Experiments with Test Case Generation and Runtime Analysis

Cyrille Artho[1], Doron Drusinksy[2], Allen Goldberg[3], Klaus Havelund[3], Mike Lowry[4], Corina Pasareanu[3], Grigore Roşu[5], and Willem Visser[6]

[1] Computer Systems Institute, ETH Zurich, CH-8092 Zurich, Switzerland
[2] Naval Postgraduate School, Monterey, California, USA, and Time Rover, Inc, Cupertino, California, USA
[3] Kestrel Technology, NASA Ames Research Center, Moffett Field, California USA
[4] NASA Ames Research Center, Moffett Field, California USA
[5] Department of Computer Science, University of Illinois at Urbana-Champaign
[6] RIACS, NASA Ames Research Center, Moffett Field, California USA

Abstract. Software testing is typically an ad hoc process where human testers manually write many test inputs and expected test results, perhaps automating their execution in a regression suite. This process is cumbersome and costly. This paper reports preliminary results on an approach to further automate this process. The approach 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, or analyzed using advanced algorithms for detecting concurrency errors such as data races and deadlocks. The approach suggests to generate specifications dynamically per input instance rather than statically once-and-for-all. The paper describes experiments with variants of this approach in the context of two examples, a planetary rover controller and a space craft fault protection system.

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 procedure seems complicated but ultimately 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 [3] is described a case study, where an 8,000 line Java application was tested by different student groups using different testing techniques. 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 reflections and preliminary work on applying low-budget automated testing to identify bugs quickly. The paper shall be seen as a position statement of future work, based on experiments using our testing tools on case studies.

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.*

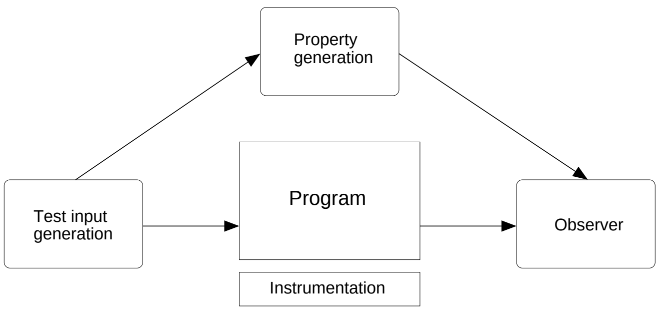

Fig. 1. Test case generation (test input generation and property generation) and runtime analysis (instrumentation and observation).

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 "observes" the executing program, checking its behavior against the generated set of properties. That is, the observer takes as input an execution trace and the set of properties generated by the property generator. 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 many 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 instrumentation and observer modules are generic tools that are re-used on different applications. In the rest of this paper we use the term *test case generation* to refer to test input generation and property generation, and the term *runtime analysis* to refer to instrumentation as well as observation.

The above described framework was applied to two case studies, a planetary rover controller and a space craft fault protection system. In each case the system properties were expressed in temporal logic. For the rover controller, test cases were generated using a model checker and the properties generated were *specific to a single test case.* For the fault protection system, test cases were generated by a small program, and *universal correctness properties* were manually constructed.

Property generation is the difficult step in this process. We are investigating problem characteristics and tradeoffs between the two approaches used in the studies. The approach of generating properties specific to a single test case is more novel and will be investigated further.

The paper is organized as follows. Section 2 outlines the abstract framework for test case generation that we have tried to adhere to. Section 3 describes the runtime analysis techniques. Sections 4 and 5 describe the two case studies. Finally Section 6 concludes the paper and outlines how this preliminary work will be continued.

2 Test Case Generation

This section presents, in abstract, the test case generation framework. As mentioned earlier, we consider test generation as consisting of *test input generation* and *property generation*.

2.1 Test Input Generation

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 [21,25,6,10]. 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 [20,24]. 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 the 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. symbolic execution based approaches [19,21]). 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.

To construct a model of the expected system behavior can, however, 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. We propose to use a model checker to traverse the space of possible valid inputs, in order to generate test inputs. We describe the input model as a nondeterministic program that describes all valid inputs, and then we use the model checker to traverse the state space of this program. We also assert, as the property the model checker should check for, that no such test input exists – this causes the model checker to produce a counterexample whenever a valid test input has been generated and from this counterexample trace we then produce the test input. 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 we use the Java PathFinder model checker (JPF) that analyzes Java programs [26] 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.

The most closely related work to ours is the Korat tool [2] that generates test inputs from Java predicates, but instead of model checking they use a dedicated, efficient, search strategy. The use of the counterexample capability of a model checker to generate test inputs have also been studied by many others (see [18] 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.

2.2 Property Generation

As mentioned earlier, the ideal goal is from a particular test input to generate properties that can be checked on executions of the program on that input. More precisely, assume a particular program that we want to test and the domain *Input* of inputs. Then we have to construct the following objects. First of all, we need to define what is the domain of observable behaviors. We shall regard the executing program as generating an execution trace in the domain *Trace*, where a trace is a sequence of observable events in the domain *Event*. We must define a function:

$$\text{execute} : \mathit{Input} \rightarrow \mathit{Trace}$$

that for a given input returns the execution trace generated by the program when applied to that input. Defining the domain *Event* and the function *execute* in practice amounts to instrumenting the program to log events of importance. The resulting execution trace will then consist of these logged events. Obviously we also need to define the domain *Property* of properties that are relevant for the application and a relation: $\models \subseteq \mathit{Trace} \times \mathit{Property}$ that determines what traces satisfy what properties. We say that $(\sigma, \varphi) \in \models$, also written as $\sigma \models \varphi$, when the trace σ satisfies the property φ. Essentially what is now needed is a function translate:

$$\text{translate} : \mathit{Input} \rightarrow \mathit{Property}\textit{-}\boldsymbol{set}$$

that for a given input returns the set of properties that it is regarded as relevant to test on the execution of the program on that input. A well-behaved program satisfies the following formula:

$$\forall i \in \mathit{Input} \cdot \forall \varphi \in \text{translate}(i) \cdot \text{execute}(i) \models \varphi$$

Our experience is that temporal logic is an appropriate notation for writing properties about the applications we have investigated, and which will be studied in subsequent sections. For a particular application one needs to provide the instrumentation (*execute*) and the property generator (*translate*), which generates a set of temporal logic properties for a particular input. We shall discuss each of these aspects in connection with the case studies.

3 Runtime Analysis

Runtime analysis consists of instrumenting the program and observing the execution of the instrumented program. The runtime analysis modules consist of a code instrumentation module, that augments the program under test with code that generates an event log, and an observer module that evaluates the event log for conformance to desired properties. The event log can be transmitted via inter-process communication or stored as a file. This allows for running an instrumented executable remotely and with little impact on the performance of the system under test.

In our case studies we used two different runtime analyzers: the commercial tool DBRover, based on an extension of the Temporal Rover tool [6] and the research tool JPaX [14,1]. These systems will be described in the following.

The architecture of the JPaX runtime analysis framework is designed to allow several different event interpreters to be easily plugged into the observer component. In the test studies two event interpreters were used: one algorithm analyzes temporal logic properties, as already discussed, and the other concurrency properties. These algorithms are discussed below.

3.1 Instrumentation Framework

Instrumentation takes a program and the properties it is expected to satisfy and produces an instrumented version of the program that when executed generates an event log with the information required for the observer to determine the correctness of the properties. For JPaX we have implemented a very general and powerful instrumentation package for instrumenting Java bytecode.

The requirements of the instrumentation package include power, flexibility, ease of use, portability, and efficiency. We rejected alternative approaches such as instrumenting Java source code, using the debugging interface, and modifying the Java Virtual Machine because they violated one or another of these requirements.

It is essential to minimize the impact of the instrumentation on program execution. This is especially the case for real time applications, applications that may particularly benefit from this approach. Low-impact instrumentation may require careful trades between what is computed locally by the instrumentation and the amount of data that need be transmitted to the observer. The instrumentation package allows such trades to be made by allowing seamless insertion of Java code at any program point.

Code is instrumented based on an *instrument specification* that consists of a collection of predicate-action rules. The predicate is a predicate on the bytecode instructions that are the translation of a Java statement. These predicates are conjunctions of atomic predicates that include predicates that distinguish statement types, presence of method invocations, pattern-matched references to fields and local variables and so on. The actions are specifications describing the inserted instrumentation code. The actions include reporting the program point (method, and source statement number), a time stamp, the executing thread, the statement type, the value of variables or an expression, and invocation of auxiliary methods. Values of primitive types are recorded in the event log, but if the value is an object a unique integer descriptor of the object is recorded.

This has been implemented using Jtrek [5], a Java API that provides lower-level instrumentation functionality. In general, use of bytecode instrumentation, and use of Jtrek in particular, has worked out well, but there are some remaining challenges with respect to instrumenting the concurrency aspects of program execution.

3.2 Observer Framework

As described above, run time analysis is divided into two parts: instrumenting and running the instrumented program, which produces a series of events, and observing these events. The second part, event observation (see Figure 2), can be split into two stages: event analysis, which reads the events and reconstructs the run-time context, and event interpretation. Note that there may be many event interpreters.

To minimize impact on the program under test, log entries contain minimal contextual information and the log entries from different threads are interleaved. Event analysis disentangles the interleaved entries and reconstructs the context from them. The contextual information, transmitted in *internal events*, include thread names, code locations, and reentrant acquisitions of locks (lock counts). The event analysis package maintains a database with the full context of the event log. This allows for writing simpler event interpreters, which can subscribe to particular event types made accessible through an observer interface [9] and which are completely decoupled from each other.

Each event interpreter builds its own model of the event trace, which may consist of dependency graphs or other data structures. It is up to the event interpreter to record all relevant information for keeping a history of the events, since the context maintained by the event analysis changes dynamically with the event evaluation. Any information that needs to be kept for the final output, in addition to the context information, needs to be stored by the event interpreter. If an analysis detects violations of its rules in the model, it can then show the results using the stored data and context information such as code locations, thread names, etc.

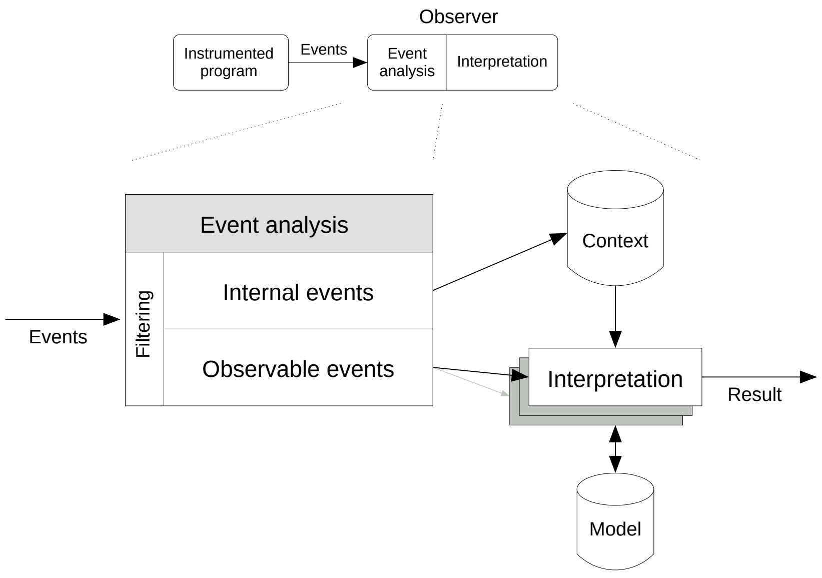

Fig. 2. The observer architecture.

3.3 Temporal Logic Monitoring

Temporal logic in general, and Linear-time Temporal Logic (LTL) in particular, has been investigated for the last twenty years as a specification language for reactive systems [22]. LTL is a propositional logic with the standard connectives $\wedge$, $\vee$, $\rightarrow$ and $\neg$. It furthermore includes four temporal operators: $\Box p$ (always p), $\Diamond p$ (eventually p), $p\,\mathcal{U}\,q$ (p until q – and q has to eventually occur), $\circ p$ (in next step p), and four dual past-time operators (always p in the past, p some time in the past, p since q, and in previous step p). As an example, consider the future time formula $\Box(p \rightarrow \Diamond q)$. It states that it is always the case ($\Box$), that when p holds, then eventually ($\Diamond$) q holds. LTL has the property of being intuitively similar to natural language and capable of describing many interesting properties of reactive systems.

Metric Temporal Logic (MTL) extends LTL so that every temporal operator can be augmented with a relative-time or real-time constraint. Hence, for example, $p\,\mathcal{U}_{c<5}\,q$ means p must be true until a future time, at most 5 c real-time units in the future, where q must hold. Here c is some clock. Similarly $p\,\mathcal{U}_{<5}\,q$ requires q to be true at most 5 cycles in the future, using the underlying state, or cycle based semantics to define the notion of a cycle. As mentioned, we have experimented with two systems that perform temporal logic monitoring: DBRover [6] and JPaX [14,1].

The DBRover Temporal Observer The DBRover is a MTL monitoring tool, based on the TemporalRover code generator [6]. DBRover extends MTL with two forms of parametrization: multi-instancing, which allows for a rule to be independently validated per instance of an object class, process, or thread, and parametrization based on time series data values [7], which enables the verification of properties such as temporal stability and monotonicity. It consists of a GUI for editing temporal assertions, a graphical LTL/MTL simulator, and an execution engine. The DBRover builds and executes temporal rules for a target program or application; at run time, the DBRover listens for event messages from the target application and evaluates corresponding temporal assertions.

The JPaX Temporal Observer With respect to temporal logics, we have implemented several specialized algorithms in JPaX: traversing the execution trace either forwards or backwards, based on either rewriting or automata generation, implemented in either Java or Maude [4]. We next briefly sketch one of these algorithms and refer the interested readers to more elaborated descriptions. Efficiency of runtime analysis algorithms is always an important aspect of our research, even if the observer operates off-line. A crucial observation is that one can design more efficient algorithms if one focusses on segments of temporal logics rather than on the entire logic. Thus, we were able to develop optimal algorithms for future time and for past time temporal logics separately. We do not regard this segmentation as a problem in practice, because in our experience so far one rarely or never uses both future and past time operators in the same requirements formula. The algorithm we are going to describe monitors future time temporal logic formulas and is entirely based on rewriting technology. The idea is to maintain a set of monitoring requirements as future time LTL formulas and modify them accordingly when a new event is emitted by the instrumented program. If one of these formulas ever becomes *false* then it means that that formula has been violated, so an error message is generated and an appropriate action is taken. Four rewriting rules, inspired from known recurrences of temporal operators, transform the formulas whenever a new nonterminal event e is received (and four others not mention here are called on terminal events):

$$\begin{array}{ll}
(\circ F)\{e\} & \to F, \\
(\Box F)\{e\} & \to F\{e\} \wedge \Box F, \\
(\Diamond F)\{e\} & \to F\{e\} \vee \Diamond F, \\
(F\,\mathcal{U}\,F')\{e\} & \to F'\{e\} \vee (F\{e\} \wedge F\,\mathcal{U}\,F')
\end{array}$$

The formula $F\{e\}$, for some formula F, is the (transformed) formula which should hold next, after receiving the event e. For example, for $\Diamond F$ to hold now, where the first event in the remaining trace is e, either F must hold now ($F\{e\}$), or $\Diamond F$ must again hold in the future, thus postponing the obligation. Using memoization (or hashing) techniques provided by advanced rewriting engines such as Maude, the simple rewriting algorithm above performs well in practice. We were for example able to monitor 100 million events in less than 3 minutes on a formula $\Box(g \to (\neg r)\,\mathcal{U}\,y)$ stating a safety policy of a traffic light controller (yellow should come after green). The interested reader is referred to [15,16] for proofs of correctness, complexity analysis and evaluation of this algorithm.

A second approach to building LTL observers based on automata construction is found in [16]. A rewriting-based algorithm for monitoring past time LTL requirements formulas has been presented in [13], which is quite different from the one for future time LTL. A dynamic algorithm approach to monitoring past time LTL formulas is presented in [17].

3.4 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 ensure mutual exclusion to shared objects. That is: to avoid data races where one thread writes to an object while other threads simultaneously either write to or read the same object. Multi-threading programming languages therefore provide constructs for ensuring mutual exclusion. To ensure mutual exclusion on a shared object, a thread can acquire a lock before accessing the object, releasing it after. If other threads acquire the same lock when accessing the object, mutual exclusion is guaranteed. If threads do not acquire the same lock (or don't acquire locks at all) when accessing an object then there is a risk of a data race. The Eraser algorithm [23] can detect such disagreements by analyzing single execution traces. The Eraser algorithm has been implemented in the JPaX tool.

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 whereafter thread T_2 acquires B, where after both threads cannot progress further. In JPaX we have implemented an algorithm for detecting such deadlock situations. It builds a lock graph, where nodes are locks and edges represent the lock hierarchy. That is, for the above example, there will be an edge from A to B and another edge from B to A. Hence for this example the graph contains a cycle, and a cycle represents a potential deadlock situation. This algorithm yields false positives (false warnings) and false negatives (missed deadlocks). In [1] an extension to this algorithm is described that reduces the number of false positives. JPaX's concurrency analysis has been integrated with the DBRover, where deadlock results are graphically displayed as UML message sequence charts.

4 Case Study 1: A Planetary Rover Controller

Our first case study 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 [3]. The executive receives plans of actions that the rover is requested to carry out, and executes these plans. First we present a description of the system, including a description of what plans (the input

```
Plan      → Node

Node      → Block | Task

Block     → (block
             NodeAttr
             :node-list (NodeList))

NodeList  → Node NodeList | ϵ

Task      → (task
             NodeAttr
             :action Symbol
             [:fail]
             [: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
    :fail
   )
  )
)
```

Fig. 3. Plan grammar (left) and an example of a plan (right).

domain) look like. Then we outline how plans (test inputs) can be automatically generated using model checking, and finally we describe how for each plan a set of temporal logic properties can be automatically generated, that the executive must satisfy when executing the plan.

4.1 System Description

The executive is a multi-threaded system (8,000 lines of Java 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. We give here a short presentation of the (simplified) language used in the description of the plans that the rover executive must execute.

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 3 (left) shows the grammar for the language; we should note that all the 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 the original plan syntax to facilitate testing of the executive. That is, during testing using test case generation, the real actions are never executed since this would require operating the rover mechanics. The `:fail` and `:duration` attributes replace the actions during testing. The *fail* flag for a task specifies the action status after execution; the *duration* specifies the duration of the action. Figure 3 (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 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, 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 action's status indicates failure, we have a task failure. The execution of a *block* simply proceeds by executing each of the nodes in the node-list in order. (3) If the time exceeds the end condition, the node fails.

On a *node failure*, when execution returns to the sequence, the value of the failed node's *continue-on-failure* flag 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 4 shows the Java code, referred to as the *universal planner*, that we used to generate plans (i.e., test inputs for the executive). We exploit nondeterminism (i.e., `choose` methods) to systematically generate all the possible plans (up to a given number of nodes specified by `nNodes`) that have the structure specified by the grammar in Figure 3; we also use nondeterminism to generate the data values for the time conditions (up to a given value specified by `tRange`). 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). We used the JPF model checker to explore the (finite) state space of the generated input plans (i.e., JPF model checks the universal planner). We used different search strategies to

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

  static void Plan(int nn, int tr) {
    nNodes = nn; tRange = tr;
    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() {
    UniversalAttributes();
    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(){
    UniversalAttributes();
    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();
    int time1 = choose(tRange);
    int time2 = time1 + choose(tRange);
    start = new TimeCondition(time1,time2);
    time1 = choose(tRange);
    time2 = time1 + choose(tRange);
    end = new TimeCondition(time1,time2);
    continueOnFailure = chooseBool();
  }
}
```

Fig. 4. Universal planner that generates input plans for system under test.

find multiple counterexamples (to the assertion); for each counterexample we ran JPF in simulation mode to `print` the generated plans to a file, which then served as input to the rover.

4.3 System Analysis

The semantics of a particular plan can very naturally be formulated in temporal logic. In writing such properties, we used the following predicates: 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); end(id) denotes success$(id) \vee$ fail(id). We instrumented the code to monitor these predicates. For each plan we further wrote a collection of temporal properties over these predicates and verified their validity on execution traces. As an example, the properties for the plan shown in Figure 3 (right) are shown in Figure 5. This set of properties does not fully represent the semantics of the plan, but the approach appeared to be sufficient to catch a large amount of bugs.

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

Fig. 5. Temporal logic properties representing partial semantics of plan in Figure 3 .

The purpose of the case study was to find a number of seeded errors in the rover by using a number of different technologies, including runtime analysis, model checking, static analysis and traditional testing. Here we just focus on the results on the runtime analysis which was done according to the framework described in this paper. We used the DBRover to monitor the temporal properties that each plan had to satisfy. We generated the formulas for each plan by hand, in a similar fashion as the formulas given in Figure 5. Using this approach most of the errors pertaining to plan semantics were easily discovered. It is worth noting that none of the other techniques found as many of the plan semantic errors as the temporal logic runtime analysis technique. It is possible to generate these properties automatically.

The rest of the errors were related to concurrency and for those JPaX's concurrency analysis was used (see Section 3.4). All but two of the concurrency errors were discovered this way. JPaX found all the seeded data races and all the seeded cyclic deadlocks (resource deadlocks), which are the errors this tool can find. The errors it could not find were communication deadlocks involving Java's **wait** and **notify** constructs. These errors were very subtle and required analysis beyond what JPaX could do at the time. Note that this supports our hypothesis that our framework is good at finding a large class of errors, but may not be as good in finding some subtle errors – for this more advanced techniques may be required, such as model checking (that did find all the concurrency errors in the rover).

5 Case Study 2: A Space Craft Fault Protection System

In this section we describe the ongoing application of the framework in a verification experiment targeted to a mission-critical NASA application called a fault protection system (FPS). A fault protection system monitors critical hardware and software components of a spacecraft and executes corrective responses to detected faults. Our approach to this case study is to:

1. write a small program to generate test cases for the FPS,
2. write a partial specification (of an instantiation) of the FPS in the form of LTL (linear temporal logic) assertions,
3. instrument the FPS to emit a log that allows checking of the LTL assertions,
4. write a test driver which runs each generated test, and invokes the LTL observer to analyze the generated log,
5. and finally, the observer reports those test cases and assertion combinations that failed or are inconclusive.

5.1 System Description

The FPS system we analyze consists of a reusable core engine and a portion customizable for a specific spacecraft. The FPS is a complex, logic-oriented system. That is, the code is dense with decision points yielding a very large number of potential control paths. Given the complexity, mission criticality, and reusability of this code, a high level of reliability is demanded. The FPS system has been the study of a model checking experiment reported in [8]. The FPS was originally written in C and has flown on spacecraft. A new implementation in Java has recently been developed. This implementation was provided with a wrapper that allows it to run stand alone using a simple command interface. We based our experiments on this Java implementation.

One of the commands informs the FPS of spacecraft problem symptoms. The FPS maintains a customizable mapping between symptoms, faults, and responses. The basic mode of execution is to respond to a symptom by mapping the symptom to a fault, the fault to a response and then execute the response. The FPS executes one response at a time. However many symptoms may have been reported requiring that responses be queued. In addition, responses have different priority levels; some responses can be interrupted; and some responses "call" other responses as subroutines. In addition, responses, symptoms, and faults, may be in various modes, such as enabled or disabled, and there are commands to reset responses and the whole fault protection system. Furthermore, the FPS must respond to ground commands to execute a response in a manner that differs from responses invoked by fault protection itself. It is these capabilities and others that contribute to the significant complexity of the FPS.

The FPS core engine is specified as finite state machines that signal and invoke each other by generating events. There are state machines for the FPS engine itself, instances of a state machine for each symptom, fault and response. In fact there are two state machines for each response. One is part of the engine itself – it records the status of the response, and a second encodes the semantics of the responses themselves (e.g. turn on a device). Code is automatically generated for the latter response state machines and become part of the code base for an instantiation of the FPS. In fact we analyzed a particular instantiation of FPS with two responses.

5.2 Test Input Generation

For this application a simple program (less than 200 lines of Python code) was written to generate test cases. A test case for the FPS is simply a sequence of commands. The generator generated commands sequences of a specified length by making independent random selections of individual commands. The generator is parameterized by:

- the number of test cases to be generated,
- the number of commands in a test case,
- the relative frequency of each command,
- for each command, the relative frequency of its arguments.

There is subtlety to this process. It should be noted that the execution of a response takes time. Generally in one time unit a response performs the action described in a single state of its state machine and then transitions to a new state. The signalling of a new time step corresponds to the execution of a command (or signal) to advance the clock. Thus the relative frequencies of the commands need to be selected such that response queues do not overflow, yet there would be enough pending responses to test the behavior when multiple faults required a response. We thank Owen O'Malley of NASA for writing the test case generator.

5.3 System Analysis

We used the JPaX temporal observer to check LTL formulas against the execution traces generated for each test execution. We used a partial static LTL specification for the FPS. As such it is not specific to any single input as is the case for the K9 experiment. It is based on the code itself, design documents describing the state machines of the FPS, and some test data and expected results. Clearly, the purpose of the specification is to describe the intended semantics of the program independently of the program itself, and so using the program is less than ideal. However we found in places that the code itself exhibits inconsistencies and resolving these was essential to specification creation. In fact, it was by this process that some bugs were found.

LTL is a very natural notation for describing the FPS. The FPS is a transaction system. Its top level functionality is as a command processor. It reads commands and executes them, potentially queuing actions. Characterizing the state of the FPS (e.g., as the state of each state machine and the value of certain variables) and then characterizing the action over time of each command on the state is a natural methodology for developing such a specification. In addition, properties asserting that each action taken is in response to a relevant command will insure that the program does only what is intended to do and nothing more. One subtlety of the process is the mapping of specification level concepts (e.g. a "response completes") with a suitable code-level event. There may be many candidate predicates or events at the code level that correspond to a specification level predicate or event. These code level predicates may differ in subtle ways and may or may not correctly model intended semantics. We found it most effective to relate predicates and events closely to the inputs and output products of the program.

These experiments are still in progress. All of the components described are in place and executing. The additional work is to strengthen the LTL specification. However, to date we have already identified program bugs and inaccuracies in the specifications. One bug is significant in that it existed in the C version that flew on spacecraft.

This case study varies from the proposed framework in two respects. First we did not use a model checker to generate test cases. The model checking approach would work fine, and would enjoy the benefits described above. Second the defined properties are universal correctness properties, valid for all inputs. The framework suggests defining a function that maps an input to a set of properties specific to that input. In this case it was straightforward to define universal properties and it is not evident that the alternative approach makes property specification any easier. Note that the specification was constructed by formalizing the semantics of each type of command, taking into account the computation state, independently of the other commands. If this approach is followed then knowing the other commands in the sequence would not be of much help in simplifying the property set.

6 Conclusions and Future Work

We have presented a framework for testing based on automated test case generation and runtime analysis. 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. We see the proposed framework as a complementary approach to testing that may be applied opportunistically to test selected system behaviors. We proposed and demonstrated the use of model checking for test case generation. We will be exploring how to improve the quality of the generated test suite by altering the model checker's search strategy, and by use of symbolic execution.

In the near future, we will continue the development of a complete testing environment for the K9 rover and seek to get this technology transferred to NASA engineers. We are continuing the work on instrumentation of Java bytecode. In a parallel effort with Robert Filman, one of our NASA Ames (RIACS) colleagues, a source code instrumentation tool is being developed that is based on ideas in Aspect Oriented Programming. Concerning the observer part, we have presented various interpreters for temporal logic. Current and future work is devoted to the design of a specification language that is attractive to engineers and powerful enough to capture a majority of practical system requirements. We are also investigating new algorithms for concurrency analysis that extend the scope of deadlocks and data races that can be identified.

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. S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Submitted for publication, October 2002.
2. 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.
3. 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.
4. M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic, March 1999. Maude System documentation at `maude.csl.sri.com/papers`.
5. S. Cohen. Jtrek. Compaq, `www.compaq.com/java/download/jtrek`.
6. 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.
7. D. Drusinsky. Monitoring Temporal Rules with Temporal Data. Submitted for publication, October 2002.
8. M. Feather, S. Fickas, and N. Razermera-Mamy. Model-Checking for Validation of a Fault Protection System. In *Proceedings of Sixth IEEE International Symposium on High Assurance System Engineering.* IEEE Computer Society, October 2001.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
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. `www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf`.

13. K. Havelund, S. Johnson, and G. Roşu. Specification and Error Pattern Based Program Monitoring. In *Proceedings of the European Space Agency workshop on On-Board Autonomy*, Noordwijk, The Netherlands, October 2001.
14. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.
15. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings of the International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. IEEE CS Press, 2001. Coronado Island, California.
16. K. Havelund and G. Roşu. A Rewriting-based Approach to Trace Analysis. Submitted for journal publication, September 2002.
17. K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. EASST best paper award at ETAPS'02.
18. 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.
19. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
20. B. Korel. Automated Software Test Data Generation. *IEEE Transaction on Software Engineering*, 16(8):870–879, August 1990.
21. Parasoft. `www.parasoft.com`.
22. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
23. 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.
24. 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.
25. T-VEC. `www.t-vec.com`.
26. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.