

# Automated Testing using Symbolic Model Checking and Temporal Monitoring

Cyrille Artho <sup>a</sup>, Howard Barringer <sup>b</sup>, Allen Goldberg <sup>c</sup>,  
Klaus Havelund <sup>c</sup>, Sarfraz Khurshid <sup>d</sup>, Mike Lowry <sup>e</sup>,  
Corina Pasareanu <sup>c</sup>, Grigore Roşu <sup>f</sup>, Koushik Sen <sup>f</sup>, Willem Visser <sup>g</sup>,  
Rich Washington <sup>g</sup>

<sup>a</sup>*Computer Systems Institute, ETH Zurich, Switzerland*

<sup>b</sup>*University of Manchester, England*

<sup>c</sup>*Kestrel Technology, NASA Ames Research Center, USA*

<sup>d</sup>*MIT Computer Science and Artificial Intelligence Laboratory, USA*

<sup>e</sup>*NASA Ames Research Center, USA*

<sup>f</sup>*Department of Computer Science, Univ. of Illinois at Urbana-Champaign, USA*

<sup>g</sup>*RIACS, NASA Ames Research Center, USA*

---

## Abstract

Software testing is typically an ad hoc process where human testers manually write test inputs and descriptions of expected test results, perhaps automating their execution in a regression suite. This process is cumbersome and costly. This paper reports results on a framework to further automate this process. The framework consists of combining automated test case generation based on systematically exploring the program's input domain, with runtime analysis, where execution traces are monitored and verified against temporal logic specifications, and analyzed by concurrency error detection algorithms. The approach suggests a methodology for generating specifications dynamically for each input instance rather than statically once-and-for-all. This approach of generating properties specific to a single test case is novel. The paper describes an application of this methodology to a planetary rover controller.

### *Key words:*

Automated testing, test-case generation, model checking, symbolic execution, runtime analysis, temporal logic monitoring, concurrency analysis, C++, planetary rover controller.

---

## 1 Introduction

A program is typically tested by manually creating a *test suite*, which in turn is a set of *test cases*. An individual test case is a description of a *test input sequence* to the program, and a description of *properties* that the corresponding output is expected to have. This manual procedure may be unavoidable since for real systems, writing test cases is an inherently innovative process requiring human insight into the logic of the application being tested. Discussions with robotics and space craft engineers at NASA seems to support this view. However, an equally widespread opinion is that a non-trivial part of the testing work *can* be automated. In a case study, an 8,000-line Java application was tested by different student groups using different testing techniques [4]. It is conjectured that the vast majority of bugs in this system that were found by the students could have been found in a fully automatic way. The paper presents work on applying low-overhead automated testing to identify bugs quickly. We suggest a framework for generating test cases in a fully automatic way as illustrated by Figure 1. For a particular program to be tested, one establishes a test harness consisting of four modules: a *test input generator module*, a *property generator module*, a *program instrumentation module* and an *observer module*.

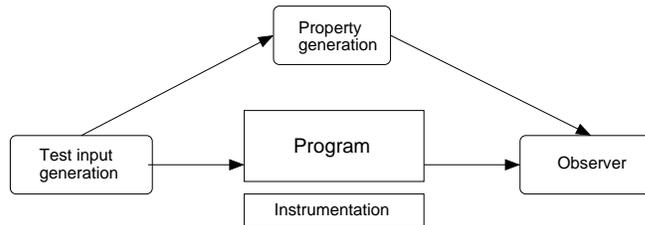


Figure 1. Test case generation (test input generation and property generation) and runtime analysis (instrumentation and observation).

The test input generator automatically generates inputs to the application, one by one. A generated input is fed to the the property generator, which automatically generates a set of properties that the program should satisfy when executed on that input. The input is then fed to the program, which executes, generating an execution trace. The observer module checks the behavior of the executed program against the generated set of properties. Hence, it takes an execution trace and the set of properties generated by the property generator as input. The program itself must be instrumented to report events that are relevant for monitoring that the properties are satisfied on a particular execution. This instrumentation can in some cases be automated. The test input generator and the property generator are both written (“hard-wired”) specifically for the application that is tested. This replaces manual construction of test cases. However, the observer module is generic and can be re-used on different applications. In the rest of this paper the term *test case generation* is used to refer to test input generation and property generation, and the term *runtime analysis* to refer to instrumentation as well as observation.

The framework described above has been applied to a case study, a planetary rover

controller. Test cases are generated using a model checker and the properties generated are *specific to a single test case*. Properties are expressed in temporal logic. The approach of generating properties specific to a single test case is novel.

The paper is organized as follows. Section 2 outlines our technology for test case generation: symbolic execution and model checking. Section 3 describes the runtime analysis techniques: temporal logic monitoring and concurrency analysis. Section 4 describes the case study, where these technologies are applied to a planetary rover controller. Finally Section 5 concludes the paper and outlines how this work will be continued.

## 2 Test Case Generation

This section presents the test case generation framework. As mentioned earlier, test generation is considered as consisting of *test input generation* and *property generation*.

### 2.1 Test Input Generation

#### 2.1.1 Model based testing

In practice today, the generation of test inputs for a program under test is a time-consuming and mostly manual activity. However, test input generation naturally lends itself to automation, and therefore has been the focus of much research attention – recently it has also been adopted in industry [18,23,7,10]. There are two main approaches to generating test inputs automatically: a static approach that generates inputs from some kind of model of the system (also called model-based testing), and a dynamic approach that generates tests by executing the program repeatedly, while employing criteria to rank the quality of the tests produced [16,22]. The dynamic approach is based on the observation that test input generation can be seen as an optimization problem, where the cost function used for optimization is typically related to code coverage (e.g. statement or branch coverage). The model-based test input (test case) generation approach is used more widely (see Hartman’s survey of the field [12]). The model used for model-based testing is typically a model of expected system behavior and can be derived from a number of sources, namely, a model of the requirements, use cases, design specifications of a system [12] – even the code itself can be used to create a model (e.g. approaches based on symbolic execution [15,18]). As with the dynamic approach, it is most typical to use some notion of coverage of the model to derive test inputs, i.e., generate inputs that cover all transitions (or branches, etc.) in the model.

On the one hand, constructing a model of the expected system behavior can be a costly process. On the other hand, generating test inputs just based on a specification of the input structure and input pre-conditions can be very effective, while typically less costly. In [14] a verification framework is presented that combines *symbolic execution* and model checking techniques in a novel way. The framework can be used for test input generation as follows: the input model is described as a non-deterministic program annotated with constraints that describes all valid inputs, and the model checker is used to traverse the (symbolic) state space of this program. As the property the model checker should check for, one asserts that no such test input exists – this causes the model checker to produce a counter-example whenever a valid test input has been generated. From this counter-example trace the test input is produced. It is important that various techniques for searching the state space should be available since this gives the flexibility to generate a large array of test inputs to achieve better coverage of the behavior of the system under test. For test input generation the Java PathFinder model checker (JPF) is used that analyzes Java programs [24] and supports various heuristic search strategies (for example, based on branch coverage [11] or random search). In Section 4.2 we show how this model checker is used to generate test inputs for the Mars K9 rover.

Using symbolic execution for test case generation is a well-known approach, but typically only handles sequential code with simple data. In previous work, this technique has been extended to handle complex data-structures (e.g. lists and trees), concurrency as well as linear constraints on integer data [14]. Symbolic execution of a program path results in a set of constraints that define program inputs that execute the path; these constraints can often be solved using off-the-shelf decision procedures to generate concrete test inputs. When the program represents an executable specification, symbolic execution of the specification enables us to generate inputs that give us, for instance, full specification coverage. Note that these specifications are typically not very large – no more than a few thousand lines, in our experience – and hence will allow efficient symbolic execution.

The most closely related work to ours is the Korat tool [3] that generates test inputs from Java predicates, but instead of model checking they use a dedicated, efficient, search strategy. The use of the counter-example capability of a model checker to generate test inputs have also been studied by many others (see [13] for a good survey), but most of these are based on a full system model, not just the input structure and pre-conditions as suggested here.

### 2.1.2 *Symbolic Execution for Test Input Generation*

The enabling technology for black-box test-input generation from an input specification is the use of symbolic execution. Optionally the system under test itself can be symbolically executed, for white-box testing, the techniques are in fact the same. The main idea behind symbolic execution [15] is to use symbolic values, instead

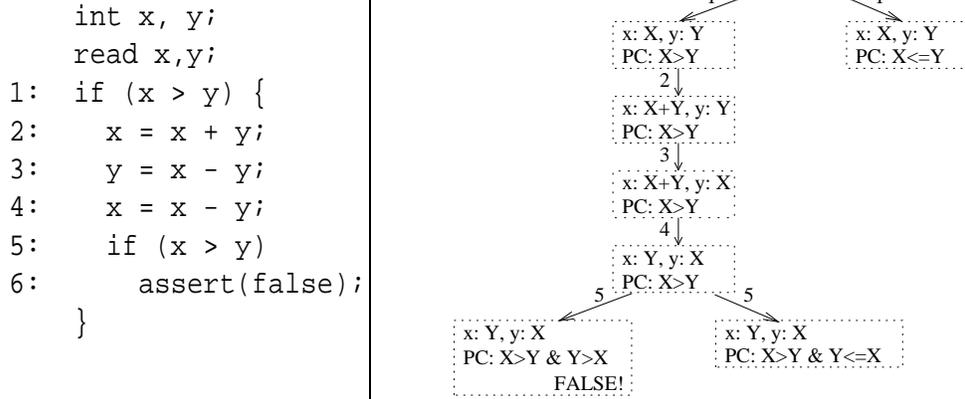


Figure 2. Code for swapping integers and corresponding symbolic execution tree.

of actual data, as input values, and to represent the values of program variables as symbolic expressions. The state of a symbolically executed program includes, in addition to the (symbolic) values of program variables and the program counter, a path condition. The path condition is a (quantifier-free) Boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Consider as an example (taken from [14]) the code fragment in Figure 2, which swaps the values of integer variables  $x$  and  $y$ , when  $x$  is greater than  $y$ . Figure 2 also shows the corresponding symbolic execution tree. Initially, the path condition, PC, is *true* and  $x$  and  $y$  have symbolic values  $X$  and  $Y$ , respectively. At each branch point, PC is updated with assumptions about the inputs according to the alternative (possible) paths. For example, after the execution of the first statement, both then and else alternatives of the *if* statement are possible, and PC is updated accordingly. If the path condition becomes *false*, i.e., there is no set of inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for that path. For example, statement (6) is unreachable. In order to find a test input to reach branch statement (5) one needs to solve constraint  $X > Y$  – e.g. make inputs  $x$  and  $y$  1 and 0, respectively.

Symbolic execution traditionally arose in the context of sequential programs with a fixed number of integer variables. We have extended this technique to handle dynamically allocated data structures (e.g. lists and trees), complex preconditions (e.g. lists that have to be acyclic), other primitive data (e.g. strings) and concurrency. A key feature of our algorithm is that it starts the symbolic execution of a procedure on *uninitialized* inputs and it uses *lazy initialization* to assign values to these inputs, i.e., it initializes parameters when they are first accessed during the procedure’s symbolic execution. This allows symbolic execution of procedures

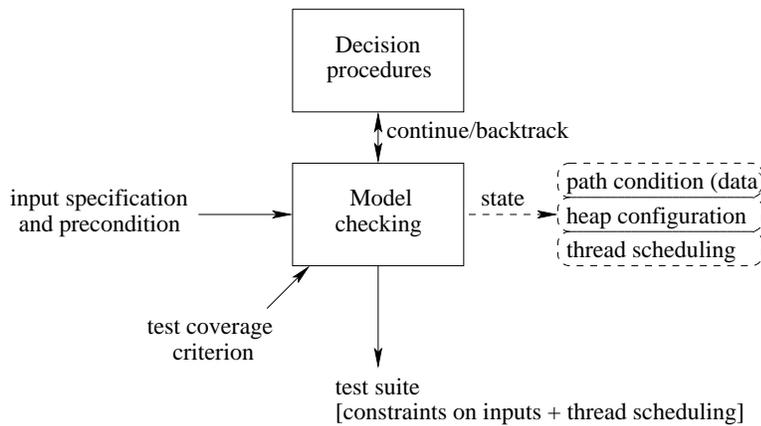


Figure 3. Framework for test input generation.

without requiring an a priori bound on the number of input objects. Procedure pre-conditions are used to initialize inputs only with valid values.

As mentioned before our symbolic execution-based framework is built on top of the Java PathFinder (JPF) model checker [24]. JPF is an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). It can handle all of the language features of Java, and in addition it treats non-deterministic choice expressed in annotations of the program being analyzed. For symbolic execution the model checker was extended to allow backtracking whenever a path-condition is unsatisfiable (determined by calling a decision procedure).

### 2.1.3 Framework for Test Input Generation

Figure 3 illustrates our framework for test input generation. The input specification is given as a non-deterministic Java program that is instrumented to add support for manipulating formulas that represent path conditions. The instrumentation allows JPF to perform symbolic execution. Essentially, the model checker explores the (symbolic) state space of the program (for example, the symbolic execution tree in Figure 2). A symbolic state includes a heap configuration, a path condition on integer variables, and thread scheduling information. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure; currently our system uses the Omega library [19] that manipulates linear integer constraints. If the path condition is unsatisfiable, the model checker backtracks. A testing coverage criterion is encoded in the property the model checker should check for. This causes the model checker to produce a counter-example whenever a valid (symbolic) test input has been generated. From this trace a (concrete) test input is produced. Since only input variables are allowed to be symbolic, all constraints that are part of a counter-example are described in terms of inputs, and finding a solution to these constraints will allow a valid set of test data to be produced. Currently a simple approach is used to find these solutions. Only the first

solution is considered. In future work we will refine the solution discovery process to also consider characteristics such as boundary cases.

Currently, the model checker is not required to perform state matching, since state matching is, in general, undecidable when states represent path conditions on unbounded data. It is also important that performing symbolic execution on programs with loops can explore infinite execution trees (and it might not terminate). Therefore, for systematic state space exploration, limited depth-first search or breadth-first search is used; our framework also supports heuristic-based search [11].

## 2.2 *Property Generation*

Any verification activity is in essence a consistency check between two artifacts. In the framework presented here the check is between the execution of the program on a given input, and an automatically generated specification for that given input, consisting of a set of properties about the corresponding execution trace. In other contexts it may be a check of the consistency between the program and a complete specification of the program under all inputs. This redundancy of providing a specification in addition to the program is expensive but necessary. The success of a verification technology partly depends on the cost of producing the specification. The hypothesis of this work is twofold. First, focusing on the test effort itself and writing “testing oriented” properties, rather than a complete formal specification may be a cheaper development process. Second, automatically generating the specification from the input may be easier than writing a specification for all inputs.

More precisely, the artifact produced here is a program that takes as input an input to a program and generates a set of properties, typically assertions in linear temporal logic. The assertions are then checked against each program execution using the runtime analysis tools described in Section 3. For the case study presented in Section 4, writing this program was straightforward, and considerably easier than writing a single set of properties relevant to *all* inputs.

Notice that this approach leverages the runtime monitoring technology to great effect, just as test case generation leverages model checking and symbolic analysis. In addition, we anticipate the development of property generation tools specific to a domain or class of problems. The software under test in our case study is an interpreter for a plan execution language. In this circumstance, the program to generate properties uses the decomposition of the plan with respect to the grammar of the plan language. Like a trivial compiler, the property generator produces test-input-specific properties as semantic actions corresponding to the parse. Several of NASA’s software systems have an interpreter structure, and it is anticipated that this testing approach can be applied to several of these as well.

### 3 Runtime Verification

Many different languages and logics have been proposed for specifying and analyzing properties of program state or event traces, each with characteristics that make it more or less suitable for expressing various classes of trace properties; they range from stream-based functional, single assignment and dataflow languages, through regular (and extended regular) expression based pattern-matching languages, to a whole host of modal and, in particular, linear-time temporal logics. In Section ??, such languages and logics that have been applied directly to run-time property analysis are discussed more fully. Suffice it to say here that the general framework of linear-time temporal logics (LTL) appeared most appropriate for our own work but that none of the proposed temporal logics for run-time analysis, of which we were aware, provided the right combination of expressivity, naturality, flexibility, effectiveness, convenience and ease of use we desired. Of course, more often than not, it can be observed that the greater the expressivity of the property specification logic, the higher the computational cost for its analysis. As a consequence this has led us in the past to research efficient algorithms for the evaluation of restricted sub-logics, e.g. pure past time LTL, pure future LTL, extended regular expressions, metric temporal logic, and so forth. But we were dissatisfied that (i) we had no unifying base logic from which these different temporal logics could be built and (ii) we were overly restrictive on the way properties could be expressed, e.g. forcing pure past, or pure future, etc.. Our research thus led us to develop and implement a core, discrete temporal logic, EAGLE, that supports recursively defined formulas, parameterisable by both logical formulas and data expressions, over a set of four primitive modalities corresponding to the “next”, “previous”, “concatenation” and “sequential temporal composition” operators. The logic, whilst primitive, is expressively rich and enables users define their own set of more complex temporal predicates tuned to the particular needs of the run-time verification application. Indeed, in the paper [?] we have shown how a range of finite trace monitoring logics, including future and past time temporal logic, extended regular expressions, real-time and metric temporal logics, interval logics, forms of quantified temporal logics and context free temporal logics, can be embedded within EAGLE. However, in order to be truly fit for purpose, the implementation of EAGLE must ensure that “users only pay for what they use”.

#### 3.1 Syntax of EAGLE

The syntax of EAGLE is shown in Figure 4. A specification  $S$  consists of a declaration part  $D$  and an observer part  $O$ . The declaration part,  $D$ , comprises zero or more rule definitions  $R$ , and similarly, the observer part,  $O$ , comprises zero or more monitor definitions  $M$ , which specify the properties that are to be monitored. Both rules and monitors are named ( $N$ ), however, rules may be recursively defined, whereas

$$\begin{aligned}
S &::= DO \\
D &::= R^* \\
O &::= M^* \\
R &::= \{\mathbf{max} \mid \mathbf{min}\} N(T_1 x_1, \dots, T_n x_n) = F \\
M &::= \mathbf{mon} N = F \\
T &::= \mathbf{Form} \mid \textit{primitive type} \\
F &::= \mathbf{True} \mid \mathbf{False} \mid x_i \mid \textit{expression} \\
&\quad \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid F_1 \leftrightarrow F_2 \\
&\quad \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid F_1; F_2 \mid N(F_1, \dots, F_n) \mid x_i
\end{aligned}$$

Figure 4. Syntax of EAGLE

monitors are simply non-recursive formulas. Each rule definition  $R$  is preceded by a keyword **max** or **min**, indicating whether the interpretation given to the rule is either maximal or minimal. Rules may be parameterized; hence a rule definition may have formal arguments of type **Form**, representing formulas, or of primitive type **int**, **long**, **float**, etc., representing data values.

An atomic formula of the logic is either a logical constant **True** or **False**, or a boolean expression over the observer state, or a type correct formal argument  $x_i$ , i.e. of type **Form** or of primitive type **bool**. Formulas can be composed in the usual way through the traditional set of propositional logic connectives,  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\rightarrow$  and  $\leftrightarrow$ . Temporal formulas are then built using the two monadic temporal operators,  $\bigcirc F$  (in the next state  $F$  holds) and  $\odot F$  (in the previous state  $F$  holds), and the dyadic temporal operators,  $F_1 \cdot F_2$  (concatenation) and  $F_1; F_2$  (sequentially compose). Importantly, a formula may also be the recursive application of a rule to some appropriately typed actual arguments. That is, an argument of type **Form** can be any formula, with the restriction that if the argument is an expression, it must be of boolean type; an argument of a primitive type must be an expression of that type.

The body of a rule/monitor is thus a (boolean-valued) formula of the syntactic category *Form* (with meta-variables  $F$ , etc.). We further require that any recursive call on a rule is strictly guarded by a temporal operator.

### 3.2 Semantics of EAGLE

The models of our logic are observation (or execution) traces. An observation trace  $\sigma$  is a finite sequence of observed program states  $\sigma = s_1 s_2 \dots s_n$ , where  $|\sigma| = n$  is the length of the trace. Note that the  $i^{\text{th}}$  state  $s_i$  of a trace  $\sigma$  is denoted by  $\sigma(i)$  and

the term  $\sigma^{[i,j]}$  denotes the sub-trace of  $\sigma$  from position  $i$  to position  $j$ , both positions being included. The semantics of the logic is then defined in terms of a satisfaction relation between observation traces and specifications. That is, given a trace  $\sigma$  and a specification  $D O$ , satisfaction is defined as follows:

$$\sigma \models_D O \text{ iff } \forall (\mathbf{mon} N = F) \in O . \sigma, 1 \models_D F$$

A trace satisfies a specification if the trace, observed from position 1 - the index of the first observed program state - satisfies each monitored formula. The definition of the satisfaction relation  $\models_D \subseteq (\text{Trace} \times \mathbf{nat}) \times \mathbf{Form}$ , for a set of rule definitions  $D$ , is defined inductively over the structure of the formula and is presented in Figure 5. First of all, note that the satisfaction relation  $\models_D$  is actually defined

$$\begin{array}{ll}
\sigma, i \models_D \text{exp} & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \text{evaluate}(\text{exp})(\sigma(i)) == \text{true} \\
\sigma, i \models_D \mathbf{True} & \\
\sigma, i \not\models_D \mathbf{False} & \\
\sigma, i \models_D \neg F & \text{iff } \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff } \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \vee F_2 & \text{iff } \sigma, i \models_D F_1 \text{ or } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \rightarrow F_2 & \text{iff } \sigma, i \models_D F_1 \text{ implies } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \leftrightarrow F_2 & \text{iff } \sigma, i \models_D F_1 \text{ is equivalent to } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff } i \leq |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \odot F & \text{iff } 1 \leq i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff } \exists j \text{ s.t. } i \leq j \leq |\sigma| + 1 \text{ and } \sigma^{[1, j-1]}, i \models_D F_1 \text{ and } \sigma^{[j, |\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D F_1 ; F_2 & \text{iff } \exists j \text{ s.t. } i < j \leq |\sigma| + 1 \text{ and } \sigma^{[1, j-1]}, i \models_D F_1 \text{ and } \sigma^{[j-1, |\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D N(F_1, \dots, F_m) & \text{iff } \left\{ \begin{array}{l} \text{if } 1 \leq i \leq |\sigma| \text{ then:} \\ \quad \sigma, i \models_D F[x_1 \mapsto F_1, \dots, x_m \mapsto F_m] \\ \quad \text{where } (N(T_1 x_1, \dots, T_m x_m) = F) \in D \\ \text{otherwise, if } i = 0 \text{ or } i = |\sigma| + 1 \text{ then:} \\ \quad \text{rule } N \text{ is defined as } \mathbf{max} \text{ in } D \end{array} \right.
\end{array}$$

Figure 5. Definition of  $\sigma, i \models_D F$  for  $0 \leq i \leq |\sigma| + 1$  for some trace  $\sigma = s_1 s_2 \dots s_{|\sigma|}$

for the index range  $0 \leq i \leq |\sigma| + 1$  and thus provides a value for a formula before the start of observations and also after the end of observations - this approach was taken to fit with our model of program observation and evaluation of monitoring

formulas, i.e. you only know the end has been reached when you've passed it and no more observation states are forthcoming - and it is at that point that a value for the formula may need to be determined. At these boundary points, expressions involving reference to the observation state (where no state exists) are trivially false. A next time (resp. previous time) formula also evaluates false at the point beyond the end (resp. before the beginning). A rule, however, has its value at such points determined by whether it is maximal, in which case it is true, or minimal, in which case it is false. Indeed, there is a correspondence between this evaluation strategy and maximal (minimal) fixed point solutions to the recursive definitions. Thus, for example, the formula  $\text{Always}(\phi)$  will evaluate to true on an empty trace - since  $\text{Always}$  was defined maximal, whereas the formula  $\text{SometimeP}(\phi)$  will evaluate to false on an empty trace - as it was declared as minimal.

The propositional connectives are given their usual interpretation. The next and previous time temporal operators are as expected. The concatenation and sequential temporal composition operators are, however, not standard in linear temporal logics, although the sequential temporal composition often features in interval temporal logics and can also be found in process logics. A concatenation formula  $F_1 \cdot F_2$  is true if and only if the trace  $\sigma$  can be split into two sub-traces  $\sigma = \sigma_1 \sigma_2$ , such that  $F_1$  is true on  $\sigma_1$ , observed from the current position  $i$ , and  $F_2$  is true on  $\sigma_2$  from position 1 (relative to  $\sigma_2$ ). Note that the first formula  $F_1$  is not checked on the second trace  $\sigma_2$ , and, similarly, the second formula  $F_2$  is not checked on the first trace  $\sigma_1$ . Also note that either  $\sigma_1$  or  $\sigma_2$  may be an empty sequence. The sequential temporal composition differs from concatenation in that the last state of the first sequence is also the first state of the second sequence. Thus, the formula  $F_1 ; F_2$  is true if and only if the trace  $\sigma$  can be split into two overlapping sub-traces  $\sigma_1$  and  $\sigma_2$  such that  $\sigma = \sigma_1^{[1..|\sigma_1|-1]} \sigma_2$  and  $\sigma_1(|\sigma_1|) = \sigma_2(1)$  and such that  $F_1$  is true on  $\sigma_1$ , observed from the current position  $i$ , and  $F_2$  is true on  $\sigma_2$  from position 1 (relative to  $\sigma_2$ ). This operator captures the semantics of sequential composition of finite programs.

Finally, applying a rule within the trace, i.e. positions  $1 \dots n$ , consists of replacing the call by the right-hand side of its definition, substituting the actual arguments for formal parameters. At the boundaries (0 and  $n + 1$ ) a rule application evaluates to true if and only if it is maximal.

### 3.3 Examples of EAGLE Formulas

We shall as example set up the framework we need for the case study to be presented in Section 4. Consider that we observe a robotics controller that executes a set of actions. We want to observe that actions start and terminate correctly, and within expected time periods. Actions can terminate either with success or with failure. Events are time stamped with the number of milliseconds since the start of the application. The controller is instrumented to emit events containing a command,

an action name (a string) and a time stamp (integer):

$$\begin{aligned} \langle event \rangle & ::= \langle command \rangle \text{ string int} \\ \langle command \rangle & ::= \text{start} \mid \text{success} \mid \text{fail} \end{aligned}$$

As events are received by the monitor, they are parsed and stored in a state, which the EAGLE formulas can refer to. The state is an object of a user defined Java class, and an example is given below. The class defines the state and a set of methods observing the state, which can be referred to in EAGLE formulas.

```
public class State{
    public int command; // 1=start, 2=success, 3=fail
    public String activity;
    public int time;

    public boolean start(){
        return command == 1;
    }

    public boolean start(String a){
        return start() && activity.equals(a);
    }

    public boolean end(){
        return command == 2 || command == 3;
    }

    public boolean end(String a){
        return end() && activity.equals(a);
    }
}
```

To illustrate EAGLE, the following EAGLE fragment defines three rules, Always, Eventually, and Previously - corresponding to the usual temporal operators for “always in the future”, “eventually in the future” and “previously in the past”:

$$\begin{aligned} \mathbf{max} \text{ Always}(\mathbf{Term } t) &= t \wedge \bigcirc \text{Always}(t) \\ \mathbf{min} \text{ Eventually}(\mathbf{Term } t) &= t \vee \bigcirc \text{Eventually}(t) \\ \mathbf{min} \text{ Previously}(\mathbf{Term } t) &= t \vee \odot \text{Previously}(t) \end{aligned}$$

The following two monitors check that every observed start of the particular action “turn” is matched by a subsequent end of that action, and reversely, that every end

of the action is preceded by a start of that action:

$$\begin{aligned} \mathbf{mon} \ M1 &= \text{Always}(\text{start}(\text{"turn"}) \rightarrow \text{Eventually}(\text{end}(\text{"turn"}))) \\ \mathbf{mon} \ M2 &= \text{Always}(\text{end}(\text{"turn"}) \rightarrow \text{Previously}(\text{start}(\text{"turn"}))) \end{aligned}$$

Consider the more generic property “for all activities, if they start they must end”, and reversely for the past time case. This could be stated as follows:

$$\begin{aligned} \mathbf{min} \ \text{EventuallyEnd}(\mathbf{String} \ a) &= \text{Eventually}(\text{end}(a)) \\ \mathbf{min} \ \text{PreviouslyStart}(\mathbf{String} \ a) &= \text{Previously}(\text{start}(a)) \\ \mathbf{mon} \ M3 &= \text{Always}(\text{start}() \rightarrow \text{EventuallyEnd}(\text{activity})) \\ \mathbf{mon} \ M4 &= \text{Always}(\text{end}() \rightarrow \text{PreviouslyStart}(\text{activity})) \end{aligned}$$

Consider next properties about real-time behavior, such as the property “when a turn starts, it should eventually end within 10-30 seconds”. For this is needed a real-timed version of the Eventually operator. The formula EventuallyWithin( $t, l, r$ ) monitors that  $t$  occurs within the relative time bounds  $l$  (lower bound) and  $r$  (upper bound), measured in seconds, from when it is applied. It is defined with the help of the auxilliary rule EventuallyAbs, which is an absolute timed version:

$$\begin{aligned} \mathbf{min} \ \text{EventuallyAbs}(\mathbf{Term} \ t, \mathbf{int} \ al, \mathbf{int} \ ah) &= \\ & \text{time} \leq ah \wedge \\ & ((t \wedge \text{time} \geq al) \vee \\ & (\neg t \wedge \bigcirc \text{EventuallyAbs}(t, al, ah))) \\ \\ \mathbf{min} \ \text{EventuallyWithin}(\mathbf{Term} \ t, \mathbf{int} \ l, \mathbf{int} \ h) &= \\ & \text{EventuallyAbs}(t, \text{time} + (l * 1000), \text{time} + (h * 1000))) \end{aligned}$$

Note that the time variable is defined in the state and contains the latest timestamp. The property “when a turn starts, it should eventually end within 10-30 seconds” can now be stated as follows:

$$\mathbf{mon} \ M5 = \text{Always}(\text{start}(\text{"turn"}) \rightarrow \text{EventuallyWithin}(\text{end}(\text{"turn"}), 10, 30))$$

### 3.4 Online Evaluation Algorithm and Implementation

A monitoring algorithm for EAGLE determines whether a trace  $\sigma$  models a monitoring specification  $D O$ . Our algorithm operates in an online fashion. That is, it

is presented sequentially each state of  $\sigma$  and does not refer back to past states or forward to future states. This allows the algorithm to be used in online monitoring contexts.

Ideally if a monitoring specification is expressible in a more restricted system, e.g. LTL, then the EAGLE algorithm should perform about as well as an efficient algorithm for the restricted system. We have prove this true for LTL, and will be exploring other logics.

The algorithm employs a function  $eval(D, s)$  that examines a state,  $s$  and transforms a monitor  $D$  into a monitor  $D'$  such that  $s \models D$  iff  $\sigma \models D'$ .

The algorithm is, where possible, a direct implementation of the definition of EAGLE semantics. So for example if  $D$  monitors a formula  $F_1 \vee F_2$ , then (with a slight overloading of the notation)

$$eval(F_1 \vee F_2, s) = eval(F_1, s) \vee eval(F_2, s).$$

Furthermore,

$$eval(\bigcirc F, s) = F.$$

However, an online algorithm that examines a trace in temporal order cannot treat the previous state operator so easily. Thus the algorithm maintains an auxiliary data structure (assumed available to  $eval$ ) used by  $eval$  on sub-formulas headed by the  $\bigcirc$  operator, that records the result of (partially) evaluating the formula in the previous state. This is illustrated as follows:

$$\begin{aligned} \mathbf{min} \text{ Sometime}(\mathbf{Form} F) &= F \vee \bigcirc \text{Sometime}(F) \\ \mathbf{min} \text{ R}(\mathbf{int} k) &= \bigcirc(y + 1 == k) \\ \mathbf{mon} M &= \text{Sometime}R(x) \end{aligned}$$

This monitor will be true if somewhere in the trace there are two successive states such that in the value of  $y$  in the first state is one less than the value of  $x$  in the second state. More generally, notice that the combination of parameterizing rules with data values and use of the next and previous state operators enable constraints that relate the values of state variables occurring different states.

Since  $eval$  recursively decomposes the formulas, eventually  $eval$  will be called on  $\bigcirc y + 1 == k$ . Note the state variable  $y$  refers to the value of  $y$  in the previous state, while the formal parameter  $k$  is bound to the value of  $x$  in the current state. Since the previous state is unavailable, in the prior step the algorithm must take some action to record relevant information. Our algorithm pre-evaluates and caches the evaluation any formula  $P$  headed by a previous state operator, in this case the formula  $y + 1 == k$ . However since the value of  $k$  will not be known at that point, the evaluation is partial. In particular note that the atomic formulas and the underlying

expression language (in our case this is Java expressions), must be partially evaluated<sup>1</sup>. Also note that since the formula  $P$  can be arbitrarily complex, in particular another previous state operator may be nested within, the pre-evaluation is done by a recursive call to *eval*.

This is basic idea of the algorithm. One subtle point is that the sub-formulas that must be pre-evaluated must be identified and properly initialized prior to processing the first state. This is done by expanding monitor formulas by unfolding rule definitions, while avoiding infinite expansion due to recursive rule definitions.

*eval* yields a formula that may be simplified without altering the correctness of the algorithm. Indeed the key to efficient monitoring and provable space bounds is adequate simplification. In our implementation, formulas are represented in disjunctive normal form where each literal is an instance of negation, the previous or next operator or a rule application. Subsumption, i.e. simplifying  $(a \wedge b) \vee a$  to  $a$  is essential.

### 3.5 Complexity of EAGLE

It is evident from the semantics given above in Section 3.1 that, in theory, EAGLE is a highly expressive and powerful language; indeed, given the unrestricted nature of the data types and expression language, it is straightforward to see it is Turing-complete. This is not of much interest practically. Furthermore, assuming that all data is finitely bounded with, of course, finite input traces is also of not much interest. What is of more interest is the cost of evaluation, using our implementation algorithms, of an Eagle temporal constraint under, say, zero cost evaluation of any data computation that may ensue; this then informs us of the complexity of the temporal aspects of the evaluation of the logic. An alternative way of viewing this is to show that our algorithm can meet known optimal bounds for various sub-logics embedded within Eagle. To that end, there are some initial complexity results that are of interest.

Our first result relates to an embedding of propositional linear time temporal logic (LTL), over both future and past. In [?], we show that the step evaluation for an initial LTL formula of size  $m$  has an upper time complexity bound of  $O(m^4 2^{2m} \log^2 m)$  and a space bound of  $O(m^2 2^m \log m)$ , thus showing that the evaluation at any point is not dependent on the length of the history, i.e. the input seen so far. The result is close to the lower bound of  $O(2^{\sqrt{m}})$  for monitoring LTL given in [?].

**Kouhsik and Grigore - are there any results that can be put down about the algorithm when restricted to metric temporal logic - as in Grigore's paper**

---

<sup>1</sup> A simpler alternative to partial evaluation is to form a closure and do the complete evaluation when all variables are bound.

**for RV'04? I haven't had time to investigate.**

### 3.6 *Experimental Timings*

### 3.7 *Event Extraction*

The event extraction can be achieved in a number of ways, including *wrapping* and *instrumentation*. In a *wrapping* approach, the standard execution environment is replaced with a customized one that allows observation by wrapping system libraries. This is the approach of Purify [20]. In the *instrumentation* approach, source code (or object code) is augmented with code that generates the event stream. Our experiments use both approaches. The instrumentation approach is used to generate events for the temporal logic monitoring. As will be explained, this monitoring examines events indicating the start and end of task executions, and the code has been manually instrumented to generate these events. The wrapping approach is used to generate events for the concurrency analysis, where lock acquisitions and lock releases are monitored. These events are generated by wrapping method calls around POSIX [17] thread methods, and then instrument the wrappers.

We are working on technology for automatic program instrumentation. Here code is instrumented based on an *instrument specification* that consists of a collection of predicate/action rules. The predicate is a predicate on program statements. These predicates are conjunctions of atomic predicates that include method invocations, references to global variables, object variables, and local variables, and lock usage. The actions are specifications describing the inserted instrumentation code. They include reporting the program location, a time stamp, the value of an expression, and invocation of auxiliary methods. Values of primitive types are recorded in the event itself, but if the value is an object, a unique integer descriptor of the object is recorded.

Such an instrumentation package, named jSpy, has been implemented for instrumenting Java bytecode [9]. This was first implemented using Jtrek [5], a Java API that provides lower-level instrumentation functionality. Use of bytecode instrumentation, and of Jtrek in particular, has worked out well. However, at the time of writing, a new version is being implemented based on the BCEL library [6], which is Open Source, unlike Jtrek, which has been discontinued.

### 3.8 *Observer Framework*

As described above, runtime analysis is divided into two parts: instrumentation and execution of the instrumented program. To minimize the impact on the program un-

der test, events should contain minimal information. Two categories of information need to be transmitted: *essential* information, needed to detect property violations, and *contextual* information, needed to print out informative error messages when properties get violated. Instrumentation is done such that contextual information is sent only when it changes, and not with every event. The event observer, (see Figure 6), can be correspondingly be split into two stages. The *event dispatcher* reads the events and sends a reconstructed version to one or more *property analyzers*.

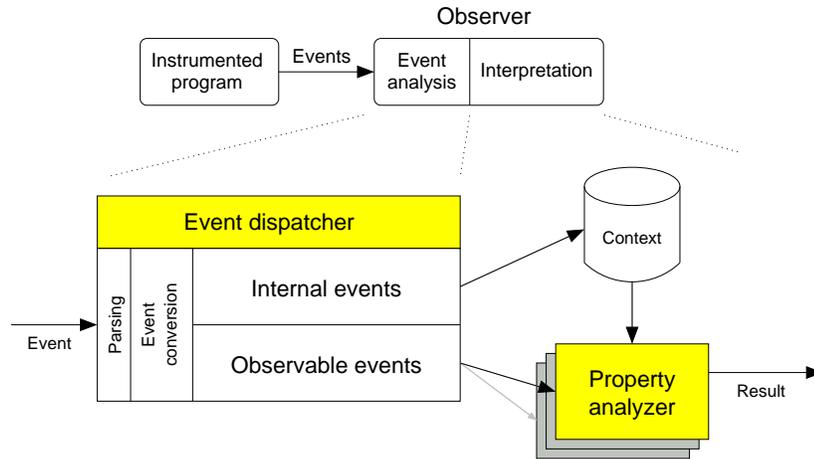


Figure 6. The observer architecture.

The event dispatcher parses events, and converts them into a unified format, accommodating different instrumentation packages. The contextual information, transmitted in *internal events* (not emitted to the property analyzers), include thread names, code locations, and reentrant acquisitions of locks (lock counts). The event dispatcher package maintains a database with the full context of the events. This allows for writing simpler property analyzers. The property analyzers subscribe to particular event types made accessible through an observer interface [8] and are completely decoupled from each other.

It is up to each property analyzer to record all relevant information for keeping a history of the events, since the context maintained by the event dispatcher changes dynamically with event evaluation. The property analyzer reports violations of properties in its model using the stored data and context information. The main advantages of this approach are the decoupling of the instrumentation package from observation, and the ability to re-use one event stream for several property analyzers.

### 3.9 The JPaX Concurrency Analyzer

Multi-threaded programs are particularly difficult to test due to the fact that they are non-deterministic. A multi-threaded program consists of several threads that execute in parallel. A main issue for a programmer of a multi-threaded application

is to avoid data races where several threads access a shared object simultaneously. Multi-threading programming languages therefore provide constructs for ensuring mutual exclusion, usually in the form of locks. If other threads utilize the same lock when accessing an object, mutual exclusion is guaranteed. If threads do not acquire the same lock (or do not acquire locks at all) when accessing an object then there is a risk of a data race. It algorithm [21] can detect such disagreements by analyzing single execution traces. The Eraser algorithm has been implemented in the JPaX tool. Recent work has shown that another kind of error, high-level data races, can still be present in programs that use mutual exclusion for accessing individual fields, but not sets of fields, correctly [1].

Deadlocks can occur when two or more threads acquire locks in a cyclic manner. As an example of such a situation consider two threads  $T_1$  and  $T_2$  both acquiring locks  $A$  and  $B$ . Thread  $T_1$  acquires first  $A$  and then  $B$  before releasing  $A$ . Thread  $T_2$  acquires  $B$  and then  $A$  before releasing  $B$ . This situation poses a deadlock situation since thread  $T_1$  can acquire  $A$  where after thread  $T_2$  acquires  $B$ , where after both threads cannot progress further. JPaX includes such a deadlock detection algorithm. An extension to this algorithm reduces the number of false positives [2].

## 4 Case Study: A Planetary Rover Controller

The case study described here is the planetary rover controller K9, and in particular its *executive* subsystem, developed at NASA Ames Research Center – a full account of this case study is described in [4]. The executive receives plans of actions that the rover is requested to carry out, and executes these plans. First a description of the system is presented, including a description of what plans (the input domain) look like. Then it is outlined how plans (test inputs) can be automatically generated using model checking. Finally it is described how, for each plan, one can automatically generate a set of temporal logic properties that the executive must satisfy when executing the plan.

### 4.1 System Description

The executive is a multi-threaded system (35,000 lines of C++ code) that receives flexible plans from a planner, which it executes according to a plan language semantics. A plan is a hierarchical structure of actions that the rover must perform. Traditionally, plans are deterministic sequences of actions. However, increased rover autonomy requires added flexibility. The plan language therefore allows for branching based on conditions that need to be checked, and also for flexibility with respect to the starting time and ending time of an action. This section gives a short presentation of the (simplified) language used in the description of the plans that the rover

<i>Plan</i>	→	<i>Node</i>		(block
<i>Node</i>	→	<i>Block</i>   <i>Task</i>		:id plan
<i>Block</i>	→	(block		:continue-on-failure
		<i>NodeAttr</i>		:node-list (
		:node-list ( <i>NodeList</i> )		(task
<i>NodeList</i>	→	<i>Node</i> <i>NodeList</i>   ε		:id drive1
<i>Task</i>	→	(task		:start-condition (time +1 +5)
		<i>NodeAttr</i>		:end-condition (time +1 +30)
		:action <i>Symbol</i>		:action BaseMove1
		[:fail]		:duration 20
		[:duration <i>DurationTime</i> ]		)
<i>NodeAttr</i>	→	:id <i>Symbol</i>		(task
		[:start-condition <i>Condition</i> ]		:id drive2
		[:end-condition <i>Condition</i> ]		:end-condition (time +10 +16)
		[:continue-on-failure]		:action BaseMove2
<i>Condition</i>	→	(time <i>StartTime</i> <i>EndTime</i> )		:fail
				) ) )

Figure 7. Plan grammar (left) and an example of a plan (right).

executive must execute.

#### 4.1.1 Plan Syntax

A plan is a *node*; a node is either a *task*, corresponding to an *action* to be executed, or a *block*, corresponding to a logical group of nodes. Figure 7 (left) shows the grammar for the language. All node attributes, with the exception of the node's *id*, are optional. Each node may specify a set of *conditions*, e.g. the *start condition* (that must be true at the beginning of the node execution) and the *end condition* (that must be true at the end of the node execution). Each condition includes information about a relative or absolute time window, indicating a lower and an upper bound on the time. The *continue-on-failure* flag indicates what the behavior will be when node failure is encountered.

The attributes *fail* and *duration* were added to facilitate testing of the executive. During testing using test case generation, the real actions are never executed since this would require operating the rover mechanics. Attributes *:fail* and *:duration* replace the actions during testing. Flag *fail* specifies the action status of a task after execution; *duration* specifies the duration of the action. Figure 7 (right) shows a plan that has one block with two tasks (*drive1* and *drive2*). The time windows here are relative (indicated by the '+' signs in the conditions).

#### 4.1.2 Plan Semantics

For every node, execution proceeds through the following steps: (1) Wait until the start condition is satisfied; if the current time passes the end of the start condition,

```

class UniversalPlanner { ...
  static int nNodes = 0;
  static int tRange = 0;

  static void Plan(int nn, int tr) {
    nNodes = nn; tRange = tr;
    UniversalAttributes();
    Node plan = UniversalNode();
    print(plan);
    assert(false);
  }

  static Node UniversalNode() {
    if (nNodes == 0) return null;
    if (chooseBool()) return null;
    if (chooseBool())
      return UniversalTask();
    return UniversalBlock();
  }

  static Node UniversalTask() {
    Symbol action = new Symbol();
    boolean fail = chooseBool();
    int duration = choose(tRange);
    Task t =
      new Task(id, action, start,
              end, continueOnFailure,
              fail, duration);

    nNodes--;
    return t;
  }
}

static Node UniversalBlock() {
  nNodes--;
  ListOfNodes l = new ListOfNodes();
  for (Node n = UniversalNode();
       n != null; n = UniversalNode())
    l.pushEnd(n);
  Block b =
    new Block(id, l, start, end,
              continueOnFailure);
  return b;
}

static Symbol id;
static TimeCondition start, end;
static boolean continueOnFailure;

static UniversalAttributes() {
  id = new Symbol();
  Symbolic sTime1 = new SymInt();
  Symbolic sTime2 = new SymInt();
  Symbolic._Path_cond._add_GT(sTime2, sTime1);
  start =
    new TimeCondition(sTime1.solution(),
                     sTime2.solution());
  Symbolic eTime1 = new SymInt();
  Symbolic eTime2 = new SymInt();
  Symbolic._Path_cond._add_GT(eTime2, eTime1);
  end = new TimeCondition(eTime1.solution(),
                        eTime2.solution());
  continueOnFailure = chooseBool();
}
}

```

Figure 8. Code that generates input plans for system under test.

the node times out and this is a node failure. (2) The execution of a *task* proceeds by invoking the corresponding action. The action takes exactly the time specified in the `:duration` attribute. Note that this attribute during testing replaces the actual execution of the action on the rover. The action's status must be fail, if `:fail` is true or the time conditions are not met; otherwise, the status must be success. If the status of the action indicates failure, its task fails. The execution of a *block* simply proceeds by executing each of the nodes in the node-list in order. (3) If time exceeds the end condition, the node fails. On a *node failure*, when execution returns to the sequence, the value of the *continue-on-failure* flag of the failed node is checked. If true, execution proceeds to the next node in the sequence. Otherwise the node failure is propagated to any enclosing nodes. If the node failure passes out to the top level of the plan, the remainder of the plan is aborted.

## 4.2 Test Input Generation

Figure 8 shows part of the Java code, referred to as the *universal planner*, that is used to generate plans (i.e., test inputs for the executive). The framework suggested in Section 2 is used where an annotated Java program specifies only the structure of the inputs together with the preconditions on this structure. Model checking with symbolic execution generates the inputs. In order to specify the structure non-deterministic choice (choose methods) are exploited over all structures allowed in

the grammar presented in Figure 7, and preconditions are specified as constraints over some of the integer variables in the structure. For the latter, only time points are considered. Furthermore, these represent *inputs* to our specification, to allow symbolic execution and constraint solving to generate valid test cases. For brevity, only a small sample set of constraints is shown here (stating that the end time is larger than the start time of an interval).

To illustrate the flexibility in our approach, some of the variables are considered concrete inputs, e.g. the number of nodes in the total structure (`nNodes`), and yet others, e.g. the duration of a task (`duration`), is concretized by non-deterministic choice. The assertion in the program specifies that it is not possible to create a “valid” plan (i.e., executions that reach this assertion generate valid plans). The JPF model checker model checks the universal planner and is thus used to explore the (infinite) state space of the generated input plans. Different search strategies find multiple counterexamples; for each counterexample JPF is run in simulation mode to print the generated plan to a file, which then serves as input to the rover.

```

class Symbolic { ...
  static PathCondition _Path_cond;
  Symbolic_plus(Symbolic e) { ... }
  Symbolic_minus(Symbolic e) { ... }
  int solution() { ... }
}

class PathCondition { ...
  Constraints c;
  void _add_GT(Symbolic e1,
              Symbolic e2){
    c.add_constraint_GT(e1,e2);
    if (!c.is_satisfiable())
      backtrack();
  }
  return;
}

```

Figure 9. Library classes for symbolic execution.

Figure 9 gives part of the library classes that provide symbolic execution. Class `Symbolic` implements all symbolic constraints and includes a subclass `SymInt` that represents symbolic integer values. The static field `Symbolic._Path_cond` stores the (numeric) path condition. Method `_add_GT` updates the path condition with the *greater-than* constraint. Method `is_satisfiable` uses the Omega library to check if the path condition is infeasible (in which case, JPF will backtrack). The `solution` method first solves the constraints and then returns one solution value for a symbolic integer (solutions are currently not defined for non-integer symbolic values).

### 4.3 System Analysis

The semantics of a particular plan can very naturally be formulated in temporal logic. In writing such properties, the following predicates were used: `start(id)` (true immediately after the start of the execution of the node with the corresponding `id`), `success(id)` (true when the execution of the node ends successfully), `fail(id)` (true when the execution of the node ends with a failure), and `end(id)`, which denotes `success(id) ∨ fail(id)`. The code was instrumented to monitor these predicates. Furthermore, for each plan a collection of temporal properties over these predicates

- $\diamond \text{start}(\text{plan})$ , i.e., the initial node `plan` should eventually start.
- $\square(\text{start}(\text{plan}) \rightarrow \diamond_{1,5} \text{start}(\text{drive1}))$ , i.e., if the `plan` starts, then task `drive1` should begin execution within 1 and 5 time units.
- $\square(\text{start}(\text{drive1}) \rightarrow (\diamond_{1,30} \text{success}(\text{drive1}) \vee \diamond \text{fail}(\text{drive1})))$ , i.e., if task `drive1` starts, then it should end successfully within 1 and 30 time units or it should eventually terminate with a failure.
- $\square(\text{success}(\text{drive1}) \rightarrow \diamond \text{start}(\text{drive2}))$ , i.e., if task `drive1` ends successfully, then task `drive2` should eventually begin execution.
- $\square(\text{end}(\text{drive2}) \rightarrow \diamond \text{success}(\text{plan}))$ , i.e., termination of task `drive2` implies successful termination of the whole plan (due to `continue-on-failure` flag).
- $\diamond \text{success}(\text{drive1})$ , i.e., task `drive1` should end successfully (since `:duration` is within time window).
- $\diamond \text{fail}(\text{drive2})$ , i.e., task `drive2` should fail (due to `:fail`).

Figure 10. Temporal logic properties representing partial semantics of plan in Fig. 7.

was generated automatically. Their validity was verified on execution traces. As an example, the properties for the plan shown in Figure 7 (right) are shown in Figure 10. This set of properties does not fully represent the semantics of the plan, but the approach appears to be sufficient to catch errors.

The runtime analysis identified a number of errors in the executive. A preliminary, partially automated system for runtime testing found a deadlock and a data race. For the deadlock, the additional instrumentation triggered the deadlock during execution, but in fact the pattern existed in the un-instrumented version of the executive, and would have been identified by the instrumentation, even if it had not occurred explicitly. The data race, involving access to a shared variable used to communicate between threads, was suspected by the developers, but had not been confirmed in code. The trace analysis allowed the developers to see the read/write pattern clearly and redesign the communication. The fully automated testing system detected a bug that had been seeded in the code for verification purposes: the bug produced an execution failure when a plan node was processed after the beginning of its start window. Finally, the automated testing system found a missing feature that had been overlooked by the developers: the lower bounds on execution duration were not enforced, so the temporal logic model predicted failure when execution in fact succeeded. This latter error was unknown to the developers, and it showed up later during actual rover operation before it was corrected.

## 5 Conclusions and Future Work

A framework for testing based on automated test case generation and runtime analysis has been presented. This paper proposed and demonstrated the use of model checking and symbolic execution for test case generation, and the use of rewriting-based temporal logic monitoring during the execution of the test cases. The framework requires construction of a test input generator and a property generator for the application. From that, an arbitrarily large test suite can be automatically generated, executed and verified to be in conformity with the properties. For each input sequence (generated by the test input generator) the property generator constructs a set of properties that must hold when the program under test is executed on that input. The program is instrumented to emit an execution log of events. An observer checks that the event log satisfies the set of properties.

We take the position that writing test oracles as temporal logic formulas is both natural and leverages algorithms that efficiently check if execution on a test input conforms to the properties. While property definition is often difficult, at least for some domains, an effective approach is to write a property generator, rather than a universal set of properties that are independent of the test input. Note also that the properties need not completely characterize correct execution. Instead, a user can choose among a spectrum of weak but easily generated properties to strong properties that may require construction of complex formulas.

In the near future, we will continue the development of a complete testing environment for the K9 rover executive, and seek to get this technology transferred to NASA engineers. We will be exploring how to improve the quality of the generated test suite by altering the search strategy of the model checker, and by improving the symbolic execution technology. We will also investigate the use of real-time logic and other more complicated logics. In particular, the Eagle logic should provide a good framework for monitoring. We are continuing the work on instrumentation of Java bytecode and will extend this work to C and C++. Our research group has done fundamental research in other areas, such as software model checking (model checking the application itself, and not just the input domain), and static analysis. In general, our ultimate goal is to combine the different technologies into a single coherent framework.

## References

- [1] C. Artho, K. Havelund, and A. Biere. High-level Data Races. In *VVEIS '03*, April 2003.
- [2] S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Internal report, to be published, October 2002.

- [3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.
- [4] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. A Comparative Field Study of Advanced Verification Technologies. Internal report, in preparation for submission, November 2002.
- [5] S. Cohen. Jtrek. Compaq, <http://www.compaq.com/java/download/jtrek>.
- [6] Markus Dahm. BCEL. <http://jakarta.apache.org/bcel/>.
- [7] 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.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [9] A. Goldberg and K. Havelund. Instrumentation of Java Bytecode for Runtime Analysis. In *Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports from ETH Zurich*, Switzerland, 2003. ETH Zurich.
- [10] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.
- [11] A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, July 2002.
- [12] A. Hartman. Model Based Test Generation Tools. [http://www.agedis.de/documents/ModelBasedTestGenerationTools\\_cs.pdf](http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf).
- [13] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2002.
- [14] S. Khurshid, C. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of TACAS'03: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, Warsaw, Poland, April 2003.
- [15] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [16] B. Korel. Automated Software Test Data Generation. *IEEE Transaction on Software Engineering*, 16(8):870–879, August 1990.
- [17] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly, 1998.
- [18] Parasoft. <http://www.parasoft.com>.
- [19] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 31(8), August 1992.

- [20] *Purify: Fast Detection of Memory Leaks and Access Errors*. January 1992.
- [21] 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.
- [22] N. Tracey, J. Clark, and K. Mander. The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.
- [23] T-VEC. <http://www.t-vec.com>.
- [24] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'02: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.