Monitoring Java Programs with Java PathExplorer

Klaus Havelund[1], Grigore Roşu[2]

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

Abstract

We present recent work on the development of Java PathExplorer (JPAX), a tool for monitoring the execution of Java programs. JPAX can be used during program testing to gain increased information about program executions, and can potentially furthermore be applied during operation to survey safety critical systems. The tool facilitates automated instrumentation of a program's byte code, which will then emit events to an observer during its execution. The observer checks the events against user provided high level requirement specifications, for example temporal logic formulae, and against lower level error detection procedures, usually concurrency related such as deadlock and data race algorithms. High level requirement specifications together with their underlying logics are defined in rewriting logic using Maude, and then can either be directly checked using Maude rewriting engine, or be first translated to efficient data structures and then checked in Java.

1 Introduction

Correctness of software is becoming an increasingly important issue in many branches of our society. People's lives often depend on software systems even though they tend to not be aware of it. The success of most technological experiments, including space craft and rover technology within the space agencies, heavily depends on the correctness of software. It is widely accepted that future space crafts will become highly autonomous, taking decisions without communication from ground, so the required software is becoming significantly more complex, increasing the risk of mission failures. Two common ways to approach the delicate problem of software correctness is *program synthesis*, which gives a high degree of confidence but seems to work properly only on very restricted domain-specific problems, and *program verification*, which is concerned with detecting as many errors as possible in existing programs.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Two important aspects of program verification are *testing* and the use of *formal methods*. Traditional testing techniques, however, are very ad hoc and do not allow for formal specification and verification of high level logical properties that a system needs to satisfy. On the other hand, traditional formal methods such as model checking and theorem proving are usually very heavy and rarely can be used in practice successfully without considerable manual effort.

The Automated Software Engineering group at NASA Ames Research Center has for some time investigated advanced formal methods for insuring software correctness, in both areas of program synthesis [14,5,19] and program verification [8,9,18,7,11]. Program synthesis is not discussed here, but it is worth noticing that code and/or data structures synthesized from logical formulae, such as finite state machines, Büchi automata or dynamic programming algorithms, are often used in program verification. We have performed various verification case studies using formal techniques, in particular model checking, to analyze space craft software [8]. Two model checkers have furthermore been developed, both supporting full state space exploration of Java programs using explicit state model checking techniques [9,18]. These techniques allow for proving temporal logic properties on programs that have a few million states, but fail to apply on large programs.

This paper is the fourth, after [10,16,11], in a series describing our effort in *runtime verification*, which can be roughly defined as combining testing and formal methods. 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 theorem proving and model checking. In this paper, we present the current status of a new runtime verification system, called Java PathExplorer (JPaX), for monitoring Java programs by analyzing (exploring) particular execution traces. The general idea consists of extracting state events from an executing program, and then analyzing them via a remote observer process. The observer performs two kinds of verification, namely *logic based monitoring* and *error pattern analysis*.

Logic based monitoring consists of checking formal requirement specifications on the executing program, written in high level logics by users of the system. Logics are currently implemented in Maude [2], a high-performance system supporting both rewriting logic and membership equational logic. One can very naturally and easily define new logics in Maude, such as for example temporal logics, together with their operational semantics. Currently, JPaX supports two builtin logics, future time and past time linear temporal logics. The implementation of both these logics in Maude together with an infrastructure module that handles atomic propositions that will most likely be part of any other more general logic, covers less than 130 lines. Therefore, defining new logics should be very feasible for advanced users. 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. Hence, we have decided to use Maude as the logical monitoring engine that performs the conformance checks of events against specifications at this early stage of JPaX.

Error pattern analysis consists of analyzing one execution trace of events using various error detection algorithms that can identify error-prone programming practices, such as unhealthy locking disciplines that may lead to data races and/or deadlocks. The important and appealing aspect of these algorithms is that they find error potentials even in the case where errors do not explicitly occur in the examined execution trace. They are usually fast and scalable, and often catch the problems they are designed to catch, that is, the randomness in the choice of run does not seem to imply a similar randomness in the analysis results. Two such known algorithms focusing on concurrency errors have been implemented in JPaX, one for deadlocks and the other for data races, but the system is designed in such a way that users can relatively easily attach new such algorithms.

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) [4], and in the MaC tool [13]. TR allows the user to specify future time temporal formulae as comments in programs, which are then translated into appropriate Java code before the compilation. The MaC tool is closer in spirit to what we describe in this paper, except that its specification language is fixed and very limited compared to the Maude language and doesn't provide support for error pattern analysis. On the other hand, tools like Visual Threads [6,17] contain hardwired error pattern analysis algorithms and therefore are impossible to change or extend by a user.

Since the programming languages of the monitored program and the observer are not required to be the same, eventually the system should allow to monitor programs composed of subprograms written in different programming languages including also C++ and C. However, for simplicity the system described in this paper will focus only on Java. A case study of 90,000 lines of C++ code for a rover controller has been carried out, leading to the detection of a deadlock with a minimal amount of effort. One of the main design goals is to make the system as general and generic as possible, allowing to handle multiple language systems and new verification rules to be defined, even defining new specification logics using Maude. Our hope is to make JPaX a basis for experiments rather than a fixed system.

The paper is organized as follows. Section 2 gives an overview of JPaX. Section 3 describes the underlying logic formalisms for writing requirement specifications, while Section 4 describes some of the error detection algorithms for debugging concurrent programs. Finally, Section 5 contains conclusions and a description of future work.

2 Overview of JPaX

JPaX can be regarded as consisting of three main modules: an *instrumentation* module, an *observer* module, and an *interconnection* module that ties them together through the observed event stream (see Figure 1). The instrumentation module performs a script-driven automated instrumentation of the program to be observed. The instrumented program, when run, will emit relevant events to the inter-connection module, which further transmits them to the observation module. The observer may run on a different computer, in which case the events are transmitted over a socket. Hence, the input to JPaX consists of references to two entities: the Java program in byte code format to be monitored (created using a standard Java compiler) and the specification script defining the kinds of verification requested. The output is a (possibly empty) set of warnings printed on a special screen.

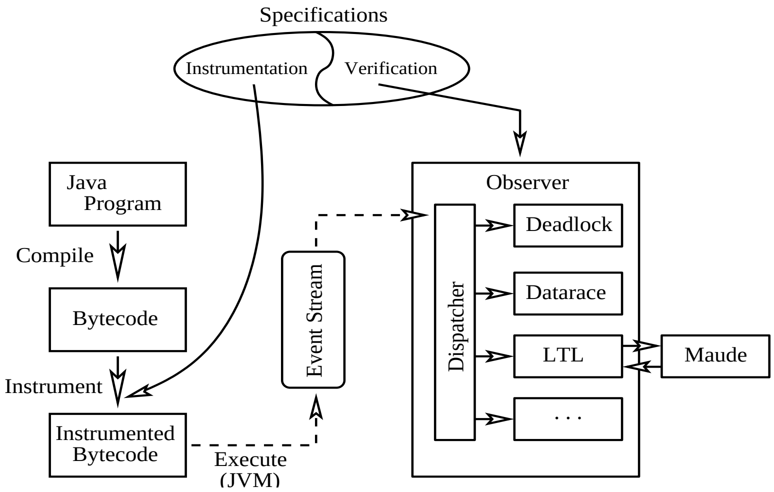

Fig. 1. Overview of JPaX

More specifically, the specification script defines what (if any) kind of error pattern detection algorithms should be activated, and what (if any) kind of logic based monitoring should be performed, and in that case what the requirements are. For logic based monitoring, we have been inspired by the MaC language framework [13] and have split the specification into an instrumentation script and a verification script. The verification script identifies the high level requirement specifications that events are to be checked against. The propositions referred to in these specifications are abstract boolean flags, and do hence not refer directly to entities in the concrete program. The instrumentation script establishes this connection between the concrete boolean program predicates and the abstract propositions. The advantage of this layered approach, as also stated in [13], is that the requirement specification can be created without considering low level issues, and can even be created before the construction of the program. Currently, the scripts are written in Java. Thus, high level Java language constructs can be used to define the boolean predicates to be observed.

The Java byte code instrumentation is performed using the powerful Jtrek Java byte code engineering tool [3] from Compaq. Jtrek makes it possible to easily read Java class files (byte code files), and traverse them as abstract syntax trees while examining their contents, and insert new code. The inserted code can access the contents of various runtime data structures, such as for example the call-time stack, and will, when eventually executed, emit events carrying this extracted information to the observer.

The observer receives the events and dispatches these to a set of observer rules, each rule performing a particular analysis that has been requested in the verification script. Generally, this modular rule based design allows a user to easily define new runtime verification procedures without interfering with legacy code. Observer rules are written in Java, but can call programs written in other languages, such as for example Maude. Maude plays a special role in that high level requirement specifications can be written in the Maude rewriting logic. The Maude rewriting engine can then be used in two different ways: as a monitoring engine during program execution, or as a translation engine before execution. In the former case, execution events are submitted to the Maude program, which in turn evaluates them against the requirement specification. In the latter case, the specification is translated into a data structure optimal for program monitoring, which is then sent back to Java, and used within the Java program to check the events during execution.

JPaX is built on a generic environment, named PathExplorer (PaX), which only consists of the interconnection module and the observer module. The goal is to make it possible to monitor programs in other programming languages, such as for example C and C++, by just providing a language specific instrumentation module. Such an experiment has been performed in collaboration with Rich Washington, a member of the Robotics group at NASA Ames, on a 90,000 line C++ application for controlling a rover. The experiment just activated the deadlock detection rule, and located a deadlock potential in the application that had not been discovered through testing.

3 Logic Based Monitoring

Logic based monitoring consists of checking execution events against a user-provided requirement specification written in some logic, typically an assertion logic with states as models, or a temporal logic with traces as models. JPaX allows the user to define such new logics in a flexible manner using the Maude algebraic specification language. Maude [2] is a modularized specification and verification system that very efficiently implements rewriting logic. A Maude module consists of operator declarations, and equations relating terms over the operators and universally quantified variables. Modules can be composed. It is relatively widely accepted that rewriting logic acts like a universal logic, in the sense that other logics, or more precisely their syntax and operational semantics, can be implemented in rewriting logic. JPAX currently provides linear temporal logics, both future time and past time, as builtin logics. Notice that multiple logics can be used in parallel, so each property can be expressed in its most suitable language. Since the Maude implementations of the current logics are quite compact, we include them below. The Maude notation will be introduced "on the fly" as we give the examples.

3.1 Propositional Calculus

We begin with the following module for propositional calculus, which is heavily used in JPAX, since most logics are based on it. It implements an efficient procedure due to Hsiang [12] to decide validity of propositions:

```
fmod PROP-CALC is ex FORMULA .
*** Constructors ***
  op _/\_ : Formula Formula -> Formula [assoc comm] .
  op _++_ : Formula Formula -> Formula [assoc comm] .

  vars X Y Z : Formula . var As* : AtomState* .

  eq true /\ X = X .
  eq false /\ X = false .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ X = X .
  eq X /\ (Y ++ Z) = (X /\ Y) ++ (X /\ Z) .
*** Derived operators ***
  op _\/_ : Formula Formula -> Formula .
  op _->_ : Formula Formula -> Formula .
  op _<->_ : Formula Formula -> Formula .
  op !_    : Formula -> Formula .
  eq X \/ Y = (X /\ Y) ++ X ++ Y .
  eq ! X = true ++ X .
  eq X -> Y = true ++ X ++ (X /\ Y) .
  eq X <-> Y = true ++ X ++ Y .
*** Semantics
  eq (X /\ Y){As*} = X{As*} /\ Y{As*} .
  eq (X ++ Y){As*} = X{As*} ++ Y{As*}
endfm
```

The module `FORMULA`, which is extended (imported), defines the infrastructure for all the user-defined logics. This will be further described in subsequent sections. For now it suffices to say that it includes some designated basic sorts (or types) such as `Formula` for syntactic formulae; `FormulaDS` for formula data structures needed when more information than the formula itself should be kept for the next transition as in the case of past time LTL; `Atom` for atoms, or state variables, which in the state denote a boolean value; `AtomState` for such assignments of boolean values to atoms, and `AtomState*` for such assignments together with final assignments, i.e., those that are followed by the end of a trace, requiring a special evaluation as described in the sections on future time and past time LTL (our semantics for the end of the execution trace is that of a continuous process that doesn't change the state). The propositions that hold in a certain program state are generated from the executing instrumented program. Perhaps the most important operator provided by the module `FORMULA` is an operation `_{_}:FormulaDS AtomState -> FormulaDS` which updates the formula data structure when an (abstract) state change occurs during the execution of the program. Notice that in the case of propositional calculus, this update operation basically evaluates propositions in the new state (the last two lines). The user is free to extend all these types and operations as in the module above.

Operators are introduced after the `op` and `ops` (when more than one operator is introduced) symbols. Operators can be given attributes in square brackets, such as associativity and commutativity. Universally quantified variables used in equations are introduced after the `var` and `vars` symbols. Finally, equations are introduced after the `eq` symbol. The specification shows the flexible mix-fix notation of Maude, using underscores to stay for arguments, which allows us to define the syntax of a logic in the most natural way.

3.2 Future Time LTL

The first monitoring logic that we present, and which is built on propositional logic, is a variant of Future Time LTL [15]. Future Time LTL is a logic with execution traces as models, which makes it convenient for program monitoring. LTL provides operators such as $\Box X$ (always X), $\diamond X$ (eventually X), $\circ X$ (next X), and $X \cup Y$ (X until Y), and their composition with standard propositional operators. Usually in formal methods literature, concerned with subjects such as model checking and theorem proving, LTL models are infinite traces. In a testing context, however, traces are finite: sooner or later, the monitored program will be stopped and so its execution trace. Hence the operational semantics has to reflect this. Future time LTL can be implemented efficiently more easily than we initially thought on top of propositional calculus:

```
fmod FT-LTL is ex PROP-CALC .
*** Syntax ***
  op []_ : Formula -> Formula .
  op <>_ : Formula -> Formula .
  op o_  : Formula -> Formula .
  op _U_ : Formula Formula -> Formula .

*** Semantics ***
  vars X Y : Formula .  var As : AtomState .
  eq ([] X){As}  = ([] X) /\ X{As} .
  eq (<> X){As}  = (<> X) \/ X{As} .
  eq (o X){As}   = X .
  eq (X U Y){As} = Y{As} \/ (X{As} /\ (X U Y)) .

  eq ([] X){As *}  = X{As *} .
  eq (<> X){As *}  = X{As *} .
  eq (o X){As *}   = X{As *} .
  eq (X U Y){As *} = Y{As *} .
endfm
```

The four LTL operators are added to those of the propositional calculus using the symbols: `[]_` (always), `<>_` (eventually), `o_` (next), and `_U_` (until). The operational semantics of these operators is based on a *formula transformation* idea, and is defined by 8 rules, divided into two groups, all refining the operator `_{_}:FormulaDS AtomState -> FormulaDS` that comes from the `FORMULA` module. Note that in the future time LTL case the formulae themselves are used as data structures (`Formula` is a subsort of `FormulaDS`). This operator defines how a formula is *transformed* by the occurrence of a state change (a new state), and evaluated on the propositional leaves. The intuition behind the `_{_}` operator can be elaborated as follows. Assume a formulae `X` we want to hold on an execution trace of which the first state is `As`. Then the equation `X{As} = X'`, where `X'` is a formula resulting from applying the `_{_}` operator to `X` (and `As`), carries the following intuition: "*in order for X to hold on the rest of the trace, given that the first state in the trace is As, then X' must hold on the trace following As*". The first set of rules describes this semantics assuming that the state `As` is not the last state in the trace, while the last four rules apply when the state `As` is the last in the trace. The term `As *` represents a state that is the last in the trace, and reflects the intuition that the *finite* trace can be regarded as an *infinite* trace where the last state of the finite trace is repeated infinitely. The two rules for each operator implement the following simple equivalences:

$$s \frown t \models \varphi \text{ iff } t \models \varphi\{s\}$$

$$s \frown end \models \varphi \text{ iff } \varphi\{s*\} = true,$$

where $s \frown t$ is a trace formed by a state s followed by a nonempty trace t, and $s \frown end$ is the trace consisting of s followed by the end of trace (the last state in the finite trace). As an example, consider the formula `[](X -> <>Y)` and a trace where the first state `As` makes `X` true but `Y` false. In this case `[](X -> <>Y)`$\{$`As`$\}$ = `[](X -> <>Y)` $\wedge$ `<>Y` (modulo propositional calculus rewriting). This reflects the fact that after the state change, `<>Y` now has to be true on the remaining trace, in addition to the original *always*–formula. A proof of correctness of this algorithm is given in [10]. Despite its overall exponential complexity, this algorithm tends to be quite acceptable in practical situations. We couldn't notice any significant difference in global concrete experiments with JPAX between this simple 8 rule algorithm and an automata-based one developed by Dimitra Giannakopoulou, that implements in 1,400 lines of Java code a Büchi automata inspired algorithm adapted to finite trace LTL (see Subsection 3.4).

Such a finite trace semantics for LTL used for program monitoring has, however, some characteristics that may seem unnatural. At the end of the execution trace, when the observed program terminates, the observer needs to take a decision regarding the validity of the checked properties. Let us consider again the formula `[](p -> <>q)`. If each `p` was followed by at least one `q` during the monitored execution, then, at some extent one could say that the formula was satisfied; although one should be aware that this is not a definite answer because the formula could have been very well violated in the future if the program hadn't been stopped. If `p` was true and it was not followed by a `q`, then one could say that the formula was violated, but it may have been very well satisfied if the program had been left to continue its execution. Furthermore, every `p` could have been followed by a `q` during the execution, only to be violated for the last `p`, in which case we would likely expect the program to be correct if we terminated it by force. There are of course LTL properties that give the user absolute confidence during the monitoring. For example, a violation of a safety property reflects a clear misbehavior of the monitored program.

The lesson that we learned from experiments with LTL monitoring is twofold. First, we learned that, unlike in model checking or theorem proving, LTL formulae and especially their violation or satisfaction must be viewed with extra information, such as for example statistics of how well a formula has "performed" along the execution trace. Second, we developed a belief that

LTL may not be the most appropriate formalism for logic based monitoring; other more specific logics, such as real time LTL, interval logics, past time LTL, or even undiscovered ones, could be of greater interest than pure LTL. In the next subsection we describe an implementation of past time LTL in Maude, a perhaps more natural logic for runtime monitoring.

3.3 Past Time LTL

Past time LTL is useful for especially safety properties. These properties are very suitable for logic based monitoring because they only refer to the past, and hence their value is always either true or false in any state along the trace, and never *to-be-determined* as in future time LTL. The implementation of past time LTL is, however, surprisingly slightly more tedious than the above implementation of future time LTL. It is also built on top of propositional calculus, by adding the usual two past time operators, `~_` for *previous* and `_S_` for *since*, and then appropriate data structures and semantics. The implementation appears similar to the one used in [13] (according to private communication), which also uses a version of past time logic. We here present the past time logic module as is, and then give a step-wise explanation.

```
fmod PT-LTL is ex PROP-CALC .
*** Syntax ***
  op ~_  : Formula -> Formula .
  op _S_ : Formula Formula -> Formula .

*** Semantic Data structure ***
  op ptLtl : Formula -> FormulaDS .
  op atom  : Atom Bool -> FormulaDS .
  op prev  : FormulaDS Bool -> FormulaDS .
  op and   : FormulaDS FormulaDS Bool -> FormulaDS .
  op xor   : FormulaDS FormulaDS Bool -> FormulaDS .
  op since : FormulaDS FormulaDS Bool -> FormulaDS .

  var A : Atom .  var As : AtomState .
  var B : Bool .
  vars X Y : Formula .
  vars D D' Dx Dx' Dy Dy' : FormulaDS .

  eq [atom(A,B)] = B .
  eq [prev(D,B)] = B .
  eq [since(Dx,Dy,B)] = B .
  eq [and(Dx,Dy,B)] = B .
  eq [xor(Dx,Dy,B)] = B .

  eq ptLtl(true){As} = true .
  eq ptLtl(false){As} = false .
  eq ptLtl(A){As} =
    atom(A, (A{As} == true)) .
  eq ptLtl(~ X){As} = false .
  ceq ptLtl(X S Y){As} = since(Dx,Dy,[Dy])
     if Dx := ptLtl(X){As} /\
        Dy := ptLtl(Y){As} .
  ceq ptLtl(X /\ Y){As} =
     and(Dx,Dy,[Dx] and [Dy])
       if Dx := ptLtl(X){As} /\
          Dy := ptLtl(Y){As} .
  ceq ptLtl(X ++ Y){As} =
     xor(Dx,Dy,[Dx] xor [Dy])
       if Dx := ptLtl(X){As} /\
          Dy := ptLtl(Y){As} .

*** Semantics ***
  eq atom(A,B){As} =
     atom(A, (A{As} == true)) .
  eq prev(D,B){As} = prev(D{As},[D]) .
  ceq since(Dx,Dy,B){As} =
     since(Dx',Dy',[Dy'] or B and [Dx])
       if Dx' := Dx{As} /\
          Dy' := Dy{As} .
  ceq and(Dx,Dy,B){As} =
     and(Dx',Dy',[Dx'] and [Dy'])
       if Dx' := Dx{As} /\
          Dy' := Dy{As} .
  ceq xor(Dx,Dy,B){As} =
     xor(Dx',Dy',[Dx'] xor [Dy'])
       if Dx' := Dx{As} /\
          Dy' := Dy{As} .
endfm
```

The module first introduces the syntax of the logic, the *previous*–operator and the *since*–operator. The next two sections of the module introduce the semantic data structure needed for past time LTL formulae, and its semantics. The data structure is represented by the sort `FormulaDS`, introduced in the `FORMULA` module, and is needed to represent a formula during execution. This is in contrast to future time LTL, where a formula represented itself, and a transformation caused by a state transition was performed by transforming the formula into a new formula that had to hold on the rest of the trace. In past time LTL this technique does not apply. Instead, for each formula a special tree-like data structure is introduced, which keeps track of the boolean value of all subformulae of the formula in the previous state. These values are used to correctly evaluate the value of the entire formula in the next state. The operation `ptLtl` initializes/creates the data structure representing a formula. The constructors of the type `FormulaDS` correspond to the different kinds of past time LTL operators: `atom` (for atomic propositions), `and`, `xor`, `prev`, and `since`. Hence, for example, the formula `~ A` (*previous* `A`) for some atomic proposition `A` is represented by `prev(atom(A,true),false)` in the example case that `A` is true in the current state, but was false in the previous state. Hence the second boolean argument represents the current value of the formula, and is returned by the `[_]` operation. The `ptLtl` operation that creates the initial data structures from formulae is defined through equations that also define the operation `_{_}:FormulaDS AtomState -> FormulaDS` on the initial atomic state. Hence, this defines how the data structure of a formula is initialized. Note that this operation now is applied to the data structure of a formula. The equations for the three binary operators (`since`, `and` and `xor`) are defined using conditional equations (`ceq`). Conditions are provided after the `if` keyword and introduce new variables used in the equations.

3.4 Efficient Observer Generation

Logic-based monitoring can add overhead to the normal execution of programs. Because of the high complexity of validity in many logics, it is very easy to design and implement inefficient algorithms. We deliberately decided that, at this early stage of JPaX, it is more important to concentrate our efforts on finding and experimenting with more expressive and natural logics for monitoring, rather than implementing very efficient algorithms for particular logics which may soon turn out not to be the most appropriate ones. However, since LTL seems to be a good candidate logic, we started to investigate efficient runtime formula verification algorithms for both future time and past time LTL. More precisely, we are looking for algorithms that generate efficient *observers* from formulae, i.e., (Java) code or data structures that "encode" the formulae and can be executed or modified synchronously with the observed program, returning an appropriate message when the formula is violated.

After experimenting with runtime verification algorithms for LTL [10,16,11], each with its advantages and drawbacks, we realized that in order for one to properly compare these, one needs to first understand and establish criteria for "good" runtime verification algorithms. Consider a fixed logic. The following is a list of priorities that currently influence the choice of runtime algorithms in JPaX :

Forwards Design. Algorithms that visit the execution traces backwards involve storing the trace and cannot throw exceptions or guide the program when a formula is violated.

Runtime Efficiency. An algorithm that is exponential in the size of the trace is unusable, while an algorithm that is exponential in the size of formula is usable but better be avoided.

Initialization. The time required to generate code or data structures from formulae cannot be ignored, but it is considered less important than the previous criteria.

A trivial rewriting algorithm for future time LTL that blindly implements the semantics is immediate (see also [10]), but it is exponential in the size of the trace, so it is impractical. The simple and elegant procedure shown in Subsection 3.2 and proved correct in [10] is worst-case exponential only in the size of the formula but linear in the size of the trace. We found it quite good in practice so far and the fact that it can be implemented in only a few lines of Maude code makes it a very good choice at this incipient stage of JPAX. Dynamic programming algorithms generated from future time LTL formulae [16] run in time $O(nm)$, where n is the size of the trace and m is the size of the formula. Unfortunately, these algorithms visit the execution trace backwards so they fail to satisfy the first criterion. Fortunately, the same idea applies to past time LTL and, by dualization, yields forwards algorithms of the same complexity. Therefore, past time LTL is a very nicely computable logic for monitoring. Besides that, the naturalness with which one expresses safety requirements in it makes us believe that it is a better choice than future time LTL.

However, we next very briefly present some concepts that lead to a future time finite-trace LTL formula-checking algorithm that is the best one of which we are aware satisfying the criteria above. It visits the execution trace forwards and its worst-case runtime complexity is $O(nk)$, where n is the length of the trace and k is the number of variables of the formula. The complete details together with optimality proofs will appear elsewhere soon.

We first introduce some data structures that will be needed to encode a formula. Intuitively, a *binary transition tree* is a binary tree where the nodes are atomic propositions, while the leaves are states or truth values. For simplicity in writing, we make use of a C/Java-like operator $\_?\_ : \_$ having the typical intuition: $a?t_1 : t_2$ means "if a then t_1 else t_2". For example, if $P = \{a, b, c\}$ is a set of "atomic propositions" and $S = \{1, 2, 3\}$ is a set of "states", then $a?(b?1 : 2) : 1$ and $a?(c?2 : \mathsf{false}) : (c?\mathsf{true} : (b?3 : 1))$ are all well-formed $\langle P?S\rangle$-binary transition trees. We next give a compact formal definition which can be skipped by the impatient reader. Let Bool be the set $\{\mathsf{true}, \mathsf{false}\}$ and let us consider two sorts Prop and State that stay for propositions and states, respectively.

Definition 3.1 Given sets $P : \mathsf{Prop}$ and $S : \mathsf{State}$, respectively, then a $\langle P?S\rangle$-

binary transition tree (or simply $\langle P?S\rangle$**-BTT** or even **BTT**) is a term of sort BTT of the order-sorted free algebra $T_\Sigma(P, S \cup \mathsf{Bool})$ over a signature Σ consisting of the sorts Prop, State and BTT with State a subsort of BTT, and the operation[1] $(\_?\_ : \_) : \mathsf{Prop} \times \mathsf{BTT} \times \mathsf{BTT} \to \mathsf{BTT}$. If S is empty then $\langle P?\emptyset\rangle$-BTT's are called P**-binary decision trees** (or simply P-**BDT**'s or **BDT**'s; see [1]).

If size of a BTT becomes an important issue, than one can change this definition to take advantage of repetitions of subtrees, thus obtaining directed acyclic graphs instead of trees, like in the case of binary decision diagrams (see for example [1]). However, the size of BTT's doesn't seem to be important yet, in the sense that it doesn't affect any of the three criteria above.

Definition 3.2 A **BTT finite state machine** (or simply **BTT FSM**) consists of sets P and S, together with a total function next that maps each element in S into a $\langle P?S\rangle$-BTT. A **BTT finite trace FSM** is a **BTT FSM** together with a total function end that maps each element in S into a P-BDT.

The function end decides whether a state is accepting or not when a trace ends there. The notion of accepted "execution" trace should be next defined but space doesn't allow us to go into more formal aspects. We only show how the LTL formula $\Box(a \to \diamond b)$ can be encoded as a BTT finite trace FSM: in this case $P = \{a, b\}$, $S = \{1, 2\}$, $\mathsf{next}(1) = a?(b?1 : 2) : 1$, $\mathsf{end}(1) = a?(b?\mathsf{true} : \mathsf{false}) : \mathsf{true}$, and $\mathsf{next}(2) = b?1 : 2$, $\mathsf{end}(2) = b?\mathsf{true} : \mathsf{false}$. The intuition for this data structure is as follows. If the monitored program, say $\mathcal{P}$, is in a state which is not the end of the observed trace, then: if the observer is in state 1 then evaluate the atomic proposition a in the current state of $\mathcal{P}$ and if this is true then evaluate b and if this is false then change the state of the observer to 2; if the observer is in state 2 then evaluate only b and if this is true then change the observer state to 1. If one decides to stop the monitoring of $\mathcal{P}$, then the end BDT's are evaluated similarly. Notice that false is returned when an a occurred in the execution trace which was not followed by a b. The reader may have already noticed that we payed special attention to the evaluation of atomic propositions: they are evaluated only when needed. This is because the evaluation process can be often long; for example, an atomic proposition can test whether an array is sorted.

We have designed and implemented in Maude (in less than 200 lines of code) a relatively easy and elegant procedure that generates an optimal BTT finite trace FSM from any LTL formula. Despite its worst-case exponential complexity, it is quite fast on typical formulae and it never needed more than 30 seconds (on a 400MHz laptop) to generate an optimal data structure; it needed more than 1 second only on hand-crafted artificial formulae. This initialization time is spent only once, at the beginning of the monitoring. The following are a few examples of optimal BTT finite trace finite state machines

[1] Written in mix-fix notation.

generated by our current implementation:

Formula	State	next	end
$\Box \diamond a$	1	1	$a?\mathsf{true} : \mathsf{false}$
$\diamond(\Box a \vee \Box\neg a)$	1	1	true
$\Box(a \to \diamond b)$	1	$a?(b?1 : 2) : 1$	$a?(b?\mathsf{true} : \mathsf{false}) : \mathsf{true}$
	2	$b?1 : 2$	$b?\mathsf{true} : \mathsf{false}$
$a\ \mathcal{U}\ (b\ \mathcal{U}\ c)$	1	$c?\mathsf{true} : (a?1 : (b?2 : \mathsf{false}))$	$c?\mathsf{true} : \mathsf{false}$
	2	$c?\mathsf{true} : (b?2 : \mathsf{false})$	$c?\mathsf{true} : \mathsf{false}$

Notice that liveness properties do not really make sense in finite trace LTL without statistical analysis. In particular, the formula $\Box \diamond a$ is violated if and only if a is false in the last observed state of the monitored program. The formula $\diamond(\Box a \vee \Box\neg a)$ is always true in finite trace LTL and our optimal generator proved that.

4 Error Pattern Analysis

Logic based analysis of execution traces can reveal domain specific high level errors, but it implies human intervention in designing the application requirements or/and their underlying logics. However, many errors are lower level and are usually due to bad programming practice or lack of attention, and fortunately, an interesting portion of them can be revealed automatically. Even if some of these error patterns could be specified using adequate requirements formalisms and then enforced using the same logic-based approach as above, we think that this procedure is too heavy for this kind of errors, and that it is actually more appropriate to allow the users attach designated efficient algorithms to JPAX. We have implemented the algorithms described below in both Maude and Java, but the current JPAX uses the Java implementations.

Error pattern runtime analysis algorithms explore an execution trace and detect error potentials. The important and appealing aspect of these algorithms is that they find error potentials even in the case where errors do not explicitly occur in the examined execution trace. They are usually fast and scalable, and often catch the problems they are designed to catch, that is, the randomness in the choice of run does not seem to imply a similar randomness in the analysis results. The trade off is that they have less coverage than heavyweight formal methods and often suggest problems which, after a careful semantical analysis, turn out not to be errors. Two examples of such algorithms focusing on concurrency errors have been implemented in JPAX: the Eraser [17] data race analysis algorithm and a deadlock analysis algorithm based on analyzing lock cycles. Both these algorithms have been previously implemented by Compaq in the Visual Threads tool [6] to work for C and C++. Inspired by the Visual Threads tool, we also previously implemented the data race algorithm and a variant of the deadlock algorithm in Java PathFinder [7], modifying the Java Virtual Machine described in [18]. Our contribution in error pattern analysis for JPaX is to make these algorithms work for Java using byte code instrumentation, to integrate them with logic based monitoring, and to allow advanced users to program new error pattern analysis rules in a flexible manner. The rest of this section shortly describes the data race and deadlock detection algorithms.

4.1 Data Race Analysis

We briefly describe here how easily data races can occur in concurrent programming and how Eraser [17] has been implemented in JPaX to work on Java programs. A *data race* occurs when two or more concurrent threads access a shared variable, at least one access is a *write*, and the threads use no explicit mechanism to prevent the accesses from being simultaneous. The Eraser algorithm detects data races by studying a single execution trace of the monitored program, trying to conclude whether there exist valid runs where data races are possible. We illustrate the data race analysis with the following example.

```
 1. class Value{
 2.   private int x = 1;
 3.
 4.   public synchronized void add(Value v){x = x + v.get();}
 5.
 6.   public int get(){return x;}
 7. }
 8.
 9. class Task extends Thread{
10.   Value v1; Value v2;
11.
12.   public Task(Value v1,Value v2){
13.     this.v1 = v1; this.v2 = v2;
14.     this.start();
15.   }
16.
17.   public void run(){v1.add(v2);}
18. }
19.
20. class Main{
21.   public static void main(String[] args){
22.     Value v1 = new Value(); Value v2 = new Value();
23.     new Task(v1,v2); new Task(v2,v1);
24.   }
25. }
```

The class `Value` contains an integer variable `x`, a synchronized method `add` that updates `x` by adding the content of another `Value` variable, and an unsynchronized method `get` that simply returns the value of `x`. `Task` is a thread class: its instances are started with the method `start` which executes the user defined method `run`. Two such tasks are started in `Main`, on two instances of the `Value` class, `v1` and `v2`. When running JPaX with the Eraser option switched on, a data race potential is found, reporting that the variable `x` in class `Value` is accessed unprotected by the two threads in lines 4 and 6, respectively. The generated warning message gives a scenario under which a data race might appear, summarizing the following. One `Task` thread can call the `add` method on the object `v1` with the parameter `Value` object `v2`, whose content is thus read via the unsynchronized `get` method. The other thread can simultaneously do the same thing, i.e., call the `add` method on `v2`. Therefore, the content of `v2` might be accessed simultaneously by the two threads. Two data race warnings are actually emitted, since the the other task can perform the same behavior with `v1` and `v2` interchanged.

Roughly, the algorithm works and is implemented in JPAX as follows. The instrumented byte code of the monitored program emits to the observer appropriate events when variables are read or updated, and when locks are acquired or released as a result of executing Java's `synchronized` statements or from calling/returning from synchronized methods. The observer maintains two data structures: a *thread map* that keeps track of all the locks owned by each thread, and a *variable map* that associates with each (shared) variable the intersection of the set of locks that has been commonly owned by all accessing threads in the past. If this set ever becomes empty then a data race potential exists. More precisely, when a variable is accessed for the first time, the locks owned by the accessing thread at that moment are stored in the variable's variable set. Subsequent accesses by other threads causes the set to be refined to its intersection with the locks owned by those threads. An extra state machine is also maintained for each variable to keep track of how many threads have accessed the variable and how (read/write). This is used to reduce the number of false warnings, such as situations in which variables are initialized by a single thread without locks (which is safe) or several threads only read a variable after it has been initialized (which is also safe).

Deadlock Detection

Deadlock potentials are hard to find in general, but there are classical deadlock situations which occur when multiple threads take locks in different order. For example, a deadlock will arise if a thread acquires a lock and then, without releasing it, acquires another lock, while another thread first acquires the second lock and then the first one. One can simply create such a situation in the previous Java example if one wrongly tries to repair the data race by also defining the `get` method in line 6 as synchronized:

```
6. public synchronized int get(){return x;}
```

It is clear now that the data race algorithm will indeed not return a warning anymore because the variable `x` can no longer be accessed simultaneously from two threads. However, there is a deadlock potential now and JPAX detects it. More exactly, when running JPAX on the modified program, a lock order problem is found and an appropriate warning message is issued summarizing the fact that two object instances of the `Value` class are taken in a different order by the two `Task` threads. It also indicates the line numbers where the threads may potentially deadlock: line 4 where the `get` method called from add may lock the second object. Notice that this deadlock doesn't need to appear in the examined trace in order for this warning to be issued. In fact, deadlock potentials might be reported in general even if those deadlocks will never appear in any execution of the program. Any execution of the modified program above will cause a warning to be issued.

The runtime deadlock analysis algorithm is also implemented in the observer and it needs only a subset of the events generated for the data race algorithm, namely those related to lock acquires and releases of locks that result from executing Java's `synchronized` statements or from calling/returning from synchronized methods. Two data structures are maintained in the observer: as in the data race algorithm a *thread map* keeps track of the locks owned by each thread, while a second data structure, a *lock graph*, updates a graph that accumulates as nodes all the locks taken by any thread during an execution, the edges recording locking orders. In other words, an edge is introduced from a lock to another each time when a thread that already owns the first lock acquires the other. If during the execution of the program this graph becomes cyclic, then there is a deadlock potential related to lock ordering in the program. This simple algorithm can reveal more complex deadlock potentials between more than two threads, as illustrated for example by the classical dining philosopher's example.

5 Conclusions

We have presented JPAX, a runtime verification tool under development at NASA Ames Research Center. JPAX provides an integrated environment for instrumenting Java byte code to emit events during execution to an observer, which performs two kinds of analysis: logic based monitoring, checking events against high level requirements specifications, and error pattern analysis, searching for low level programming errors. It has been shown how the two kinds of verification can be combined by viewing them as rules within an extensible set of rules. It has in particular been demonstrated how the Maude rewriting logic can be used to define new logics for runtime verification in a very flexible manner, and how the Maude inference engine can be used to perform the monitoring itself. In the case where efficiency is required, we have shown that optimal data structures can be generated from future time and past time LTL. Finally, two known error pattern detection algorithms, one for data races and one for deadlocks, have been implemented to work on Java.

The project as described above mainly focuses on applying the tool during testing of a software application. Hence, with this perspective the goal is to smoothly combine testing and formal methods, while avoiding some of the pitfalls from ad hoc testing and the complexity of full-blown theorem proving and model checking. However, an at least equally interesting application of runtime verification is to apply it during operation, and influence the program behavior when requirements are violated. Our future research will focus on this aspect. In general, integration in the overall NASA Ames automated software engineering effort is highlighted, and here two crucial issues are: how can testing be made more formal, and how can missions be made safer in the face or errors occurring during flight that survived tests.

Of other future work can be mentioned that we will experiment with new logics in Maude more appropriate to monitoring than LTL, such as interval and real time logics and UML notations. The latter allows to check original designs (via state charts and/or sequence diagrams) against "real" execution traces. Future work on error pattern analysis will try to develop new algorithms for detecting concurrency errors other than data races and deadlocks, and of course to try to improve existing algorithms. We will also study completely new functionalities of the system, such as guided execution via code instrumentation to explore more of the possible interleavings of a non-deterministic concurrent program during testing. Dynamic program visualization is also a future subject, where we regard a visualization package as just another rule in the observer. A more user friendly interface, both graphical and functional, will be provided, and finally the tool will be evaluated against NASA safety critical applications.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[2] M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *LNCS*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag. System description.

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

[4] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

[5] B. Fischer, T. Pressburger, G. Rosu, and J. Schumann. The AutoBayes Program Synthesis System - System Description. In *Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (CALCULEMUS 2001), Siena, Italy*, June 2001.

[6] J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.

[7] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.

[8] K. Havelund, M. Lowry, and J. 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.

[9] K. Havelund and T. 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] M. Lowry, A. Philpot, T. Pressburger, I. Underwood, R. Waldinger, and M. Stickel. Amphion: Automatic Programming for the NAIF Toolkit. In *NASA Science Information Systems Newsletter*, volume 31, February 1994.

[15] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[16] 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.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[18] 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, September 2000.

[19] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, June 2000.