code will then be instrumented to emit changes in the predicate p . More specifically, first the initial value of the predicate is transmitted to the observer. Subsequently, whenever one of the two variables is updated, the predicate is evaluated, and in case its value has changed since last evaluation, the predicate name p is transmitted to the observer as a toggle. The observer keeps track of the value of the predicate, based on its initial value, and the subsequent predicate toggles. Figure 2 shows an execution trace where x and y initially are 0, and then subsequently updated. The corresponding values of p are shown. Also shown are the events that are sent to the observer. That is, the initial value of p and the subsequent p toggles.

3 Finite Trace Past Time LTL

In this section we remind some basic notions of finite trace linear past time temporal logic [27,28], establish some conventions and introduce some operators that we found particularly useful for runtime monitoring. We emphasize that the semantics of past time LTL can be elegantly defined recursively, thus allowing us to implement

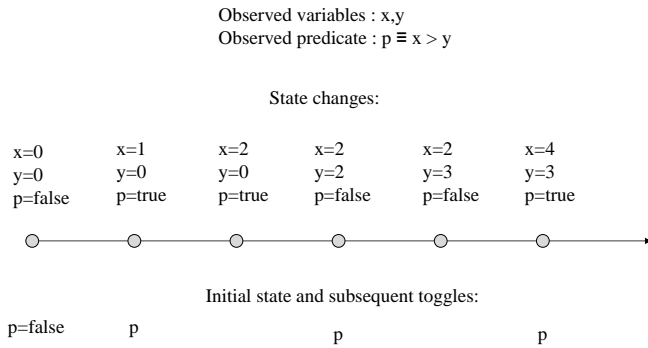


Fig. 2. Events corresponding to observing predicate $p \equiv x > y$.

monitoring algorithms that only need to look one step backwards. We also show that past time LTL can be entirely defined using just the special operators, that were introduced essentially because of practical needs, thus strengthening our belief that past time LTL is an appropriate candidate logic for expressing monitoring safety requirements.

3.1 Syntax

We allow the following constructors for formulae, where A is a finite set of “atomic propositions”:

$$\begin{aligned}
 F ::= & \text{true} \mid \text{false} \mid A \mid \neg F \mid F \text{ op } F && \text{(propositional operators)} \\
 & \circ F \mid \diamond F \mid \Box F \mid F \mathcal{S}_s F \mid F \mathcal{S}_w F && \text{(standard past time operators)} \\
 & \uparrow F \mid \downarrow F \mid [F, F]_s \mid [F, F]_w && \text{(monitoring operators)}
 \end{aligned}$$

The propositional binary operators, op , are the standard ones, that is, disjunction, conjunction, implication, equivalence, and exclusive disjunction.

The standard past time and the monitoring operators are often called “temporal operators”, because they refer to other (past) moments in time. The operator $\circ F$ should be read “previously F ”; its intuition is that F held at the immediately previous moment in time. $\diamond F$ should be read “eventually in the past F ”, with the intuition that there is some past moment in time when F was true. $\Box F$ should be read “always in the past F ”, with the obvious meaning. The operator $F_1 \mathcal{S}_s F_2$, which should be read “ F_1 strong since F_2 ”, reflects the intuition that F_2 held at some moment in the past and, since then, F_1 held all the time. $F_1 \mathcal{S}_w F_2$ is a weak version of “since”, read “ F_1 weak since F_2 ”, saying that either F_1 was true all the time or otherwise $F_1 \mathcal{S}_s F_2$.

The monitoring operators \uparrow , \downarrow , $[-, -]_s$, and $[-, -]_w$ were inspired by work in runtime verification in [26]. We found these operators often more intuitive and compact than the usual past time operators in specifying runtime requirements, despite the fact that they have the same

expressive power as the standard ones, as we discovered later. The operator $\uparrow F$ should be read “start F ”; it says that the formula F just started to be true, that is, it was false previously but it is true now. Dually, the operator $\downarrow F$ which is read “end F ”, carries the intuition that F ends to be true, that is, it was previously true but it is false now. The operators $[F_1, F_2]_s$ and $[F_1, F_2]_w$ are read “strong/weak interval F_1, F_2 ” and they carry the intuition that F_1 was true at some point in the past but F_2 has not been seen to be true since then, including that moment. For example, if START and DOWN are predicates on the state of a web server to be monitored, 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 down recently, meaning that it was either not down at all recently or it was rebooted and since then it was not down.

3.2 Formal Semantics

We next present formally the intuitive semantics described above. We regard a trace as a finite sequence of abstract states. In practice, these states are generated by events emitted by the program or system that we want to observe. Such events could indicate when variables’ values are changed or when locks are acquired or released by threads or processes, or even when a physical action takes place, such as opening or closing a valve, a gate, or a door. If s is a state and a is an atomic proposition then $a(s)$ is true if and only if a holds in the state s . Notice that we are loose with respect to what “holds” means, because, depending on the context, it can mean anything. However, in the case of JPAX the atomic predicates are just any Java boolean expressions and their satisfaction is decided by evaluating them in the current state of the Java program. If $t = s_1 s_2 \dots s_n$ ($n \geq 1$) is a trace then we let t_i denote the trace $s_1 s_2 \dots s_i$ for each $1 \leq i \leq n$. The formal semantics of the operators defined in the previous subsection is given in Figure 3.

Notice the special semantics of the operator “previously” on a trace of one state: $s \models \circ F$ iff $s \models F$. This is consistent with the view that a trace consisting of exactly one state s is considered like a *stationary* infinite trace containing only the state s . We adopted this view because of intuitions related to monitoring. One can start monitoring a process potentially at any moment, so the first state in the trace might be different from the initial state of the monitored process. We think that the “best guess” one can have w.r.t. the past of the monitored program is that it was stationary. Alternatively, one could consider that $\circ F$ is false on a trace of one state for any atomic proposition F , but we find this semantics inconvenient because some atomic propositions may be related, such as, for example, a proposition “gate-up” and a proposition “gate-down”.

