Automated Testing using Symbolic Execution and Temporal Monitoring

Cyrille Artho [a], Allen Goldberg [b], Klaus Havelund [b], Sarfraz Khurshid [c], Mike Lowry [d], Corina Pasareanu [b], Grigore Roşu [e], Willem Visser [f], Rich Washington [f]

[a] *Computer Systems Institute, ETH Zurich, Switzerland*

[b] *Kestrel Technology, NASA Ames Research Center, USA*

[c] *MIT Computer Science and Artificial Intelligence Laboratory, USA*

[d] *NASA Ames Research Center, USA*

[e] *Department of Computer Science, Univ. of Illinois at Urbana-Champaign, USA*

[f] *RIACS, NASA Ames Research Center, USA*

Abstract

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

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

1 Introduction

A program is typically tested by manually creating a *test suite*, which in turn is a set of *test cases*. An individual test case is a description of a *test input sequence* to the program, and a description of *properties* that the corresponding output is expected to have. This manual procedure may be unavoidable since for real systems, writing test cases is an inherently innovative process requiring human insight into the logic of the application being tested. Discussions with robotics and space craft engineers at NASA seems to support this view. However, an equally widespread opinion is that a non-trivial part of the testing work *can* be automated. In a case study, an 8,000-line Java application was tested by different student groups using different testing techniques [5]. It is conjectured that the vast majority of bugs in this system that were found by the students could have been found in a fully automatic way. The paper presents work on applying low-overhead automated testing to identify bugs quickly. We suggest a framework for generating test cases in a fully automatic way as illustrated by Figure 1. For a particular program to be tested, one establishes a test harness consisting of four modules: a *test input generator module*, a *property generator module*, a *program instrumentation module* and an *observer module*.

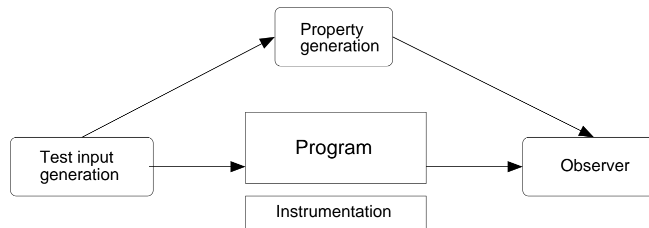

Figure 1. Test case generation (test input generation and property generation) and runtime analysis (instrumentation and observation).

The test input generator automatically generates inputs to the application, one by one. A generated input is fed to the the property generator, which automatically generates a set of properties that the program should satisfy when executed on that input. The input is then fed to the program, which executes, generating an execution trace. The observer module "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 some cases be automated. The test input generator and the property generator are both written ("hard-wired") specifically for the application that is tested. This replaces manual construction of test cases. However, the observer module is generic and can be re-used on different applications. In the rest of this paper

the term *test case generation* is used to refer to test input generation and property generation, and the term *runtime analysis* to refer to instrumentation as well as observation.

The framework described above has been applied to a case study, a planetary rover controller. Test cases are generated using a model checker and the properties generated are *specific to a single test case*. Properties are expressed in temporal logic. The approach of generating properties specific to a single test case is novel.

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

2 Test Case Generation

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

2.1 Test Input Generation

2.1.1 Model based testing

In practice today, the generation of test inputs for a program under test is a time-consuming and mostly manual activity. However, test input generation naturally lends itself to automation, and therefore has been the focus of much research attention – recently it has also been adopted in industry [28,35,9,12]. 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 [24,34]. 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 [14]). 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 [14] – even the code itself can be used to create a model (e.g. symbolic execution based approaches [23,28]). As with the dynamic approach, it is most typical to use some notion of coverage of the model to derive test inputs, i.e., generate inputs that cover all transitions (or branches, etc.) in the model.

On the one hand, constructing a model of the expected system behavior can be a costly process. On the other hand, generating test inputs just based on a specification of the input structure and input pre-conditions can be very effective, while typically less costly. In [22] we present a verification framework that combines *symbolic execution* and model checking techniques in a novel way. The framework can be used for test input generation as follows: the input model is described as a non-deterministic program annotated with constraints that describes all valid inputs, and the model checker is used to traverse the (symbolic) state space of this program. As the property the model checker should check for, one asserts that no such test input exists – this causes the model checker to produce a counter-example whenever a valid test input has been generated, and from this counter-example 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 [36] and supports various heuristic search strategies (for example, based on branch coverage [13] or random search). In Section 4.2 we show how this model checker is used to generate test inputs for the Mars K9 rover.

Using symbolic execution for test case generation is a well-known approach, but typically only handles sequential code with simple data. In previous work, this technique has been extended to handle complex data-structures (e.g. lists and trees), concurrency as well as linear constraints on integer data [22]. Symbolic execution of a program path results in a set of constraints that define program inputs that execute the path; these constraints can often be solved using off-the-shelf decision procedures to generate concrete test inputs. When the program represents an executable specification, symbolic execution of the specification enables us to generate inputs that give us, for instance, full specification coverage. Note that these specifications are typically not very large – no more than a few thousand lines, in our experience – and hence will allow efficient symbolic execution.

The most closely related work to ours is the Korat tool [4] that generates test inputs from Java predicates, but instead of model checking they use a dedicated, efficient, search strategy. The use of the counter-example capability

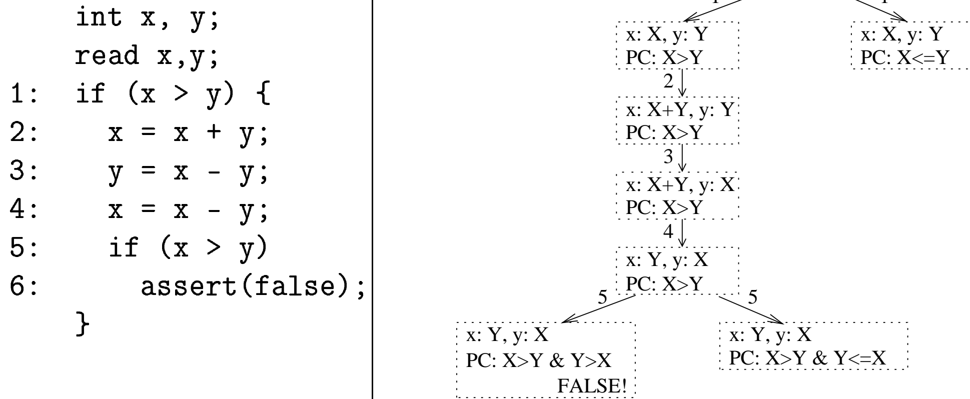

Figure 2. Code for swapping integers and corresponding symbolic execution tree.

of a model checker to generate test inputs have also been studied by many others (see [20] for a good survey), but most of these are based on a full system model, not just the input structure and pre-conditions as suggested here.

2.1.2 Symbolic Execution for Test Input Generation

The enabling technology for black-box test-input generation from an input specification is the use of symbolic execution. Optionally the system under test itself can be symbolically executed, for white-box testing, the techniques are in fact the same. The main idea behind symbolic execution [23] 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 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 [22]) the code fragment in Figure 2, which swaps the values of integer variables `x` and `y`, when `x` is greater than `y`. Figure 2 also shows the corresponding symbolic execution tree. Initially, the path condition, `PC`, is *true* and `x` and `y` have symbolic values `X` and `Y`, respectively. At each branch point, `PC` is updated with assumptions about the inputs according to the alternative (possible) paths. For example, after the execution of the first statement, both `then` and `else` alternatives of the `if` statement are possible, and `PC` is updated accordingly. If the path condition becomes *false*, i.e., there is no set of inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for

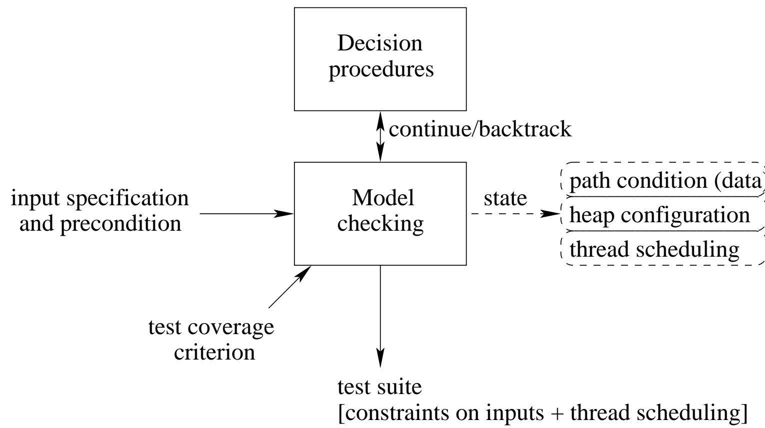

Figure 3. Framework for test input generation.

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 the inputs `x` and `y`, respectively 1 and 0.

Symbolic execution traditionally arose in the context of sequential programs with a fixed number of integer variables. We have extended this technique to handle dynamically allocated data structures (e.g. lists and trees), complex preconditions (e.g. only acyclic lists), other primitive data (e.g. strings) and concurrency. A key feature of our algorithm is that it starts the symbolic execution of a procedure on *uninitialized* inputs and it uses *lazy initialization* to assign values to these inputs, i.e., it initializes parameters when they are first accessed during the procedure's symbolic execution. This allows symbolic execution of procedures without requiring an a priori bound on the number of input objects. Procedure preconditions are used to initialize inputs only with valid values.

As mentioned before our symbolic execution-based framework is built on top of the Java PathFinder (JPF) model checker [36]. JPF is an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). It can handle all of the language features of Java, and in addition it treats non-deterministic choice expressed in annotations of the program being analyzed. For symbolic execution the model checker was extended to allow backtracking whenever a path-condition is unsatisfiable (determined by calling a decision procedure).

2.1.3 Framework for Test Input Generation

Figure 3 illustrates our framework for test input generation. The input specification is given as a non-deterministic Java program that is instrumented to add support for manipulating formulas that represent path conditions. The instrumentation allows JPF to perform symbolic execution. Essentially, the

model checker explores the (symbolic) state space of the program (for example, the symbolic execution tree in Figure 2). A symbolic state includes a heap configuration, a path condition on integer variables, and thread scheduling information. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure; currently our system uses the Omega library [30] that manipulates linear integer constraints. If the path condition is unsatisfiable, the model checker backtracks. A testing coverage criterion is encoded in the property the model checker should check for. This causes the model checker to produce a counter-example whenever a valid (symbolic) test input has been generated and from this trace we produce a (concrete) test input. Since only input variables are allowed to be symbolic, all constraints that are part of a counter-example are described in terms of inputs, and finding a solution to these constraints will allow a valid set of test data to be produced. Currently a simple approach is used to find these solutions. Only the first solution is considered. In future work we will refine the solution discovery process to also consider characteristics such as boundary cases.

Currently, the model checker is not required to perform state matching, since state matching is, in general, undecidable when states represent path conditions on unbounded data. It is also important that performing symbolic execution on programs with loops can explore infinite execution trees (and it might not terminate). Therefore, for systematic state space exploration, limited depth-first search or breadth-first search is used; our framework also supports heuristic-based search [13].

2.2 Property Generation

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

More precisely, the artifact produced here is a program that takes as input an input to a program and generates a set of properties, typically assertions in

linear temporal logic. The assertions are then checked against each program execution using the runtime analysis tools described in Section 3. For the case study presented in Section 4, writing this program was straightforward, and considerably easier than writing a single set of properties relevant to *all* inputs.

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

3 Runtime Analysis

Runtime analysis consists of observing the execution of a program, which in turn requires capturing relevant execution events through *event extraction*. The events generated by the program are evaluated by an *observer* for conformance to desired properties. Events can be transmitted via inter-process communication or stored as a file. This allows for running the observer remotely and with little impact on the performance of the system under test. In our case study, the research tool Java PathExplorer (JPaX) was used [16]. 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 experiment, two event interpreters were used: one algorithm analyzes temporal logic properties, and the other one checks concurrency properties. These algorithms are discussed below.

3.1 Event Extraction

The event extraction can be achieved in a number of ways, including *wrapping* and *instrumentation*. In a *wrapping* approach, the standard execution environment is replaced with a customized one that allows observation by wrapping system libraries. This is the approach of Purify [31]. In the *instrumentation* approach, source code (or object code) is augmented with code that generates the event stream. Our experiments use both approaches. The instrumentation approach is used to generate events for the temporal logic monitoring. As will

be explained, this monitoring examines events indicating the start and end of task executions, and the code has been manually instrumented to generate these events. The wrapping approach is used to generate events for the concurrency analysis, where lock acquisitions and lock releases are monitored. These events are generated by wrapping method calls around POSIX [27] thread methods, and then instrument the wrappers.

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

Such an instrumentation package, named jSpy, has been implemented for instrumenting Java bytecode [11]. This was first implemented using Jtrek [7], 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. However, at the time of writing, a new version is being implemented based on the BCEL library [8], which is Open Source, unlike Jtrek, which has been discontinued.

3.2 Observer Framework

As described above, runtime analysis is divided into two parts: instrumentation and execution of the instrumented program. To minimize the impact on the program under test, events should contain minimal information. Two categories of information need to be transmitted: *essential* information, needed to detect property violations, and *contextual* information, needed to print out informative error messages when properties get violated. Instrumentation is done such that contextual information is sent only when it changes, and not with every event. The event observer, (see Figure 4), can be correspondingly be split into two stages. The *event dispatcher* reads the events and sends a reconstructed version to one or more *property analyzers.*

The event dispatcher parses events, and converts them into a unified format, accommodating different instrumentation packages. The contextual information, transmitted in *internal events* (not emitted to the property analyzers),

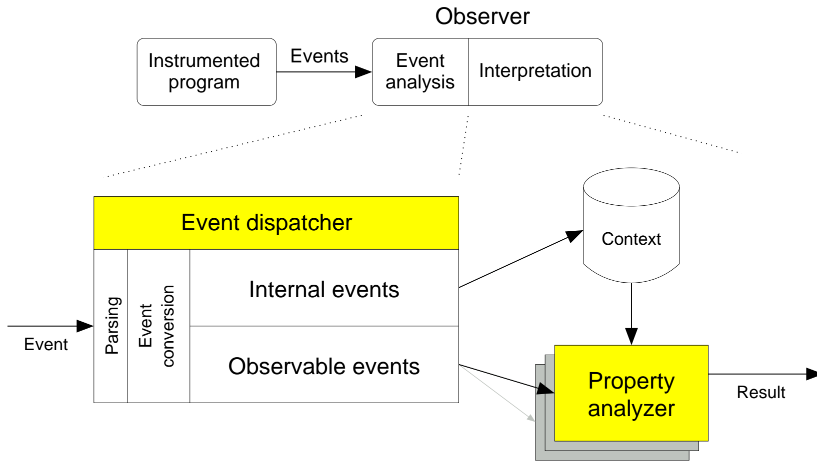

Figure 4. The observer architecture.

include thread names, code locations, and reentrant acquisitions of locks (lock counts). The event dispatcher package maintains a database with the full context of the events. This allows for writing simpler property analyzers. The property analyzers subscribe to particular event types made accessible through an observer interface [10] and are completely decoupled from each other.

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

3.3 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 [29]. LTL is an extension of propositional logic, which contains the standard connectives $\wedge$, $\vee$, $\rightarrow$ and $\neg$, with 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.

With respect to temporal logics, several specialized algorithms are imple-

mented in JPaX: traversing the execution trace either forward or backward, based on either rewriting or automata generation, implemented in either Java or Maude [6]. One of these algorithms will be briefly sketched. The interested reader is referred to the bibliography for more elaborate descriptions.

Efficiency of runtime analysis algorithms is 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 focuses on segments of temporal logics rather than on the entire logic. This observation allowed us to develop optimal algorithms for future-time and for past-time temporal logics separately. This segmentation does not arise 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 described here 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 mentioned here, are called on terminal events – terminating a trace):

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

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. In addition, a rewriting based boolean simplification procedure, due to Hsiang [21] and based on the Boolean ring simplification, is used on-the-fly to keep the formula in a canonical compact form, more precisely as an exclusive disjunction of conjunctions. The following theorem claims that the just presented very simple and succinct rewriting-based monitoring algorithm above is optimal:

Theorem 1 *For any formula F of size m and any sequence of events to be monitored e_1, e_2, ..., e_n, the formula $F\{e_1\}\{e_2\}...\{e_n\}$ needs $O(2^m)$ space to be stored. Moreover, the exponential space cannot be avoided: any monitoring al-*

gorithm for LTL requires space $\Omega(2^{c\sqrt{m}})$ space, where c is some fixed constant. It can be shown that the lower bound can be further refined to $\Omega(2^m)$.

Proof As mentioned above, due to the Boolean ring simplification rules, any LTL formula is kept in a canonical form, which is an exclusive disjunction of conjunctions. Each conjunct is either a proposition or otherwise it has a temporal operator at the top. Moreover, after processing any number of events e_1, e_2, ..., e_n, the conjuncts in the normal form of $F\{e_1\}\{e_2\}...\{e_n\}$ are sub-terms of the initial formula `F`, each being a proposition or otherwise having a temporal operator at its top. Since there are at most m such sub-formulas of F, it follows that there are at most 2^m possibilities to combine them in a conjunction. Therefore, one needs space $O(2^m)$ to store any exclusive disjunction of such conjunctions. This reasoning only applies to "idealistic" rewriting engines, which carefully optimize space needs during rewriting. Since the implementation details of Maude are not public, it is not clear to us whether Maude is able to attain this space upper bound in all practical situations. However, space or time resources were never a problem in our practical experiments.

For the space lower bound of any finite trace LTL monitoring algorithm, consider a simplified framework with only two atomic predicates and therefore only four possible states. For simplicity, these four states are encoded by 0, 1, # and \$. Consider also some natural number k and the language:

$$L_k = \{\sigma\#w\#\sigma'\$w \mid w \in \{0,1\}^k \text{ and } \sigma, \sigma' \in \{0,1,\#\}^*\}.$$

This language was previously used in several works [25,26,32] to prove lower bounds. The language can be shown to contain exactly those finite traces satisfying the following LTL formula [26] of size $\Theta(k^2)$:

$$\begin{aligned}\phi_k = {} & [(\neg\$)\ \mathcal{U}\ (\$\ \wedge\ \circ\Box(\neg\$))]\ \wedge \\ & \Diamond[\#\ \wedge\ \circ^{k+1}\#\ \wedge\ \textstyle\bigwedge_{i=1}^{k}((\circ^i 0\ \wedge\ \Box(\$ \rightarrow \circ^i 0)) \vee\ (\circ^i 1\ \wedge\ \Box(\$ \rightarrow \circ^i 1)))].\end{aligned}$$

Let us define an equivalence relation on finite traces in $(0+1+\#)^\star$. For a $\sigma \in (0+1+\#)^\star$, define $S(\sigma) = \{w \in (0+1)^k \mid \exists\lambda_1, \lambda_2.\ \lambda_1\#w\#\lambda_2 = \sigma\}$. $\sigma_1 \equiv_k \sigma_2$ if and only if $S(\sigma_1) = S(\sigma_2)$. Now observe that the number of equivalence classes of $\equiv_k$ is 2^{2^k}; this is because for any $S \subseteq (0+1)^k$, there is a σ such that $S(\sigma) = S$.

Since $|\phi_k| = \Theta(k^2)$, it follows that there is some constant c' such that $|\phi_k| \leq c'k^2$ for all large enough k. Let c be the constant $1/\sqrt{c'}$. This lower bound result

can be proven by contradiction. Suppose $\mathcal{A}$ is an LTL forward monitoring algorithm that uses less that $2^{c\sqrt{m}}$ space for any LTL formulas of large enough size m. Consider the behavior of the algorithm $\mathcal{A}$ on inputs of the form ϕ_k. So $m = |\phi_k| \leq c'k^2$, and $\mathcal{A}$ uses less than 2^k space. Since the number of equivalence classes of $\equiv_k$ is 2^{2^k}, by the pigeon hole principle there must be two strings $\sigma_1 \not\equiv_k \sigma_2$ such that the memory of $\mathcal{A}$ on ϕ_k after reading $\sigma_1\$$ is the same as the memory after reading $\sigma_2\$$. In other words, $\mathcal{A}$ running on ϕ_k will give the same answer on all traces of the form $\sigma_1\$w$ and $\sigma_2\$w$. Now since $\sigma_1 \not\equiv_k \sigma_2$, it follows that $(S(\sigma_1) \setminus S(\sigma_2) \cup (S(\sigma_2) \setminus S(\sigma_1)) \neq \emptyset$. Take $w \in (S(\sigma_1) \setminus S(\sigma_2) \cup (S(\sigma_2) \setminus S(\sigma_1))$. Then clearly, exactly one out of $\sigma_1\$w$ and $\sigma_2\$w$ is in L_k, and so $\mathcal{A}$ running on ϕ_k gives the wrong answer on one of these inputs. Therefore, $\mathcal{A}$ is not correct. $\square$

Using memoization (or hashing) techniques provided by advanced rewriting engines such as Maude, the simple rewriting algorithm above performs quite well in practice. It was able to monitor 100 million events in less than 3 minutes on a formula $\Box(g \rightarrow (\neg r) \;\texttt{U}\; y)$ stating a safety policy of a traffic light controller (yellow should come after green). The interested reader is referred to [17,18] 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 [18]. A rewriting-based algorithm for monitoring past-time LTL requirements formulas has been presented in [15], which is quite different from the one for future-time LTL. A dynamic programming approach to monitoring past-time LTL formulas is presented in [19]. Recently, a new rule-based framework for runtime monitoring, named Eagle, has been developed [2]. This framework is very powerful in allowing the user to define a custom temporal logic using simple equational definitions. Since rules can be parameterized with data values, as well as with formulas, the framework allows to define logics over data, with real-time constraints being a special case.

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. The goal is to avoid data races where one thread writes to an object while other threads simultaneously either write to or read from the same object. Multi-threading programming languages therefore provide constructs for ensuring mutual exclusion, usually in the form of locks. If other threads utilize the same lock when accessing an object, mutual exclusion is guaranteed. If

threads do not acquire the same lock (or do not acquire locks at all) when accessing an object then there is a risk of a data race. The Eraser algorithm [33] can detect such disagreements by analyzing single execution traces. The Eraser algorithm has been implemented in the JPaX tool. Recent work has shown that another kind of error, high-level data races, can still be present in programs that use mutual exclusion for accessing individual fields, but not sets of fields, correctly [1].

Deadlocks can occur when two or more threads acquire locks in a cyclic manner. As an example of such a situation consider two threads T_1 and T_2 both acquiring locks A and B. Thread T_1 acquires first A and then B before releasing A. Thread T_2 acquires B and then A before releasing B. This situation poses a deadlock situation since thread T_1 can acquire A where after thread T_2 acquires B, where after both threads cannot progress further. JPaX includes such a deadlock detection algorithm. 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). An extension to this algorithm reduces the number of false positives [3].

4 Case Study: A Planetary Rover Controller

The case study described here is the planetary rover controller K9, and in particular its *executive* subsystem, developed at NASA Ames Research Center – a full account of this case study is described in [5]. The executive receives plans of actions that the rover is requested to carry out, and executes these plans. First a description of the system is presented, including a description of what plans (the input domain) look like. Then it is outlined how plans (test inputs) can be automatically generated using model checking. Finally it is described how, for each plan, one can automatically generate a set of temporal logic properties that the executive must satisfy when executing the plan.

4.1 System Description

The executive is a multi-threaded system (35,000 lines of C++ code) that receives flexible plans from a planner, which it executes according to a plan language semantics. A plan is a hierarchical structure of actions that the rover must perform. Traditionally, plans are deterministic sequences of actions. How-

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
) ) )
```

Figure 5. Plan grammar (left) and an example of a plan (right).

ever, 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 must execute.

4.1.1 Plan Syntax

A plan is a *node*; a node is either a *task*, corresponding to an *action* to be executed, or a *block*, corresponding to a logical group of nodes. Figure 5 (left) shows the grammar for the language. All node attributes, with the exception of the node's *id*, are optional. Each node may specify a set of *conditions*, e.g. the *start condition* (that must be true at the beginning of the node execution) and the *end condition* (that must be true at the end of the node execution). Each condition includes information about a relative or absolute time window, indicating a lower and an upper bound on the time. The *continue-on-failure* flag indicates what the behavior will be when node failure is encountered.

The attributes *fail* and *duration* were added to 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 5 (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).

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

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

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

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

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

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

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

Figure 6. Code that generates input plans for system under test.

4.1.2 Plan Semantics

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

Figure 6 shows part of the Java code, referred to as the *universal planner*, that is used to generate plans (i.e., test inputs for the executive). The framework suggested in Section 2 is used where an annotated Java program specifies only the structure of the inputs together with the preconditions on this structure. Model checking with symbolic execution generates the inputs. In order to specify the structure non-deterministic choice (`choose` methods) are exploited over all structures allowed in the grammar presented in Figure 5, and preconditions are specified as constraints over some of the integer variables in the structure. For the latter, only time points are considered. Furthermore, these represent *inputs* to our specification, to allow symbolic execution and constraint solving to generate valid test cases. For brevity, only a small sample set of constraints is shown here (stating that the end time is larger than the start time of an interval). In the full version there are constraints relating different time-points, corner cases where start-times are given after end times, etc.

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

```
class Symbolic { ...
  static PathCondition _Path_cond;
  Symbolic _plus(Symbolic e) { ... }
  Symbolic _minus(Symbolic e) { ... }
  int solution() { ... }
}
```

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

Figure 7. Library classes for symbolic execution.

Figure 7 gives part of the library classes that provide symbolic execution. Class `Symbolic` implements all symbolic constraints and has (amongst others) a subclass `SymInt` that represents symbolic integer values. The `static` field `Symbolic._Path_cond` stores the (numeric) path condition. Method `_add_GT` updates the path condition with the *greater-than* constraint. Method `is_satisfiable` uses the Omega library to check if the path condition is infeasible (in which case, JPF will backtrack). The `solution` method first solves the constraints and then returns one solution value for a symbolic integer (solutions are currently not defined for non-integer symbolic values).

- $\Diamond$start(*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).
- $\Diamond$success(*drive1*), i.e., task `drive1` should end successfully (since `:duration` is within time window).
- $\Diamond$fail(*drive2*), i.e., task `drive2` should fail (due to `:fail`).

Figure 8. Temporal logic properties representing partial semantics of plan in Fig. 5.

4.3 System Analysis

The semantics of a particular plan can very naturally be formulated in temporal logic. In writing such properties, the following predicates were used: start(*id*) (true immediately after the start of the execution of the node with the corresponding *id*), success(*id*) (true when the execution of the node ends successfully), fail(*id*) (true when the execution of the node ends with a failure), and end(*id*), which denotes success(*id*) $\vee$ fail(*id*). We instrumented the code to monitor these predicates. For each plan we further automatically generated 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 5 (right) are shown in Figure 8. This set of properties does not fully represent the semantics of the plan, but the approach appears to be sufficient to catch errors.

The runtime analysis identified a number of errors in the executive. A preliminary, partially automated system for runtime testing found a deadlock and a data race. For the deadlock, the additional instrumentation triggered the deadlock during execution, but in fact the pattern existed in the un-instrumented version of the executive, and would have been identified by the instrumentation, even if it had not occurred explicitly. The data race, involving access to a shared variable used to communicate between threads, was suspected by the developers, but had not been confirmed in code. The trace analysis allowed the developers to see the read/write pattern clearly and redesign the communication. The fully automated testing system detected a bug that had been

seeded in the code for verification purposes: the bug produced an execution failure when a plan node was processed after the beginning of its start window. Finally, the automated testing system found a missing feature that had been overlooked by the developers: the lower bounds on execution duration were not enforced, so the temporal logic model predicted failure when execution in fact succeeded. This latter error was unknown to the developers, and it showed up later during actual rover operation before it was corrected.

5 Conclusions and Future Work

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

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

In the near future, we will continue the development of a complete testing environment for the K9 rover executive, and seek to get this technology transferred to NASA engineers. We will be exploring how to improve the quality of the generated test suite by altering the search strategy of the model checker, and by improving the symbolic execution technology. We will also investigate the use of real-time logic and other more complicated logics. In particular, the Eagle logic should provide a good framework for monitoring. We are continuing the work on instrumentation of Java bytecode and will extend this work to C and C++. Our research group has done fundamental research in other areas, such as software model checking (model checking the application itself,

and not just the input domain), and static analysis. In general, our ultimate goal is to combine the different technologies into a single coherent framework.

References

[1] C. Artho, K. Havelund, and A. Biere. High-level Data Races. In *VVEIS '03*, April 2003.

[2] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International Conference on Verification, Model Checking and Abstract Interpretation, January 2004 – to appear.*, August 2003.

[3] S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Internal report, to be published, October 2002.

[4] 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.

[5] 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.

[6] 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 `http://maude.csl.sri.com/papers`.

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

[8] Markus Dahm. BCEL. `http://jakarta.apache.org/bcel/`.

[9] 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.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] A. Goldberg and K. Havelund. Instrumentation of Java Bytecode for Runtime Analysis. In *Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports from ETH Zurich*, Switzerland, 2003. ETH Zurich.

[12] 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.

[13] 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.

[14] A. Hartman. Model Based Test Generation Tools. `http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf`.

[15] 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.

[16] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55-2 of *ENTCS*, pages 97–114, Paris, France, July 2001. Elsevier Science.

[17] 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.

[18] K. Havelund and G. Roşu. A Rewriting-based Approach to Trace Analysis. Submitted for journal publication, September 2002.

[19] 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 *LNCS*, pages 342–356. Springer, 2002.

[20] 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.

[21] Jieh Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.

[22] S. Khurshid, C. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of TACAS'03: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, Warsaw, Poland, April 2003.

[23] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[24] B. Korel. Automated Software Test Data Generation. *IEEE Transaction on Software Engineering*, 16(8):870–879, August 1990.

[25] O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From linear-time to branching-time. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.

[26] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proceedings of the Conference on Computer-Aided Verification*, 1999.

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

[28] Parasoft. `http://www.parasoft.com`.

[29] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[30] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 31(8), August 1992.

[31] *Purify: Fast Detection of Memory Leaks and Access Errors.* January 1992.

[32] G. Roşu and M. Viswanathan. Testing Extended Regular Language Membership Incrementally by Rewriting. In *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 499–514. Springer-Verlag, 2003.

[33] 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.

[34] 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.

[35] T-VEC. `http://www.t-vec.com`.

[36] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'02: The 15th IEEE International Conference on Automated Software Engineering.* IEEE CS Press, September 2000.