Specification and Error Pattern Based Program Monitoring

Klaus Havelund[1], **Scott Johnson**[2], **Grigore Roşu**[3]

[1] ***Kestrel Technology, NASA Ames Research Center, USA***
Email: havelund@email.arc.nasa.gov

[2] ***Department of Computer Science, University of Wyoming, USA***
Email: skj@uwyo.edu

[3] ***RIACS, NASA Ames Research Center, USA***
Email: grosu@email.arc.nasa.gov

Abstract

We briefly present Java PathExplorer (JPaX), a tool developed at NASA Ames for monitoring the execution of Java programs. JPaX can be used not only during program testing to reveal subtle errors, but also can be applied during operation to survey safety critical systems. The tool facilitates automated instrumentation of a program in order to properly observe its execution. The instrumentation can be either at the bytecode level or at the source level when the source code is available. JPaX is an instance of a more general project, called PathExplorer (PaX), which is a basis for experiments rather than a fixed system, capable of monitoring various programming languages and experimenting with other logics and analysis techniques.

1 Introduction

The success of most technological experiments today, including space craft and rover technology, heavily depends on the correctness of software. Two common ways to approach the delicate problem of software correctness are *testing* and *formal methods*. Although traditional testing techniques scale well and are by far the most used techniques in practice to validate software systems, they are usually *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 program synthesis [14, 5, 18] and program verification [8, 9, 17, 7, 11]. This paper, together with [10, 15, 11, 12] describes our effort in *runtime verification*, which can be roughly defined as combining testing and formal methods. 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. We are considering two apparently distinct approaches to monitoring which can be well combined. In *black-box monitoring*, the general idea consists of extracting state events from an executing program, which has been automatically instrumented, and then analyzing them via an observer process. This approach is suitable in general when certain fields (variables, data structures, etc.) need to be observed at any moment over the execution of the program. In *white-box monitoring*, one assumes that the source code of the program is available and that annotations can be introduced manually in the code at appropriate places, where they will then be expanded and compiled together with the program. This approach is more appropriate when certain fields need to be observed at specific places in the program. Many of the algorithms used are the same in both black-box and white-box approaches. JPaX currently performs two kinds of verification, namely *logic based monitoring* and *error pattern analysis*.

Logic based monitoring consists of runtime checking formal requirement specifications written in high level logics by users of the system. Logics are currently implemented in Maude [1], a high-performance system supporting both rewriting logic and membership equational logic. One can naturally and easily define new logics in Maude, such as for example temporal logics, together with their finite trace operational semantics. Currently, JPaX supports two builtin logics, future time and past time linear temporal logic. *Logic based* *monitoring* can be instantiated at either the source code level in the case of white-box monitoring, or at the byte code level in the case of black-box monitoring. The idea of using temporal logic in program testing is not new. We only mention the commercial Temporal Rover tool (TR) [3] which implements temporal logics in a white-box fashion, and the MaC tool [13] which implements a special past time temporal logic in a black-box style.

Error pattern analysis consists of analyzing one execution trace 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, but it is worth mentioning that these algorithms are usually neither sound nor complete for the class of errors they are designed for. They are generally 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 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

We next shortly describe JPaX's architecture. Unless otherwise specified, the black-box monitoring approach is considered. We use the acronym JPaXx (Java PathExplorer Expander) when we refer to the white-box monitoring aspect of JPaX. White-box monitoring can be used in conjunction with black-box monitoring and works as follows: after one annotates the source code with desired formal requirements, one passes that source file through the white-box monitoring filter, JPaXx, and the annotations will be then expanded automatically into Java source code using optimal algorithms mentioned later in the paper, where after the expanded code may be compiled as normal and executed; exceptions will be thrown when formulae are violated.

We next discuss the architecture of the black-box monitoring part of our system. JPaX can be regarded as consisting of three main modules: an *instrumentation* module, an *observer* module, and an *interconnection* module that ties the other two 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 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.

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 [2] 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 [1] is a modularized specification and verification system that very efficiently implements 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.

3.1 Future Time LTL Semantics

We first present a semantics of finite trace LTL using standard mathematical notation. Assume two total functions on traces, *head* : `Trace` $\rightarrow$ `Event` returning the head event of a trace and *length* returning the length of a finite trace, and a partial function *tail* : `Trace` $\rightarrow$ `Trace` for taking the tail of a trace. That is, $head(e,t) = head(e) = e$, $tail(e,t) = t$, and $length(\mathtt{e}) = 1$ and $length(e,t) = 1 + length(t)$. Assume further for any trace t, that t_i denotes the suffix trace that starts at position i, with positions starting at 1. The satisfaction relation $\models \subseteq$ `Trace` $\times$ `Formula` defines when a trace t satisfies a formula f, written $t \models f$, and is defined inductively over the structure of the formulae as follows, where `A` is any atomic proposition and `X` and `Y` are any formulae, and *op* are standard binary propositional connectives($\vee$, and, $\neg$, $\rightarrow$) and `[]`, `<>`, `()`, `U` stand for *always* in the future, *eventually* in the future, *next*, and *until* respectively:

$$
\begin{array}{lll}
t \models \mathtt{A} & \text{iff} & \mathtt{A} \in head(t) \\
t \models \mathtt{true} & \text{iff} & true, \\
t \models \mathtt{false} & \text{iff} & false, \\
t \models \neg\ \mathtt{X} & \text{iff} & \text{not } t \models \mathtt{X}, \\
t \models \mathtt{X\ op\ Y} & \text{iff} & t \models \mathtt{X} \text{ op } t \models \mathtt{Y}, \\
t \models \mathtt{[]X} & \text{iff} & (\forall\ i \leq length(t))\ t_i \models \mathtt{X} \\
t \models \mathtt{<>X} & \text{iff} & (\exists\ i \leq length(t))\ t_i \models \mathtt{X} \\
t \models \mathtt{X\ U\ Y} & \text{iff} & (\exists\ i \leq length(t))\ (t_i \models \mathtt{Y} \text{ and } (\forall\ j < i)\ t_j \models \mathtt{X}) \\
t \models \mathtt{()X} & \text{iff} & (\text{if } tail(t) \text{ is defined then } tail(t) \models \mathtt{X} \text{ else } t \models \mathtt{X})
\end{array}
$$

Notice that liveness properties do not really make sense in finite trace LTL without statistical analysis, as already noticed in [12, 4]. 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 mentioned in subsection 3.3 proved that.

3.2 Past Time LTL Semantics

Here we present the semantics of finite trace past time temporal logic. For *past* time the definitions change as follows: $last(t,e) = last(e) = e$, $front(t,e) = t$, and $length(\mathtt{e}) = 1$ and $length(t,e) = 1 + length(t)$. Assume further for any trace t, that t_i denotes the prefix trace that ends at position i. The satisfaction relation $\models \subseteq \mathtt{Trace} \times \mathtt{Formula}$ defines when a trace t satisfies a formula f, written $t \models f$, and is defined inductively over the structure of the formulae as follows, where `A` is any atomic proposition and `X` and `Y` are any formulae, *op* are standard binary propositional connectives($\vee$, and, $\neg$, $\rightarrow$), and `[*]`, `<*>`, `(*)`, `S` stand for *always* in the past, *eventually* in the past, *previous*, and *since* respectively:

$t \models \mathtt{A}$	iff	$\mathtt{A} \in last(t)$
$t \models \mathtt{true}$	iff	*true*,
$t \models \mathtt{false}$	iff	*false*,
$t \models \neg\ \mathtt{X}$	iff	not $t \models \mathtt{X}$,
$t \models \mathtt{X\ op\ Y}$	iff	$t \models \mathtt{X}$ op $t \models \mathtt{Y}$,
$t \models \mathtt{[*]X}$	iff	$(\forall\ i \leq length(t))\ t_i \models \mathtt{X}$
$t \models \mathtt{<*>X}$	iff	$(\exists\ i \leq length(t))\ t_i \models \mathtt{X}$
$t \models \mathtt{X\ S\ Y}$	iff	$(\exists\ i \leq length(t))\ (t_i \models \mathtt{Y}$ and $(\forall\ j < i)\ t_j \models \mathtt{X})$
$t \models \mathtt{(*)X}$	iff	(if $front(t)$ is defined then $front(t) \models \mathtt{X}$ else $t \models \mathtt{X}$)

3.3 Efficient Observer Generation

After experimenting with runtime verification algorithms for LTL [10, 15, 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 are some priorities that we also discussed in [12] which currently influence the choice of runtime algorithms in JPAX : 1) *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, so we exclude them; 2) *runtime efficiency* - the normal execution of the program should be influenced as little as possible, so we exclude algorithms that are worse than linear in the size of the trace and prefer algorithms that are not exponential in the size of formula if possible; 3) *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.

Next we 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. The interested reader can consult [12] for more detail.

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: 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 generated by our current implementation:

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

Notice that the generated finite state machines are deterministic. The drawback of generating deterministic machines is that they may be quite large for some formulae, even large enough that one cannot afford to generate and store them in practical situations. For that reason, we intend to also attach a module that generates nondeterministic automata from LTL formulae, like in [4], to JPAX to be activated when the deterministic one cannot be generated in a reasonable amount of time.

3.4 White-box Monitoring

The procedure above can be used in instrumented bytecode monitoring (black-box) as well as in source code level monitoring (white-box). If the source code is available, the second approach may be more desired for two reasons: exception handling and efficiency. With regards to exceptions, white-box logic monitoring can now be seen as a natural extension to the language that allows one to describe conditions that should raise exception, as already noticed in [3].

As stated earlier, the worst-case runtime complexity for future time LTL monitoring is $O(nk)$, where n is the length of the trace and k is the number of variables of the formula. For the past time monitoring, the worst-case runtime complexity becomes $O(nm)$, where n is the same as above and m is the number of subformulae of the formula. We conjecture that the complexity of past time monitoring can be further improved.

To demonstrate the tool we use a very simple example involving the monitoring of a traffic light controller. The following is an annotation (left) and the expanded form of that annotation (right).

```
...
<code>
...
/* JPaX: after green comes yellow

   Atom isRed = <boolean exp>;
   Atom isGreen = <boolean exp>;
   Atom isYellow = <boolean exp>;

   Formula :
   []( isGreen -> ( ! isRed U isYellow ));
*/
...
<code>
...
```

```
...
<code>
...
try {
  switch(bttFsmState) {
  case 1 :
   bttFsmState = isGreen ? isYellow ? 1 : isRed ? 0 : 2 : 1; break;
  case 2 :
   bttFsmState = isYellow ? 1 : isRed ? 0 : 2; break;
}
if(bttFsmState == 0) throw new Exception("Prop. Failure");
} catch(Exception e) { System.out.println(e.getMessage()); }
...
<code>
...
```

The annotation is commented out in a normal Java comment but contains the keyword `JPaX` followed by a title, a list of atomic declarations, and an LTL formula. Here the formula states "it is always the case that if the light is green it will not be red again until it has first been yellow". Now this new code may be compiled normally and will raise an exception if the property is violated.

Next we look at the same example, but now we express the property in *past* time finite trace LTL. Here we took the dynamic programming algorithm for future time LTL presented in [15] and converted it to past time LTL. We are planning a follow up paper that will discuss these algorithms in more detail. Once again the annotation is on the left and the expanded annotation is on the right. Not seen in this example is the initialization of the array `prev`, which holds the evaluation of a particular subformula in the previous state. Here the formula states "if the light is red then the light has not been green since it was yellow".

```
...
<code>
...
/* JPaX: after green comes yellow

      Atom isRed = <boolean exp>;
      Atom isGreen = <boolean exp>;
      Atom isYellow = <boolean exp>;

      Formula :
       isRed -> ( ! isGreen S isYellow);
*/
...
<code>
...
```

```
...
<code>
...
now[6] = isGreen;
now[5] = isYellow;
now[4] = ! now[6];
now[3] = now[5] || (now[4] && prev[3]);
now[2] = isRed;
now[1] = ! now[2] || now[3];
prev[3] = now[3];
if (! prev[1]) { throw new Exception("Prop. Failure");}
...
<code>
...
```

The ordering of the above statements is the result of parsing the formula and then visiting each subformula in the parse tree in a breadth first ordering from the bottom up. When the expanded code is executed, each statement is evaluated to determine whether the subformula holds in that state or not.

4 Error Pattern Analysis

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 [16] 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++.

4.1 Data Race Analysis

We briefly describe here how easily data races can occur in concurrent programming and how Eraser [16] 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 dual thing, i.e., call the `add` method on `v2`. Therefore, the content of `v2` might be accessed simultaneously by the two threads.

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 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. A white-box logic based monitoring approach has also been presented. 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.

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. 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] 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] Bernd Finkbeiner and Henny Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of the First International Workshop on Runtime Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[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] Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the First International Workshop on Runtime Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[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] 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.

[16] 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.

[17] 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.

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