Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 336 (2005) 209–234

Theoretical Computer Science

www.elsevier.com/locate/tcs

Combining test case generation and runtime verification

Cyrille Artho[a,1], Howard Barringer[b,2], Allen Goldberg[c], Klaus Havelund[c,*], Sarfraz Khurshid[d,3], Mike Lowry[e], Corina Pasareanu[f], Grigore Roşu[g], Koushik Sen[g,4], Willem Visser[h], Rich Washington[h]

[a]*Computer Systems Institute, ETH Zurich, Switzerland*
[b]*School of Computer Science, University of Manchester, UK*
[c]*Kestrel Technology, USA*
[d]*UT ARISE, University of Texas at Austin, USA*
[e]*NASA Ames Research Center, USA*
[f]*Kestrel Technology, NASA Ames Research Center, USA*
[g]*Department of Computer Science, Univ. of Illinois at Urbana-Champaign, USA*
[h]*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 input domain of the program with runtime verification, where execution traces are monitored and verified against properties expressed in temporal logic. Capabilities also exist for analyzing traces for concurrency errors, such as deadlocks and data races. The input domain of the program is explored using a model checker extended with symbolic execution. Properties are

*Corresponding author.
E-mail address: havelund@kestreltechnology.com (K. Havelund).
[1]Cyrille Artho is grateful to QSS for the partial support provided to conduct this research.
[2]Howard Barringer is grateful to RIACS/USRA and the UK's EPSRC under grant GR/S40435/01for the partial support provided to conduct this research.
[3]Sarfraz Khurshid is grateful to RIACS/USRA for the partial support provided to conduct this research.
[4]Koushik Sen is grateful to RIACS/USRA for the partial support provided to conduct this research.

0304-3975/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.tcs.2004.11.007

formulated in an expressive temporal logic. A methodology is advocated that automatically generates properties specific to each input rather than formulating properties uniformly true for all inputs. The paper describes an application of the technology to a NASA rover controller.

Keywords: Automated testing; Test case generation; Model checking; Symbolic execution; Runtime verification; Temporal logic; Concurrency analysis; NASA 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 single *test input* to the program, together with a description of the *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. However, we believe that a non-trivial part of the testing work *can* be automated. Evidence is found in a previous case study, where an 8000-line Java application was tested by different student groups using different testing techniques [12]. It was observed that the vast majority of faults that were found in this system could have been found in a fully automatic way. We suggest a framework for generating and executing test cases in an automated way as illustrated by Fig. 1. For a particular application to be tested, one establishes a test harness consisting of two modules: a *test case generator* and an *observer*.

The test case generator takes as input a model of the input domain of the application to be tested. The model furthermore describes a mapping from input values to properties: for each input element, the model defines what properties an execution on that input should satisfy. The test case generator automatically generates inputs to the application. For each generated input a set of properties is generated. The input is fed to the program, which executes, generating an execution trace. The observer module checks the trace against the generated set of properties. Hence, it takes the execution trace and the set of generated properties 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. 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 verification* is used to refer to instrumentation as well as observation.

Test cases are generated using the JAVA PATHFINDER model checker extended with techniques for symbolic execution and the properties generated are expressed in the EAGLE temporal logic, capable of embedding most temporal logics. The framework described is being applied to a case study, a multi-threaded NASA rover controller written in C++ (35,000 lines of code), which interprets and executes complicated activity plans. The individual techniques, model checking with symbolic execution and runtime verification in EAGLE, have been described elsewhere, respectively in [37,8]. The contribution of this paper is to demonstrate their combination on a realistic case study. A special characteristic is that the properties to be verified are generated automatically from the inputs to the program to be tested.

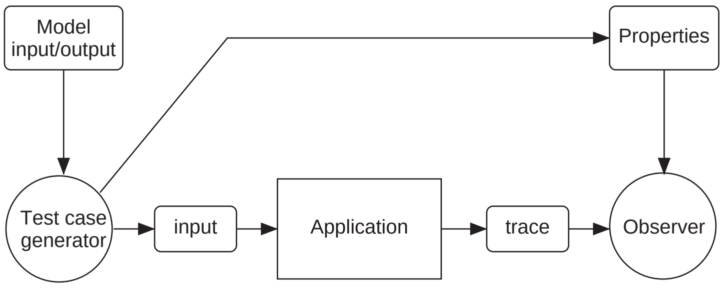

Fig. 1. Test case generation and runtime verification.

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 verification techniques: temporal logic monitoring and concurrency analysis. Section 4 describes the case study, where these technologies are applied to a planetary rover controller. Section 5 outlines some related work. Section 6 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 case 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 lends itself to automation and therefore has been the focus of much research attention—recently it has also been adopted in industry [50,66,16,26]. 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 [40,65]. 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, e.g. the TGV tool [64] for the generation of conformance test suites for protocols, and the AGEDIS tool [1] for automated generation and execution of test suites for distributed component-based software; see also Hartman's survey of the field [30]. 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 [30]—even the code itself can be used to create a model, e.g. approaches based on symbolic execution [39,50]. 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. 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. This is the approach pursued in the following.

In [37] a framework is presented that combines *symbolic execution* and model checking techniques for the verification of Java programs. The framework can be used for test input generation for *white-box* and *black-box* testing. For white-box test input generation, the framework model checks the program under test. A testing coverage criterion, e.g. branch coverage, is encoded in a temporal logic specification. Counter-examples to the specification represent paths that satisfy the coverage criterion. Symbolic execution, which is performed during model checking, computes a representation, i.e., a set of constraints, of all the inputs that execute those paths. The actual testing requires solving the input constraints in order to instantiate test inputs that can be executed. The framework can also be used for black-box test input generation. In this case, the inputs to the program under test are described by a Java input specification, i.e., a Java program, annotated with special instructions to model non-determinism and to encode constraints, for symbolic execution. The framework is then used to check this Java specification, i.e., to systematically explore the input domain of the program under test and to generate inputs according to this specification. It is in this latter context (black-box) that we use the framework from [37] in this paper. Note that for black-box test input generation, only the input specification is required to be expressed in Java; the program under test can be written in another language, e.g. C++ as it is the case for this paper. Note that in writing input specifications, we can take full advantage of the expressive power of the Java language and thus we can easily express inputs with complex structure, e.g. linked lists, red-black search trees, executive plans.

Using symbolic execution for test input generation is a well-known approach, but typically only handles sequential code with simple data. In [37], this technique has been extended to handle complex data structures, e.g. lists and trees, concurrency as well as linear constraints on integer data. Symbolic execution of a program path results in a set of constraints that define program inputs that execute the path; these constraints are then solved using off-the-shelf decision procedures to generate concrete test inputs. When the program represents an executable input 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.

2.1.2. Symbolic execution

The enabling technology for black-box test input generation from an input specification is the use of symbolic execution. In fact, the same techniques can be applied for white box testing. The main idea behind symbolic execution [39] is to use symbolic values, instead 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

```
        int     x, y;
        read    x,y;
1:      if      (x > y)     {
2:              x = x + y;
3:              y = x - y;
4:              x = x - y;
5:              if    (x > y)
6:                  assert(false);
        }
```

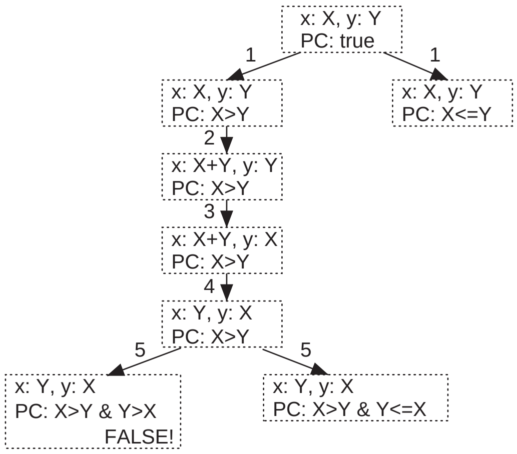

Fig. 2. Code for swapping integers and corresponding symbolic execution tree.

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 [37], the code fragment in Fig. 2, which swaps the values of integer variables `x` and `y`, when `x` is greater than `y`. Fig. 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, it 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 the 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 [37] 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 symbolic execution of the procedure. This allows symbolic execution of procedures without requiring an a priori bound on the number of input objects. Procedure preconditions are used to initialize inputs only with valid values.

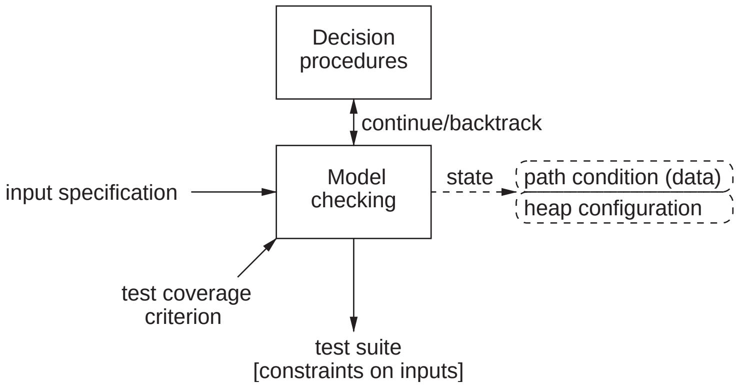

Fig. 3. Framework for test input generation.

2.1.3. Framework for test input generation

Our symbolic execution-based framework is built on top of the JAVA PATHFINDER (JPF) model checker [67]. 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 treats non-deterministic choice expressed in annotations of the program being analyzed. JPF has been extended with a symbolic execution capability which is described in detail in [37].

Fig. 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 enables JPF to perform symbolic execution. Essentially, the model checker explores the symbolic state space of the program, for example, the symbolic execution tree in Fig. 2. A symbolic state includes information about the heap configuration and the path condition on integer variables. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure; currently our system uses the Omega library [52] 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 trace, that represents a path that satisfies the coverage criterion. The model checker also outputs the input constraints for this path. 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, using an off-the-shelf constraint solver. 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. Note that symbolic execution performed on programs with loops can explore infinite execution trees, hence symbolic execution might not terminate. Therefore, for systematic state space exploration, limited depth-first search or breadth-first search is used; our framework also supports various heuristic search strategies, for example, based on branch coverage [27] or random search.

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 an input to the program under test and generates a set of properties representing the test oracle. The properties are assertions in temporal logic, which are then checked against the program execution using the runtime verification tools described in Section 3.

This approach leverages the runtime verification technology to great effect, just as test case generation leverages model checking and symbolic execution. 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 grammar of the plan language. 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

Runtime verification is divided into two parts: *instrumentation* and *event observation*. A monitor receives events from the executing program, emitted by event generators inserted during instrumentation, and dispatches them to a collection of algorithms, each of which performs a specialized trace analysis. We consider two kinds of such algorithms: the EAGLE temporal logic monitor and three concurrency analyzers, that can detect deadlock potentials, as well as two kinds of data race potentials. The concurrency analyzers are currently not yet fully integrated in the presented testing environment, but are mentioned since they form an interesting addition to temporal logic monitoring, experiments have been made and the intention is to integrate them.

Instrumentation can be achieved by code instrumentation or code wrapping. In the code instrumentation approach, code that generates the event stream is manually or automatically inserted into source or object code. In the wrapping approach, calls to system library functions within user-defined methods are replaced with calls to wrapper functions that generate the event stream and make the system calls. As an example, Purify [53] uses code instrumentation to check against illegal reads (whether e.g. `*p` accesses a valid address), and uses wrapping to trace memory allocations and deallocations.

Our experiments have used manual source code instrumentation as well as manual wrapping. The source code instrumentation approach is used to generate events for the temporal logic monitoring. The wrapping approach is used to generate events for the deadlock concurrency analysis, where POSIX thread [48] *lock* and *unlock* methods are wrapped and instrumented. In other work, we describe an instrumentation package, named JSpy [24], that automatically instruments Java bytecode. However, this could not be applied here as the code to be tested is written in C++. An automated instrumentation is necessary in order to perform data race analysis since all accesses to shared variables need to be monitored.

3.1. *Temporal logic monitoring with* EAGLE

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, state chart, single-assignment and dataflow languages, through pattern-matching languages based on regular (and extended regular) expressions, to a whole host of modal and, in particular, linear-time temporal logics (LTL). In Section 5, such languages and logics that have been applied directly to runtime verification are discussed more fully. Since for runtime verification one is interested in analyzing traces, the framework of LTL [51] appeared most appropriate for our own work, but none of the proposed temporal logics for runtime verification, of which we were aware, provided the right combination of expressivity, naturalness, flexibility, effectiveness 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-time LTL, extended regular expressions, metric temporal logic (MTL) 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, parameterizable 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 simple, is expressively rich and enables users to define their own set of more complex temporal predicates tuned to the particular needs of the run-time verification application. Indeed, in [8] it is shown how a range of finite-trace monitoring logics, including future-time and past-time temporal logic, extended regular expressions, real-time and MTL, 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".

3.2. *Syntax of* EAGLE

The syntax of EAGLE is shown in Fig. 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,

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

Fig. 4. Syntax of EAGLE.

rules may be recursively defined, whereas 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.

3.3. *Semantics of* EAGLE

The models of our logic are execution traces. An execution trace σ is a finite sequence of observed program states $\sigma = s_1 s_2 \cdots s_n$, where $|\sigma| = n$ is the length of the trace. Note that the ith state s_i of a trace σ is denoted by $\sigma(i)$ and the term $\sigma^{[i,j]}$ denotes the sub-trace of σ 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 execution traces and specifications. That is, given a trace σ and a specification $D\ O$, satisfaction is defined as follows:

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

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 $\models_D\ \subseteq\ (\textit{Trace} \times \textbf{nat}) \times \textbf{Form}$, for a set of rule definitions D, is defined inductively over the structure of the formula and is presented in Fig. 5. First of all, note that the satisfaction relation $\models_D$ is actually defined for the index range $0 \leqslant i \leqslant |\sigma| + 1$ and

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

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

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. The observer only knows the end when it has been passed and no more observation states are forthcoming. It is at that point that a final value for the formula needs 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, referring to the first three rules defined below in Section 3.4 formula `Always`(F) will evaluate to true on an empty trace — since `Always` is defined maximal, whereas formulas `Eventually`(F) and `Previously`(F) will evaluate to false on an empty trace — as they are defined as minimal.

The propositional connectives are given their usual interpretation. The next-time 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 is often featured 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 σ can be split into two sub-traces $\sigma = \sigma_1\sigma_2$, such that F_1 is true on σ_1, observed from the current position i and F_2 is true on σ_2 from position 1 (relative to σ_2). Note that the first formula F_1 is not checked on the second trace σ_2 and, similarly, the second formula F_2 is not checked on the first trace σ_1. Also note that either σ_1 or σ_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, formula F_1; F_2

```
class State extends EagleState {
  public int kind;
    // 1=start, 2=end, 3=fail
  public String action;
  public int time;

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

  public boolean end(){
    return kind == 2;
  }

  public boolean fail(){
    return kind == 3;
  }

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

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

  public boolean fail(String a){
    return fail() && action.equals(a);
  }
}
```

Fig. 6. The state in which EAGLE Java expressions are evaluated.

is true if and only if trace σ can be split into two overlapping sub-traces σ_1 and σ_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 σ_1, observed from the current position i, and F_2 is true on σ_2 from position 1 (relative to σ_2). This operator captures the semantics of sequential composition of finite programs.

Finally, applying a rule within the trace, i.e., positions $1 \ldots n$, 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.

3.4. Programming in EAGLE

To illustrate EAGLE we describe the framework for the case study to be presented in Section 4. Consider a controller for an autonomous mobile robot, referred to as a *rover*, that executes actions according to a given plan. The goal is to observe that actions start and terminate in an expected order and within expected time periods. Actions can either end successfully, or they can fail. The rover controller is instrumented to emit events containing an event kind (start, end, or fail), an action name (a string) and a time stamp (an integer)—the number of milliseconds since the start of the application.

$$\begin{array}{lcl} \langle\mathit{event}\rangle & ::= & \langle\mathit{kind}\rangle\ \ \langle\mathit{string}\rangle\ \ \langle\mathit{int}\rangle \\ \langle\mathit{kind}\rangle & ::= & \textbf{start} \mid \textbf{end} \mid \textbf{fail} \end{array}$$

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 in Fig. 6. The class defines the state and a set of methods observing the state, which can be referred to in EAGLE formulas. To illustrate the use of formulas as parameters to rules, 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”.

$$\begin{aligned}&\textbf{max}\ \texttt{Always}(\textbf{Form}\ f) = f \wedge \bigcirc\texttt{Always}(f)\\&\textbf{min}\ \texttt{Eventually}(\textbf{Form}\ f) = f \vee \bigcirc\texttt{Eventually}(f)\\&\textbf{min}\ \texttt{Previously}(\textbf{Form}\ f) = f \vee \odot\,\texttt{Previously}(f)\end{aligned}$$

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

$$\begin{aligned}&\textbf{mon}\ \texttt{M1} = \texttt{Always}(\mathit{start}(\text{“}\mathit{turn}\text{”}) \rightarrow \texttt{Eventually}(\texttt{end}(\text{“}\mathit{turn}\text{”})))\\&\textbf{mon}\ \texttt{M2} = \texttt{Always}(\mathit{end}(\text{“}\mathit{turn}\text{”}) \rightarrow \texttt{Previously}(\texttt{start}(\text{“}\mathit{turn}\text{”})))\end{aligned}$$

To illustrate data parameterization, consider the more generic property: “*for any action, if it starts it must eventually end*” and conversely for the past-time case. This is stated as follows.

$$\begin{aligned}&\textbf{min}\ \texttt{EventuallyEnd}(\textbf{String}\ a) = \texttt{Eventually}(\texttt{end}(a))\\&\textbf{min}\ \texttt{PreviouslyStart}(\textbf{String}\ a) = \texttt{Previously}(\texttt{start}(a))\\&\textbf{mon}\ \texttt{M3} = \texttt{Always}(\mathit{start}() \rightarrow \texttt{EventuallyEnd}(\mathit{action}))\\&\textbf{mon}\ \texttt{M4} = \texttt{Always}(\mathit{end}() \rightarrow \texttt{PreviouslyStart}(\mathit{action}))\end{aligned}$$

Consider the following properties about real-time behavior, such as the property “*when the rover starts a turn, the turn should end within* 10–30 s”. For this, a real-timed version of the `Eventually` operator is needed. The formula $\texttt{EventuallyWithin}(f, l, u)$ monitors that *f* occurs within the relative time bounds *l* (lower bound) and *u* (upper bound), measured in seconds. It is defined with the help of the auxiliary rule `EventuallyAbs`, which is an absolute-timed version.

$$\begin{aligned}&\textbf{min}\ \texttt{EventuallyAbs}(\textbf{Form}\ f, \textbf{int}\ al, \textbf{int}\ au) =\\&\quad \mathit{time} \leqslant au\ \wedge\\&\qquad ((f\ \wedge\ \mathit{time} \geqslant al)\ \vee\\&\qquad (\neg f\ \wedge\ \bigcirc\texttt{EventuallyAbs}(f, al, au)))\end{aligned}$$

$$\begin{aligned}&\textbf{min}\ \texttt{EventuallyWithin}(\textbf{Form}\ f, \textbf{int}\ l, \textbf{int}\ u) =\\&\quad \texttt{EventuallyAbs}(f, \mathit{time} + (l * 1000), \mathit{time} + (u * 1000))\end{aligned}$$

Note that variable *time* is defined in the state and contains the latest time stamp in milliseconds since the start of the application. The property “*when the rover starts a turn, the turn should end within* 10–30 s” can now be stated as follows:

$$\begin{aligned}&\textbf{mon}\ \texttt{M5} = \texttt{Always}(\mathit{start}(\text{“}\mathit{turn}\text{”})\\&\quad \rightarrow \texttt{EventuallyWithin}(\texttt{end}(\text{“}\mathit{turn}\text{”}), 10, 30))\end{aligned}$$

3.5. Online evaluation algorithm

A *monitoring algorithm* for EAGLE determines whether a trace σ models a monitoring specification $D\ O$. Our algorithm operates in an online fashion. That is, it is applied sequentially at each state of σ and does not refer back to prior 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 logic, e.g. LTL, then the EAGLE algorithm should perform about as well as an efficient algorithm for the restricted logic. We have for example proved this for LTL [7].

The algorithm employs a function $eval(F, s)$ that examines a state, s, and transforms a monitor F into a monitor F' such that $s \frown \sigma, 1 \models_D F$ iff $s \frown \sigma, 2 \models_D F'$.

The algorithm is, where possible, a direct implementation of the definition of the 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 used by *eval* on sub-formulas headed by the $\odot$ operator, that records the result of (partially) evaluating the formula in the previous state.

This is illustrated as follows:

$$\begin{array}{l}\textbf{min}\ \mathsf{R}(\textbf{int}\ k) = \odot(y + 1 == k)\\ \textbf{mon}\ \mathtt{M} = \mathtt{Eventually}(R(x))\end{array}$$

This monitor will be true if somewhere in the trace there are two successive states such that 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 in different states.

Since *eval* recursively decomposes the formulas, eventually *eval* will be called on $\odot(y + 1 == k)$. 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 of any formula P headed by a previous-state operator, in this case 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. [5] Also note that since 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

[5] A simpler alternative to partial evaluation is to form a closure and do the complete evaluation when all variables are bound.

state. This is done by expanding monitor formulas by unfolding rule definitions, while avoiding infinite expansion due to recursive rule definitions. At the end of the trace, function *value* is called yielding a truth value as the final result of evaluating each monitor over the trace. Function *value* implements the EAGLE semantics with respect to boundary conditions regarding the end of the trace.

Function *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, next, concatenation or sequential composition operator or a rule application. Subsumption, i.e., simplifying $(a \wedge b) \vee a$ to a, is essential.

3.6. *Complexity of* EAGLE

It is evident from the semantics given in Section 3.3 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. However, what is of interest is the performance of EAGLE on special cases, i.e., for arbitrary monitors defined over fixed rule sets that implement standard temporal logics. Furthermore one must distinguish complexity due to any data computation ascribed to methods defined for state update and predicate evaluation from the evaluation of the purely temporal aspects 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 LTL, over both future and past. In [7], we show that the step evaluation for an initial LTL formula of size m has an upper time complexity bound of $\mathrm{O}(m^4 2^{2m} \log^2 m)$ and a space bound of $\mathrm{O}(m^2 2^m \log m)$, 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 $\mathrm{O}(2^{\sqrt{m}})$ for monitoring LTL given in [59].

For MTL where time constants are stated as natural numbers, embedded in EAGLE, it can be shown that the time and space complexity of monitoring of a formula is $2^{\mathrm{O}(m)}$ where m is the size of the monitored formula plus the sum of all time constants that appear in the formula. Note that the bound, although exponential, is independent of the length of the trace. The proof for this complexity bound is similar to the proof of the same result in [63].

For real-time logic where time constants are stated as real numbers, embedded in EAGLE, the time and complexity bound, although independent of the length of the trace, is dependent on the minimum of all time differences between any two events in the trace. The bound is given by $2^{\mathrm{O}(mt/\delta)}$ where m is the size of the formula, t is the sum of all time constants appearing in the formula and δ is the minimum time difference between any two events in the trace monitored.

3.7. *A Java library for monitoring* EAGLE *properties*

The EAGLE monitoring engine implements the EAGLE monitoring algorithm as a Java library. The library provides three basic methods, `parse`, `eval`, and `value`, that can be called by any client program for the purpose of monitoring. The first method `parse` takes a file containing a specification involving several monitors (sets of monitored formulas) written in EAGLE and compiles them internally into data structures representing monitors. After compilation, the client program calls the method `eval` iteratively with an observer state. This call internally modifies the monitors according to the definition of *eval* in Section 3.5. If a monitored formula becomes false during this modification, it calls a method `error` which the client program is expected to implement. Similarly, if a formula becomes true the method `success` is called. It is up to the client program to define the observer state. The client program also modifies the observer state at every event. Once all the events are consumed the client program calls the method `value` to check if the monitored formulas are satisfied by the sequence of observer states. If a formula is not satisfied the method `warning` implemented by the client is called; otherwise, the method `nowarning` is invoked.

3.8. Concurrency analysis

A scheduler may schedule the different threads in a multi-threaded program, such as the rover controller, in a non-deterministic manner, causing the order in which threads access shared objects to differ among different executions on the same input. This may lead to different observed execution traces, causing temporal logic specifications to be violated in some traces while not being violated in others. Consequently one cannot infer that a temporal property holds for *all* traces (that is, holds for the program on some particular input) based on the observation that it holds on *some* trace. The ideal solution would be a framework for transforming temporal properties to stronger properties that when checked will be less sensitive to the non-determinism of traces. Ideally one would like to be able to infer that if the property holds on some trace then with high probability it holds on all traces. Or perhaps more importantly: if the property is violated on some trace then it is violated with high probability on any trace, thereby increasing our chance of detecting the problem on a random trace. Although this may appear a very difficult problem to solve for the general case, it actually can be done for certain properties that are generally desirable for concurrent programs: deadlock freedom and data race freedom.

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 potential since thread T_1 can acquire A where upon thread T_2 can acquire B, resulting in a deadlocked situation. Potentials for such deadlocks can be detected by identifying cycles in lock graphs [10]. Another main issue for programmers of multi-threaded applications is to avoid *data races* where several threads access a shared object simultaneously. If all threads utilize the same lock when accessing an object, mutual exclusion is guaranteed, otherwise data races are possible. The Eraser algorithm [56] can detect such data races by maintaining a so-called lock set for each monitored variable. Recent work [3] has identified another kind of data races, termed *high-level data races*, that are not detectable by the Eraser algorithm. These races can occur when sets of fields are accessed incorrectly. Monitors have been developed for analyzing traces for the three above-mentioned concurrency problems and automated instrumentation has been done for Java. For C++, manual instrumentation for deadlock analysis has been done using wrapping as mentioned earlier. For the two kinds of data race analysis, automated instrumentation of C++ remains to be done.

Although the above-mentioned concurrency algorithms have been implemented as specialized programs, one can well imagine using EAGLE for specifying such properties. As an experiment, the deadlock detection algorithm has been encoded in EAGLE as described in [7], although restricted to the detection of deadlocks between pairs of threads. The general algorithm described in [10] can detect deadlock potentials between any number of threads. Further work will integrate the concurrency algorithms and EAGLE fully.

4. Case study: a planetary Rover controller

The subject of the case study described here is a controller for the K9 planetary rover, developed at NASA Ames Research Center. A full account of this controller is described in [12]. The case study was done in collaboration with the programmer of the controller. First we present a description of the rover controller, including a description of the plan language (the input to the controller). Then, an outline is given of how plans (test inputs) and associated temporal logic properties can be automatically generated using model checking.

4.1. The Rover controller

The rover controller 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 executive executes.

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. Fig. 7 (left) shows the grammar for the plan language. All node attributes, with the exception of the *id* of the node, are optional. Each node may specify a set of *conditions*, e.g. the *start condition* (that must be true at the beginning of node execution) and the *end condition* (that must be true at the end of node execution). Each condition includes information about a relative or absolute time window, indicating a lower and an upper bound on the time. Flag *continue-on-failure* indicates what the behavior will be when an node failure is encountered. Attribute *duration* specifies the duration of the action. Fig. 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).

```
Plan      → Node
Node      → Block | Task
Block     → (block
             NodeAttr
             :node-list (NodeList))
NodeList  → Node NodeList | ε
Task      → (task
             NodeAttr
             :action Symbol
             :duration DurationTime)
NodeAttr  → :id Symbol
             [:start-condition Condition]
             [:end-condition Condition]
             [:continue-on-failure]
Condition → (time StartTime EndTime)
```

```
(block
  :id plan
  :continue-on-failure
  :node-list (
    (task
     :id drive1
     :start-condition (time +1 +5)
     :end-condition   (time +1 +30)
     :action BaseMove1
     :duration 20
    )
    (task
     :id drive2
     :end-condition (time +10 +16)
     :action BaseMove2
     :duration 20
) ) )
```

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

4.1.2. Plan semantics

For every node, execution proceeds through the following steps:

- Wait until the start condition is satisfied; if the current time passes the end of the start condition, the node times out and this is a node failure.
- The execution of a *task* proceeds by invoking the corresponding action (e.g. a routine that interacts with the rover hardware). The action takes the time specified in the `:duration` attribute when the software is run in simulation mode, with a hardware simulator. The task succeeds or fails, for example depending on whether the time window is respected. The execution of a *block* simply proceeds by executing each of the nodes in the `node-list` in order.
- If time exceeds the end condition, the node fails. On a *node failure*, when execution returns to the sequence, the value of flag *continue-on-failure* 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 node. 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

Fig. 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) and properties (i.e., test oracles, as discussed in the next section). The framework described in Section 2 is used to generate test inputs from a specification written as an annotated Java program. Model checking with symbolic execution generates the inputs. The input plans are specified using non-deterministic choice (`choose` methods) over the structures allowed in the grammar presented in Fig. 7 and constraints over the integer variables in the input structure (updates to the path condition `_pc`). For brevity, only a small sample set of constraints is shown here (stating that the time points are proper positive values defining intervals and the end time is larger than the start time of an interval). The actual testing requires solving these constraints in order to

```
class UniversalPlanner { ...
  static int nNodes; /*max number of nodes*/
  static void Plan(int nn) {
    nNodes = nn;
    Node plan = UniversalNode();
    print(plan);
    compute_and_print_properties(plan);
    assert(false);
  }
  static Node UniversalNode() {
    if (nNodes == 0) return null;
    if (chooseBool()) return null;
    if (chooseBool())
      return UniversalTask();
    return UniversalBlock();
   }
  static Node UniversalTask() {
    int id = nNodes; nNodes--;
    UniversalAttributes();
    Task t = new Task(id, start, end,
                 continueOnFailure,duration);
    return t;
  }
  static Node UniversalBlock() {
    int id = nNodes; nNodes--;
    ListOfNodes l = new ListOfNodes();
    for (Node n = UniversalNode();n != null;
         n = UniversalNode())  l.add(n);
    UniversalAttributes();
    Block b = new Block(id, l, start, end,
                        continueOnFailure);
    return b;
  }
```

```
static TimeCondition start, end;
static int duration;
static boolean continueOnFailure;

static UniversalAttributes() {
  id = new Symbol();
  SymInt sTime1 = new SymInt();
  SymInt sTime2 = new SymInt();
  SymInt eTime1 = new SymInt();
  SymInt eTime2 = new SymInt();
  SymInt d = new SymInt();

  /* constraints */
  SymInt._pc._add_GE(sTime1,0);...
  SymInt._pc._add_LT(sTime1,sTime2);
  SymInt._pc._add_LT(eTime1,eTime2);
  SymInt._pc._add_LE(sTime1,eTime1);
  ...
  duration = d.solution();
  start = new TimeCondition(sTime1.solution(),
                            sTime2.solution());
  end = new TimeCondition(eTime1.solution(),
                          eTime2.solution());
  continueOnFailure = chooseBool();
} }
```

Fig. 8. Code that generates input plans and properties.

```
class SymInt { ...
 static PathCondition _pc;
 ...
 int solution() { ... }
}
```

```
class PathCondition { ...
 Constraints c;
 void _add_LT(SymInt e1, SymInt e2){
  c.add_constraint_LT(e1,e2);
  if (!c.is_satisfiable())
     backtrack();
  return;
 }
}
```

Fig. 9. Library classes for symbolic execution.

instantiate input plans that can be then executed (method `solution`). To illustrate the flexibility of our approach, some of the variables are considered concrete inputs, e.g. the maximum allowed number of nodes in a generated structure (`nNodes`) and yet others, e.g. the boolean values, are generated using non-deterministic choice.

The assertion in the program, at the end of the `Plan` method, specifies that it is not possible to create a "valid" plan (i.e., executions that reach this assertion generate valid plans). JPF model checks the universal planner and is thus used to explore the state space of the input plans that have up to `nNodes` nodes. Different search strategies find multiple counter-examples; for each counter-example (representing a valid plan), a set of properties associated with the plan is computed. The generated plan and properties are printed to files that are then used for testing the rover.

Fig. 9 gives part of the library classes that enable JPF to perform symbolic execution. Class `SymInt` supports manipulation of symbolic integers. The `static` field `SymInt._pc` stores the (numeric) path condition. Method `_add_LT` updates the path condition with a constraint encoding `e1` *less-than* `e2`. 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 for a symbolic integer.

4.3. Property generation

For each generated plan, a set of properties formulated in the EAGLE temporal logic is automatically generated, according to the semantics of the planning language. Note that such a set of properties is generated for each plan and monitored during the execution of that specific plan. In generating these properties, the following predicates are used: start(id) (true immediately after the start of the execution of the node with the corresponding *id*), end(id) (true when the execution of the node ends successfully) and fail(id) (true when the execution of the node ends with a failure). The code has been instrumented to monitor these predicates and the validity of the generated properties is checked on execution traces. As an example, some of the generated properties for the plan from Fig. 7 (right) are shown in Fig. 10.

The set of generated properties does not fully represent the semantics of the plan. As an example, the illustrated properties do not state the fact that `drive1` should only start once. A complete specification of the plan semantics would require a more elaborate set of formulas. This would be possible since EAGLE is a very expressive logic. However, the current set of properties generated for a plan seems appropriate to catch many kinds of errors. The effort invested in designing what properties to be generated for a particular plan was minimal and likely so due to the fact that not all the plan semantics is modeled. The properties could be inferred very directly from the informal plan semantics communicated by the engineer that programmed the system.

4.4. Results

The tool is fully automated after setup and does not require any input from the user to run. The tool generates a set of test cases, each consisting of a plan (input) and a set of properties (expected of the output). A script will execute each test case, first by running the controller, together with a rover hardware simulator, on the input plan and then calling EAGLE to verify that the generated execution trace satisfies the properties. Due to the automated nature of the process, the developer of the K9 rover controller is capable of running it himself. All test results used in the process have been generated by the developer running the tool.

The automated testing system found (in the first application) a missing feature that had been overlooked by the developers: the lower bounds on execution duration were not enforced. Hence, where a certain generated temporal logic formula predicted failure, the execution in fact wrongly succeeded, and this was detected as a violation of the temporal property. The error was not corrected immediately after its detection, and showed up later during actual rover operation in a field test before it was corrected. A preliminary version of the testing environment, not using automated test case generation, found a deadlock and a data race. The data race, involving access to a shared variable used to communicate

- `M1 = Eventually(start("plan"))`
  i.e., the initial node `plan` *should eventually start.*
- `M2 = Always(start("plan") -> Eventually(end("plan")))`
  i.e., if `plan` *starts, then it should eventually terminate successfully.*
- `M3 = Always(start("plan") -> EventuallyWithin(start("drive1"),1,5))`
  i.e., if `plan` *starts, then* `drive1` *should start within 1 and 5 time units.*
-
```
M4 = Always((end("drive2") \/ fail("drive2")) ->
                   Eventually(end("plan")))
```
  i.e., successful or failed termination of `drive2` *implies successful termination of the whole plan (due to* `continue-on-failure` *flag).*
-
```
M5 = Always(start("drive1") ->
                  ( EventuallyWithin(end("drive1"),1,30) \/
                    Eventually(fail("drive1"))))
```
  i.e. if `drive1` *starts, then it should end successfully within 1 and 30 time units or it should eventually terminate with a failure.*
- `M6 = Always(fail("drive1") -> ~  Eventually(start("drive2")))`
  i.e., if `drive1` *fails, then* `drive2` *should not start.*
- `M7 = Always(end("drive1") -> Eventually(start("drive2")))`
  i.e., if `drive1` *ends successfully, then* `drive2` *should eventually start.*
- `M8 = Always(start("drive2") -> Eventually(fail("drive2")))`
  i.e., if `drive2` *starts, then it should eventually fail (due to the time conditions).*

Fig. 10. Properties representing partial semantics of plan in Fig. 7.

between threads, was suspected by the developer, but had not been confirmed in code. The trace analysis allowed the developer to see the read/write pattern clearly and redesign the communication.

The K9 rover controller, essentially an interpreter, seemed to be very well suited for this kind of testing framework. It was in particular simple to determine what temporal properties should be generated for a plan. This is, however, not as easy in general for other kinds of applications. Another drawback is the fact that only events of the form *start*, *end* and *fail* are monitored. Hence, failures which can only be detected by monitoring sub-events between these events cannot be observed.

5. Related work

5.1. Test case generation

In Section 2, we have already discussed some of the related work on specification-based testing. Here we link our approach to test input generation tools.

The idea of using constraints to represent inputs dates back at least three decades [36,15,39,54]; the idea has been implemented in various tools including EFFIGY [39], TESTGEN [41] and INKA [25]. Most of this work has been focused on solving constraints on primitive data, such as integers and booleans.

Some recent frameworks, most notably TestEra [45] and Korat [11,44], do support generation of complex structures. TestEra generates inputs from constraints given in Alloy, a first-order declarative language based on relations. TestEra uses off-the-shelf SAT solvers to solve constraints. Korat generates inputs from constraints given as Java predicates. The Korat algorithm has recently been included in the AsmL Test Generator [21] to enable generation of structures. TestEra and Korat focus on solving structural constraints. They do not directly solve constraints on primitive data as we do in our framework. Instead, they systematically try all primitive values within given bounds, which may be inefficient.

The first version of AsmLT Test Generator [26] was based on finite-state machines (FSMs): an AsmL [29] specification is transformed into an FSM and different traversals of the FSM are used to construct test inputs. Korat adds structure generation to generation based on finite-state machines [26]. AsmLT was successfully used for detecting faults in a production-quality XPath compiler [62].

Several researchers have investigated the use of model checking for test input generation (see [35] for a good survey). Gargantini and Heitmeyer [22] use a model checker to generate tests that violate known properties of a specification given in the SCR notation. Ammann and Black [2] combine model checking and mutation analysis to generate test cases from a specification. Rayadurgam et al. use a structural coverage-based approach to generate test cases from specifications given in RSML^{-e} by using a model checker [34]. Hong et al. formulate a theoretical framework for using temporal logic to specify data-flow test coverage in [35]. These approaches cannot handle structurally complex inputs.

There are many tools that produce test inputs from a description of tests. QuickCheck [14] is a tool for testing Haskell programs. It requires the tester to write Haskell functions that can produce valid test inputs; executions of such functions with different random seeds produce different test inputs. Our work differs in that it requires only a specification that characterizes valid test inputs and then uses a general-purpose search to generate *all* valid inputs up to a certain size. DGL [47] and lava [61] generate test inputs from production grammars. They were used mostly for random testing, although they can also systematically generate test inputs. However, they cannot easily represent inputs with complex structure, as we do by using Java as a specification language.

5.2. *Runtime verification*

The EAGLE logic and its implementation for runtime monitoring has been significantly influenced by earlier work on the executable, trace-generating as well as trace-checking, temporal logic METATEM [6]. In the parallel work [43] a framework is described where recursive equations are used to implement a real-time logic. Although this is a similar approach to the one presented in this paper, EAGLE goes much further and provides the language of recursive equations to the user, supporting a mixture of future-time and past-time operators and treating real time as a special case of data values, hence allowing a more general logic.

The most directly case study specific related work is presented in [9], which for the same rover application describes a framework for generating timed automata from plans. From the timed automata, monitors are generated that can monitor the plan execution. Since EAGLE can embed timed automata, EAGLE can be seen as a more general framework, that also allows for more partial temporal logic specifications. The main difference in approach is that [9] defines the full semantics of a plan, whereas the temporal logic approach presented here defines a partial semantics.

At a more general level, several runtime verification systems have recently been developed, a collection of which have been presented at a series of runtime verification (RV) workshops [20]. LTL [51] has been core to several of these attempts. The MaC tool [38] supports a past-time interval temporal logic. Real-time is modeled by introducing an explicit state in the specification, containing explicit clock variables, which get updated when new events arrive. The commercial tools Temporal Rover and DBRover [16,17], support future-time and past-time LTL properties, annotated with real-time and data constraints. Alternating automata algorithms to monitor LTL properties are proposed in [19] and a specialized LTL collecting statistics along the execution trace is described in [18]. Various algorithms to generate testing automata from temporal logic formulas are discussed in [55,49]. Complexity results for testing a finite trace against temporal formulas expressed in different temporal logics are investigated in [46]. A technique where execution events are stored in an SQL database at runtime is proposed in [42]. These events are then analyzed by queries derived from interval logic temporal formulas after the program terminates. The PET tool [28] uses a future-time temporal logic formula to guide the execution of a program for debugging purposes. The model-based specification language AsmL is being used for runtime verification [5], as well as for test case generation (see Section 5.1). AsmL is a very comprehensive general-purpose specification language for abstractly specifying computation steps. It does not directly support temporal logic.

Our own related work includes the development of several algorithms for monitoring with temporal logic, such as generating dynamic programming algorithms for past-time logic [33], using a rewriting system for monitoring future-time logic [32,31], or generating Büchi-automata-inspired algorithms adapted to finite-trace LTL [23]. A logic based on extended regular expressions is described in [58]. Java MultiPathExplorer [60] is a tool which checks a past-time LTL safety formula against a partial order extracted online from an execution trace. POTA [57] is another partial-order trace analyzer system. Java-MoP [13] is a generic logic monitoring tool encouraging "monitoring-oriented programming". JNuke [4] is a framework that combines runtime verification and model checking. It is written in C, achieving scalability through high performance and low memory usage.

6. Conclusions and future work

A framework for testing based on automated test case generation and runtime verification has been presented. This paper proposed and demonstrated the use of model checking and symbolic execution for test case generation using the JAVA PATHFINDER tool, and the use of temporal logic monitoring in EAGLE during the execution of the test cases. The framework requires construction of a test input and property generator for the application. From that, a large test suite can be automatically generated, executed and verified to be in conformity with the properties. For each input a set of EAGLE properties is generated 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. Due to EAGLE's expressive power, properties can furthermore be stated in combinations of different sub-logics and notations, such as for example temporal logic, regular expressions and state machines. While property definition in general often is difficult, an effective approach for some domains may be to write a property generator, rather than a universal set of properties that are independent of the test inputs. 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 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 improvements to the EAGLE logic and its engine. Experiments will be made combining different specification paradigms, such as temporal logic, regular expressions and state machines, all currently expressible in EAGLE within a single framework. Furthermore, an attempt will be made to integrate the concurrency analysis algorithms for deadlock and data race analysis fully into EAGLE. 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] AGEDIS—model based test generation tools, http://www.agedis.de.

[2] P. Ammann, P. Black, A specification-based coverage metric to evaluate test sets, in: Proc. Fourth IEEE Internat. Symp. on High Assurance Systems and Engineering, November 1999, pp. 239–248.

[3] C. Artho, K. Havelund, A. Biere, High-level data races, Journal on Software Testing, Verification and Reliability (STVR) 13 (4) (2003).

[4] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, B. Zweimüller, JNuke: efficient dynamic analysis for Java, in: Proc. CAV'04: Computer Aided Verification, Lecture Notes in Computer Science, Springer, Berlin, 2004.

[5] M. Barnett, W. Schulte, Contracts, components, and their runtime verification, Technical Report MSR-TR-2002-38, Microsoft Research, April 2002, Download: http://research.microsoft.com/fse.

[6] H. Barringer, M. Fisher, D. Gabbay, G. Gough, R. Owens, METATEM: an introduction, Formal Aspects Comput. 7 (5) (1995) 533–549.

[7] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Program monitoring with LTL in EAGLE, in: Proc. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD'04), Santa Fe, New Mexico, USA, April 2004.

[8] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, in: B. Steffen, G. Levi (Eds.), Proc. Fifth Internat. Conf. on Verification, Model Checking and Abstract Interpretation, Lecture Notes in Computer Science, Vol. 2937, Springer, Berlin, January 2004, pp. 44–57.

[9] S. Bensalem, M. Bozga, M. Krichen, S. Tripakis, Testing conformance of real-time software by automatic generation of observers, in: Proc. RV'04: Fourth Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, vol. 113, Barcelona, Spain, Elsevier Science, Amsterdam, 2004.

[10] S. Bensalem, K. Havelund, Deadlock analysis of multi-threaded Java programs, Kestrel Technology, NASA Ames Research Center, CA, October 2002.

[11] C. Boyapati, S. Khurshid, D. Marinov, Korat: automated testing based on Java predicates, in: Proc. Internat. Symp. Software Testing and Analysis (ISSTA), July 2002, pp. 123–133.

[12] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, Experimental evaluation of verification and validation tools on Martian rover software, in: SEI Software Model Checking Workshop, 2003, extended version, J. Formal Methods System Design, 25 (2) (2004).

[13] F. Chen, G. Roşu, Towards monitoring-oriented programming: a paradigm combining specification and implementation, in: Proc. RV'03: the Third Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, Boulder, USA, Vol. 89(2), Elsevier Science, Amsterdam, 2003.

[14] K. Claessen, J. Hughes, Testing monadic code with QuickCheck, in: Proc. ACM SIGPLAN Workshop on Haskell, 2002, pp. 65–77.

[15] L.A. Clarke, A system to generate test data and symbolically execute programs, IEEE Trans. Software Engrg. SE-2 (1976) 215–222.

[16] D. Drusinsky, The temporal rover and the ATG rover, in: Proc. SPIN'00: SPIN Model Checking and Software Verification, Lecture Notes in Computer Science, Vol. 1885, Stanford, CA, USA, Springer, Berlin, 2000, pp. 323–330.

[17] D. Drusinsky, Monitoring temporal rules combined with time series, in: Proc. CAV'03: Computer Aided Verification, Lecture Notes in Computer Science, Vol. 2725, Boulder, USA, Springer, Berlin, 2003, pp. 114–118.

[18] B. Finkbeiner, S. Sankaranarayanan, H. Sipma, Collecting statistics over runtime executions, in: Proc. RV'02: The Second Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, Vol. 70(4), Paris, France, Elsevier Science, Amsterdam, 2002.

[19] B. Finkbeiner, H. Sipma, Checking finite traces using alternating automata, Formal Methods System Design 24 (2) (2004) 101–128.

[20] First, Second, Third and Fourth Workshops on Runtime Verification (RV'01–RV'04), ENTCS, Vol. 55(2), 70(4), 89(2), 113, Elsevier Science, Amsterdam, 2001, 2002, 2003, 2004.

[21] Foundations of Software Engineering, Microsoft Research, The AsmL test generator tool. http://research.microsoft.com/fse/asml/doc/AsmLTester.html.

[22] A. Gargantini, C. Heitmeyer, Using model checking to generate tests from requirements specifications, in: Proc. Seventh European Engineering Conf. Held Jointly with the Seventh ACM SIGSOFT Internat. Symp. Foundations of Software Engineering, Springer, Berlin, 1999, pp. 146–162.

[23] D. Giannakopoulou, K. Havelund, Automata-based verification of temporal properties on running programs, in: Proc. ASE'01: Internat. Conf. on Automated Software Engineering, Institute of Electrical and Electronics Engineers, Coronado Island, CA, 2001, pp. 412–416.

[24] A. Goldberg, K. Havelund, Instrumentation of Java bytecode for runtime analysis, in: Proc. Formal Techniques for Java-like Programs, Technical Reports from ETH Zurich, Vol. 408, Switzerland, ETH Zurich, 2003.

[25] A. Gotlieb, B. Botella, M. Rueher, Automatic test data generation using constraint solving techniques, in: Proc. Internat. Symp. Software Testing and Analysis (ISSTA), Clearwater Beach, FL, March 1998, pp. 53–62.

[26] W. Grieskamp, Y. Gurevich, W. Schulte, M. Veanes, Generating finite state machines from abstract state machines, in: Proc. Internat. Symp. Software Testing and Analysis (ISSTA), July 2002, pp. 112–122.

[27] A. Groce, W. Visser, Model checking Java programs using structural heuristics, in: Proc. 2002 Internat. Symp. Software Testing and Analysis ISSTA, ACM Press, New York, July 2002, pp. 12–21

[28] E. Gunter, D. Peled, Tracing the executions of concurrent programs, in: Proc. of RV'02: Second Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, Vol. 70(4), Copenhagen, Denmark, Elsevier Science, Amsterdam, 2002.

[29] Y. Gurevich, Evolving algebras 1993: lipari guide, in: Specification and Validation Methods, Oxford University Press, Oxford, 1995, pp. 9–36.

[30] A. Hartman, Model based test generation tools, `http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf`.

[31] K. Havelund, G. Roşu, Monitoring programs using rewriting, in: Proc. of the Internat. Conf. Automated Software Engineering (ASE'01), IEEE CS Press, Coronado Island, CA, 2001, pp. 135–143, extended version to appear in J. Automat. Software Engrg.

[32] K. Havelund, G. Roşu, An overview of the runtime verification tool Java PathExplorer, Formal Methods System Design 24 (2) (2004) 189–215.

[33] K. Havelund, G. Roşu, Synthesizing monitors for safety properties, in: Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), Lecture Notes in Computer Science, Vol. 2280, Springer, Berlin, 2002, pp. 342–356, extended version in Internat. J. Software Tools Technol. Transfer 6 (2) (2004) 158–173.

[34] M.P.E. Heimdahl, S. Rayadurgam, W. Visser, D. George, J. Gao, Auto-generating test sequences using model checkers: a case study, in: Proc. Third Internat. Workshop on Formal Approaches to Testing of Software (FATES), Lecture Notes in Computer Science, Vol. 2931, Montreal, Canada, Springer, Berlin, October 2003, pp. 42–59.

[35] H.S. Hong, I. Lee, O. Sokolsky, H. Ural, A temporal logic based theory of test coverage and generation, in: Proc. Eighth Internat. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science, Vol. 2280, Grenoble, France, Springer, Berlin, April 2002, pp. 327–341.

[36] J.C. Huang, An approach to program testing, ACM Comput. Surv. 7 (3) (1975).

[37] S. Khurshid, C. Pasareanu, W. Visser, Generalized symbolic execution for model checking and testing, in: Proc. TACAS'03: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 2619, Warsaw, Poland, April 2003.

[38] M. Kim, M. Viswanathan, S. Kannan, I. Lee, O. Sokolsky, Java-MaC: a run-time assurance tool for Java, Formal Methods System Design 24 (2) (2004) 129–156.

[39] J.C. King, Symbolic execution and program testing, Comm. ACM 19 (7) (1976) 385–394.

[40] B. Korel, Automated software test data generation, IEEE Trans. Software Engrg. 16 (8) (1990) 870–879.

[41] B. Korel, Automated test data generation for programs with procedures, in: Proc. Internat. Symp. Software Testing and Analysis (ISSTA), San Diego, CA, 1996, pp. 209–215.

[42] D. Kortenkamp, R. Simmons, T. Milam, J. Fernandez, A suite of tools for debugging distributed autonomous systems, Formal Methods System Design 24 (2) (2004) 157–188.

[43] K. Jelling Kristoffersen, C. Pedersen, H.R. Andersen, Runtime verification of timed LTL using disjunctive normalized equation systems, in: Proc. RV'03: Third Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, Vol. 89(2), Boulder, USA, Elsevier Science, Amsterdam, 2003.

[44] D. Marinov, Testing using a solver for imperative constraints, Ph.D. Thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004, to appear.

[45] D. Marinov, S. Khurshid, TestEra: a novel framework for automated testing of Java programs, in: Proc. 16th IEEE Internat. Conf. Automated Software Engineering ASE, San Diego, CA, November 2001, pp. 22–34.

[46] N. Markey, P. Schnoebelen, Model checking a path (preliminary report), in: Proc. CONCUR'03: Internat. Conf. Concurrency Theory, Lecture Notes in Computer Science, Vol. 2761, Marseille, France, Springer, Berlin, August 2003, pp. 251–265.

[47] P.M. Maurer, Generating test data with enhanced context-free grammars, IEEE Software 7 (4) (1990) 50–55.

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

[49] T. O'Malley, D. Richardson, L. Dillon, Efficient specification-based oracles for critical systems, in: Proc. California Software Symp., 1996.

[50] Parasoft, http://www.parasoft.com.

[51] A. Pnueli, The temporal logic of programs, in: Proc. 18th IEEE Symp. Foundations of Computer Science, 1977, pp. 46–77.

[52] W. Pugh, A practical algorithm for exact array dependence analysis, Comm. ACM 35 (8) (1992) 102–114.

[53] Purify: Fast Detection of Memory Leaks and Access Errors, January 1992.

[54] C.V. Ramamoorthy, Siu-Bun F. Ho, W.T. Chen, On the automated generation of program test data, IEEE Trans. Software Engrg. 2 (4) (1976) 293–300.

[55] D.J. Richardson, S.L. Aha, T.O. O'Malley, Specification-based test oracles for reactive systems, in: Proc. ICSE'92: Internat. Conf. Software Engineering, Melbourne, Australia, 1992, pp. 105–118.

[56] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, ACM Trans. Comput. Systems 15 (4) (1997) 391–411.

[57] A. Sen, V.K. Garg, Partial order trace analyzer (POTA) for distributed programs, in: Proc. RV'03: the Third Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, Vol. 89(2), Boulder, USA, Elsevier Science, Amsterdam, 2003.

[58] K. Sen, G. Roşu, Generating optimal monitors for extended regular expressions, in: Proc. RV'03: Third Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, Vol. 89(2), Boulder, USA, Elsevier Science, Amsterdam, 2003.

[59] K. Sen, G. Roşu, G. Agha, Generating optimal linear temporal logic monitors by coinduction, in: V.A. Saraswat (Ed.), Proc. Eighth Asian Computing Science Conf. (ASIAN'03), Lecture Notes in Computer Science, Vol. 2896, December 2003, pp. 260–275.

[60] K. Sen, G. Roşu, G. Agha, Runtime safety analysis of multithreaded programs, in: Proc. ESEC/FSE'03: European Software Engineering Conf. and ACM SIGSOFT Internat. Symp. on the Foundations of Software Engineering, ACM, Helsinki, Finland, September 2003, pp. 337–346.
[61] E.G. Sirer, B.N. Bershad, Using production grammars in software testing, in: Proc. Second Conf. on Domain-specific languages, 1999, pp. 1–13.
[62] K. Stobie, Advanced modeling, model based test generation, and abstract state machine language AsmL, http://www.sasqag.org/pastmeetings/asml.ppt, 2003.
[63] P. Thati, G. Roşu, Monitoring algorithms for metric temporal logic, in: Proc. RV'04: Fourth Internat. Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, vol. 113, Barcelona, Spain, Elsevier Science, Amsterdam, 2004.
[64] The test sequence generator TGV, http://www-verimag.imag.fr/∼async/TGV.
[65] N. Tracey, J. Clark, K. Mander, The way forward for unifying dynamic test-case generation: the optimisation-based approach, in: Internat. Workshop on Dependable Computing and Its Applications (DCIA), IFIP, January 1998, pp. 169–180.
[66] T-VEC, http://www.t-vec.com.
[67] W. Visser, K. Havelund, G. Brat, S.-J. Park, F. Lerda, Model checking programs, Automat. Software Engrg. J. 10 (2) (2003) 203–232.