Java PathExplorer - A Runtime Verification Tool

Klaus Havelund
Kestrel Technology
NASA Ames Research Center
Moffett Field, CA, 94035
havelund@ptolemy.arc.nasa.gov

Grigore Roşu
Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA, 94035
grosu@ptolemy.arc.nasa.gov

Keywords Software testing, runtime verification, event tracking, temporal logic based monitoring, error pattern analysis, concurrent programs, deadlocks, data races, Java, byte-code instrumentation.

Abstract

We describe recent work on designing an environment, called Java PathExplorer, for monitoring the execution of Java programs. This environment facilitates the testing of execution traces against high level specifications, including temporal logic formulae. In addition, it contains algorithms for detecting classical error patterns in concurrent programs, such as deadlocks and data races. An initial prototype of the tool has been applied to the executive module of the planetary Rover K9, developed at NASA Ames. In this paper we describe the background and motivation for the development of this tool, including comments on how it relates to formal methods tools as well as to traditional testing, and we then present the tool itself.

1 Introduction

Software is getting an increased importance in the development of space craft and rover technology within the space agencies. It is recognized that future space crafts will become highly autonomous, taking decisions without communication from ground. Hence, the required software is becoming more complex, increasing the risk of mission failures. Testing of such systems therefore becomes crucial. Traditional testing techniques, however, are very ad hoc and do not allow for formal specification and verification or testing of the properties that a system needs to satisfy.

The Automated Software Engineering group at NASA Ames Research Center has for the last three years worked on developing advanced verification and testing technology for space applications. Part of this work has consisted of performing case studies using formal methods, in particular model checking, to analyze space craft software [6]. Based on the experiences of these case studies, two tools have furthermore been developed, both supporting full state space exploration of Java programs using explicit state model checking techniques [7, 14]. These techniques allow for checking temporal logic properties on programs that have a few million states, but fail to apply on large programs. Abstraction is required in order to increase the applicability of such techniques, an often manual and labor-some process.

We present a new runtime verification system, Java PathExplorer (JPaX), for monitoring of Java program execution traces. The general concept consists of extracting events from an executing program, and then analyzing the events via a remote observer process. The observer performs two kinds of verification: *logic based monitoring* and *error pattern analysis*.

Logic based monitoring consists of checking execution traces against user-provided formal requirement specifications, written in high level logics. Logics are currently implemented in the specification language Maude [1], a high-performance system supporting both membership equational logic and rewriting logic. Maude allows to define new logics in a flexible manner, such as for example temporal logics, together with their operational semantics. Currently we support future time and past time linear temporal logic as predefined logics. The implementation of both these logics cover less than 130 lines, hence defining new logics, for example domain specific ones, should be very feasible for an advanced user. 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 Maude can be used as the monitoring engine that performs the conformance checks of events against specifications.

Error pattern analysis consists of analyzing execution events using various error detection algorithms that can identify error-prone programming practices. Examples are unhealthy locking disciplines that may lead to data races and deadlocks. For example, a deadlock potential can be discovered from a single trace, even if that particular trace has no deadlocks, if it can be observed that lock acquisitions do not follow a partial order. By not requiring the errors to actually occur in order to be detected, this is a way to obtain a high degree of coverage although only one execution trace is examined. In general, we try to identify various concurrency error patterns.

The idea of using temporal logic in program testing is not new, and has already been pursued in the commercial Temporal Rover tool (TR) [3], and in the MaC tool [10]. TR allows the user to specify temporal formulae as comments in programs. The MaC tool is closer to what we describe in this paper, except that its specification language is very limited compared to the Maude language. In addition, we combine specification checking with error pattern analysis. In a tool like Visual Threads [4, 13], these runtime analysis algorithms have been hardwired into the system and are therefore difficult to change or extend by a user.

Eventually the system should allow to monitor programs composed of subprograms written in different programming languages including also C++ and C. The system described in this paper will focus 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. It is our main goal to make the system as general and generic as possible, allowing to handle multiple language systems, and allowing new verification rules to be defined, even defining new specification logics using Maude. This way we hope to make the system a basis for experiments rather than a fixed system.

The paper is organized as follows. Section 2 describes the overall architecture of the system. 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 System Architecture

The architecture of JPaX is shown in Figure 1. The input to JPaX consists of two entities (or rather pointers to these): the Java program in byte-code format to be monitored (created using a standard Java compiler) and the specification script defining what kind of analysis is requested. The output is a (possibly empty) set of warnings printed on a special screen.

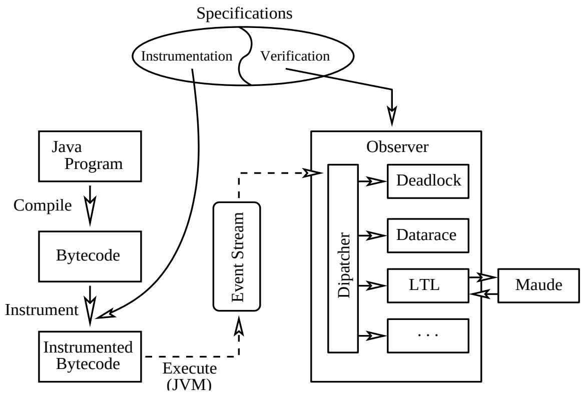

Figure 1: JPaX Architecture

The specification script consists of an instrumentation script and a verification script. The instrumentation script defines how the program should be instrumented while the verification script defines exactly what kind of analysis should be performed, and if logic based monitoring is requested: what properties should be verified. Currently, the scripts are written in Java, which calls Maude if needed. Thus, high level Java language constructs can be used to define the boolean predicates to be observed. Then the values of those predicates are shipped to Maude for deeper logic-based analysis.

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. 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 interaction 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. User defined options allow to define the general setup.

The instrumentation is performed using the Jtrek Java byte-code engineering tool [2] from Compaq. This tool allows to read Java class files (byte-code files), traverse them as abstract syntax trees while examining their contents, and insert new code in a highly flexible manner. The inserted code can access the contents of the method call-time stack at runtime, hence giving access to information needed in the analysis. The extracted information is transmitted in the events. The observer receives the events and dispatches these to a set of observer rules, each rule performing a particular analysis that has been requested. Observer rules can be written in Maude or in a traditional programming language such as Java, or even C if speed is crucial. Generally, the rule based design allows a user to easily define new runtime verification procedures.

The only language specific part of the system is the instrumentation module. If one wants to set up the environment for a different language, such as C++, one will only have to replace this module. We tried this together with Rich Washington, a member of the Robotics group at NASA Ames, on a 90,000 line C++ application, just activating the deadlock detection rule, and located a deadlock. This work will be presented in a different publication.

3 Logic based Monitoring

As previously mentioned, JPAX currently allows two conceptually independent methodologies for runtime verification. One is specification based monitoring, which is the subject of this section, and the other is error pattern analysis presented in the next section. The main difference between the two is that the first counts upon an underlying logic in which the user can express any application dependent logical requirements, while the second implements more or less standard programming language dependent algorithms that detect typical concurrency error potentials. In this way, we believe that JPAX offers a large, if not a full, spectrum of possibilities for runtime verification.

In order to write a runtime requirement specification, the user should first choose an appropriate logic to express the intended properties. JPAX currently provides linear temporal logics, both future time and past time, as builtin logics, but one could relatively easily define new logics or enrich the existing ones. 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 took the liberty to include them in the paper.

3.1 The Maude Language

Maude [1] is a modularized specification and verification system that efficiently implements rewriting logic. 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. Furthermore, Maude provides support for meta-programming, so complex logic dependent reasoning strategies can be implemented as well; however, we didn't need the meta-level yet, but we expect to need it soon, as JPAX will be extended. There is not enough space to present the Maude notation in more detail here, but we'll introduce some of it "on the fly" as we give examples, such as the following one.

3.1.1 Propositional Calculus

The following module for propositional calculus, which is heavily used in JPAX, implements an efficient procedure due to Hsiang [9] to decide validity of propositions:

```
fmod PROP-CALC is pr 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 [assoc] .
  op !_    : Formula -> Formula .
  op _->_  : Formula Formula -> Formula .
  op _<->_ : Formula 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 .
*** Data structure & Semantics
  eq (X /\ Y){As*} = X{As*} /\ Y{As*} .
  eq (X ++ Y){As*} = X{As*} ++ Y{As*} .
endfm
```

The underscores stay for arguments. The module `FORMULA` which is "protected" (or imported) defines the infrastructure for all the user-defined logics. That includes some designated basic sorts (or types) such as `Formula` for syntactic formulae, `FormulaDS` for formula data structures needed when more information then the formula itself should be kept for the next transition as in the case of past time linear temporal logic, `AtomState` for assignments of boolean values to atomic propositions and `AtomState*` for assignments as above together with final assignments, i.e., those that are followed by the end of trace (our semantics for the end of the execution trace is that of a continuous process that doesn't change the state). The user is free to extend these types and/or provide appropriate implementations for them as in the module above. Perhaps the most important operation 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 this update operation acts like a morphism for propositional calculus, so it basically evaluates the propositions in the new state.

3.2 Linear Temporal Logics

Linear temporal logics (LTL) are widely accepted as reasonably good formalisms to express requirements of reactive systems. However, there is a tricky aspect of specification based monitoring which distinguishes it from other formal methods techniques, such as model checking and theorem proving: the end of trace. Sooner or later, the monitored program will be stopped and so its execution trace. At that moment, the observer needs to take a decision regarding the validity of the checked properties. Let us consider for example the formula $\Box(p \to \Diamond q)$. 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; but 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 let to continue its execution. However, there are 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. On the one hand, we learned that, unlike in model checking or theorem proving, LTL formulae and especially their violation or satisfaction must be regarded with provisions during monitoring. On the other hand, 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, or even undiscovered ones, could be of greater interest than pure LTL. In the next subsections we briefly describe our simple implementations of future time and past time LTL in Maude.

3.2.1 Future Time LTL

Future time LTL can be implemented more easily than we initially thought on top of propositional calculus. It basically needs only 8 rules, a pair for each operator:

```
fmod FT-LTL is ex PROP-CALC .
*** Syntax ***
  ops ([]_) (<>_) (o_) : Formula -> Formula .
  op _U_ : Formula Formula -> Formula .
*** Data structure & Semantics
  vars X Y : Formula .  var As : AtomState .
  eq ([] X){As  } = ([] X) /\ X{As} .
  eq ([] X){As *} = X{As *} .
  eq (<> X){As  } = (<> X) \/ X{As} .
  eq (<> X){As *} = X{As *} .
  eq (o X){As  } = X .
  eq (o X){As *} = X{As *} .
  eq (X U Y){As  } = Y{As} \/ (X{As} /\ (X U Y)) .
  eq (X U Y){As *} = Y{As *} .
endfm
```

Each pair of rules says how a formula transforms during the execution of the program. More precisely, they implement the following simple equivalences:

$$\begin{array}{rcl} st \models \varphi & \text{iff} & t \models \varphi\{s\} \\ s \models \varphi & \text{iff} & \varphi\{s*\} = \mathit{true}, \end{array}$$

where st is a trace formed by a state s followed by a nonempty trace t, while s can also be viewed as the trace consisting of s followed by the end of trace. A proof of correctness of this algorithm is given in [8]. Despite its overall exponential complexity, this algorithm tends to be quite acceptable in practical situations. We couldn't notice any sensible difference in global concrete experiments with JPaX between this simple 8 rule algorithm and an automata based one that implements in 1,400 of Java code a Buchi automata algorithm adapted to finite trace LTL (see Subsection 3.3).

3.2.2 Past Time LTL

Past time LTL is useful for especially safety properties. These properties are very suitable for logic based monitoring because once they fail we know for sure that the program is not correct. The implementation of past time LTL is a bit more tedious. 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:

```
fmod PT-LTL is ex PROP-CALC .
*** Syntax
  op ~_  : Formula -> Formula .
  op _S_ : Formula Formula -> Formula .
*** Data structure & Semantics
  op ptLtl : Formula -> FormulaDS .
  op atom : Atom Bool -> FormulaDS .
  ops prev : FormulaDS Bool -> FormulaDS .
  ops and xor since : FormulaDS FormulaDS Bool -> FormulaDS .
  vars X Y : Formula .  vars D Dx Dy D' Dx' Dy' : FormulaDS .
  var B : Bool .  var A : Atom .  var As : AtomState .
  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} .
  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 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} .
  eq atom(A,B){As *} = true .
  eq prev(D,B){As *} = true .
  eq since(Dx,Dy,B){As *} = true .
  eq and(Dx,Dy,B){As *} = true .
  eq xor(Dx,Dy,B){As *} = true .
endfm
```

The operation `ptLTL` initializes/creates the data structure associated to a past time LTL formula, the operation `[_]` reads the current truth value of a formula, while the operator _{_} updates a formula data structure.

3.3 Observer Generation

As one naturally expects, monitoring via event extraction can significantly slow down the normal execution of the monitored program. In particular, the two event buffers of JPaX (one from the instrumented program to the observer and the other from the observer to Maude) sometimes slow down the original program by an order of magnitude. We are still investigating the real reasons for this, but at this stage we believe that a significant factor comes from the buffer communication between the observer implemented in Java and the logic engine implemented in Maude. Therefore, it may be desirable to device Java implementations that directly check formulae against execution traces, at least for those logics that turn out to be heavily used.

Since JPaX only uses linear temporal logics, we concentrated only on future time and past time LTL so far. In [12] we showed how one could generate a dynamic-programming based algorithm from any future time LTL formula, showing that it runs in time $O(nm)$, where n is the size of the trace and m is the size of the formula. Unfortunately, that algorithm visits the execution trace backwards, meaning that a formula can be tested only after the program is stopped and all its execution trace stored. Fortunately, the same idea applied on past time LTL yields by dualization a forwards algorithm which runs in the same time; it is hard to believe that one can test past time LTL formulae on finite traces faster.

Taking into account the continuously decreasing price of storage, the backwards algorithm for future time LTL [12] is acceptable even beyond the prototyping stage of the tool. Our colleague Dimitra Giannakopoulou took the challenge and implemented in about 1,400 lines of Java code a modified version of a Buchi automata algorithm that takes into account the particularities of finite trace LTL; the details of her implementation will appear elsewhere. It seems that finite trace LTL is a significantly simpler and more computable logic than the standard infinite trace LTL. In particular, we were able to show the existence and then generate a minimal standard automaton from any formula, automaton that accepts exactly those finite traces that satisfy the formula; this construction will also appear elsewhere.

Our main concern at this stage is to investigate more suitable logics for monitoring than future time LTL rather than generating efficient implementations for formula checkers. The flexibility and ease in developing and/or modifying logics in rewriting logic, as well as its expressivity, efficiency and support for meta-programming, make Maude a perfect choice as a logic engine to validate user defined requirements at this early stage of JPaX.

4 Error Pattern Analysis

Error pattern analysis is conceptually based on analyzing an execution trace using various algorithms that are able to detect error potentials even though errors do not explicitely occur in the examined execution trace. The goal is to extract as much information as possible from a single execution trace to be able to suggest problems in other execution traces that have not been explored. Two examples of such algorithms focusing on concurrency errors have been implemented in JPaX: a data race analysis algorithm and a deadlock analysis algorithm. Previously, both algorithms have been implemented in the Visual Threads tool [4] to work for C and C++. Also, in recent work we implemented the data race algorithm and a variant of the deadlock algorithm in the Java PathFinder tool [5] to work for Java by modifying the Java Virtual Machine described in [14]. Our contribution here is to make these algorithms work for Java using byte-code instrumentation; to integrate them with logic based monitoring; and to make it possible for an advanced user to program new error pattern analysis rules in a flexible manner.

Error pattern analysis algorithms typically do not guarantee that errors are found since they, after all, work on a single arbitrary trace. They also may yield false positives in the sense that analysis results indicate warnings rather than hard error messages. What is attractive about such algorithms is, however, that they scale very well, and that they often seem to 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. In the following we will shortly describe the data race and deadlock detection algorithms.

4.1 Data Race Analysis

This section describes the Eraser algorithm as presented in [13], and how it has been implemented in JPaX to work on Java programs. A *data race* occurs when two concurrent threads access a shared variable and when 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 in a program by studying a single run of the program, and from this trying to conclude whether any other runs with data races are possible. We will 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 `Value` class defines an integer variable `x`, a synchronized method `add` for updating the variable (adding the contents of another `Value` variable), and an unsynchronized method `get` for reading the variable. The `Task` class is a thread class, instances of which can be started with the `start` method to execute their `run` method. Two such tasks are started in the main program on two instances of the `Value` class. 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 `Task` threads in lines 4 and 6 respectively. The problem detected is that one `Task` thread can call the `add` method on an object, say v_1, with a parameter `Value` object v_2, and this method in turn calls the unsynchronized `get` method on v_2. The other thread can simultaneously make the dual operation, hence, call the `add` method on v_2. Hence the `x` in v_2 may be accessed simultaneously by the two threads. In fact two data race warnings are emitted since the same situation is possible with v_1 and v_2 interchanged.

One could argue that all methods should be synchronized such that data races can be detected using simple type checking. However it is often seen that Java programmers avoid defining all methods as synchronized in order to optimize the program (synchronization slows down an application somewhat). Furthermore, synchronization can alternatively be achieved by executing synchronized statements such as:

```
synchronized(v1){
  v1.add(v2)
}
```

In this case it becomes impossible to detect data race potentials syntactically. This is the situation in languages such as C and C++ using Pthreads [11].

The basic algorithm works as follows. Two data structures are maintained in the observer: a *thread map* keeps track of which locks are owned by any thread at any point in time. The second data structure, a *variable map*, associates with each (shared) variable in the program at any point in time the biggest set of locks that has been commonly owned by all accessing threads in the past. If this set becomes empty a data race potential exists. That is, when a field is accessed for the first time, the locks owned by the accessing thread at that time are stored in this set. Subsequent accesses by other threads causes the set to be reduced to its intersection with the locks owned by those threads. An extra state machine is introduced for each field to keep track of how many threads have accessed the variable and how. This is used to reduce false positives in the case for example fields are initialized by a single thread without locks (which is safe) or several threads just read a variable after it has been initialized (which is also safe). Amongst the events that are important for the data race analysis are monitor lockings and releases; either resulting from executing Java's `synchronized` statements or from calling/returning from synchronized methods. Furthermore, all accesses to field variables and class variables are analyzed.

Deadlock Detection

A classical deadlock situation can occur where two threads t_1 and t_2 share two locks v_1 and v_2, and they take the locks in different order. The deadlock will arise if t_1 takes v_1 and t_2 immediately after takes v_2. Now t_1 cannot get v_2 and t_2 cannot get v_1. Using the previous example, we can create such a situation if we wrongly try to repair the data race by defining the `get` method in line 6 as synchronized:

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

Now the `x` variable can no longer be accessed simultaneously from two threads, and the data race module will no longer give a warning. However, when running JPAX on the modified program, a lock order problem not present before is found and a warning states that two object instances of the `Value` class are taken in a different order by the two `Task` threads, and it indicates the line numbers where the threads may potentially deadlock, hence where the access to the second lock may fail: line 4 where the call of the `get` method from the `add` method will lock the second object. Note that this deadlock does not need to occur in the execution in order for this warning to be issued. In fact, any execution of this example program will cause a warning to be issued.

The algorithm works as follows. Two data structures are maintained in the observer: as in the data race algorithm a *thread map* keeps track of which locks are owned by any thread at any point in time. The second data structure, a *lock graph*, maintains an accumulating graph of all the locks taken by threads during an execution, recording locking orders as edges. That is, an edge is introduced from a lock v_1 to a lock v_2 in case a thread owns v_1 while taking v_2. If this graph ever becomes cyclic it reflects a deadlock potential. Note that this algorithm can catch deadlock potentials between many threads as illustrated for example by the classical dining philosopher's example, independent of the number of philosophers. The events that are important for the data race analysis are monitor lockings and releases; either resulting from executing Java's `synchronized` statements or from calling/returning from synchronized methods.

5 Conclusions

A brief description of JPAX, a runtime verification environment currently under experimentation and development, was presented. Motivation and integration in the overall NASA Ames automated software engineering effort was highlighted, emphasizing that our main goal was 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. A general system architecture of JPAX was depicted, followed by more detailed explanations of its components, especially logic based monitoring and error pattern analysis.

In future work on logic based monitoring, 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. A longer term agenda is oriented toward fast implementations of designated logics in more conventional programming languages than Maude, thus improving the overall speed of the monitoring process. Future work on error pattern analysis will try to develop new algorithms for detecting other kinds of concurrency errors 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; and guidance of the program during operation once a requirement specification has been violated. Dynamic programming 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, in addition to an improved modularization of the whole system such that to easily adapt it to various programming languages and various instrumenting methodologies. Of course, the tool will be evaluated on real case studies.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming.* O'Reilly, 1998.

[12] 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.

[13] 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.

[14] 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.