Model Checking of Statecharts using Automatic White Box Test Generation

Doron Drusinsky
Time Rover, Inc.,
11425 Charsan Lane, Cupertino, CA, 95014
www.time-rover.com

Abstract

This paper describes a model checking technique and tool for UML Statecharts based on automatic white box test-generation combined with automatic run-time monitoring of statechart assertions. The white box test generator is an automatically generated JUnit TestCase, which generates sequences of events, conditions, and input data for the System Under Test (SUT). It generates test sequences while observing the SUT's state, knowing the input events, conditions, and data objects that potentially affect the SUT's next state. The white box tester then chooses one of those events, conditions, and data objects, and fires the SUT, which in turn fires an embedded assertion for run-time monitoring. This combination of white box testing with assertion monitoring constitutes automatic model checking. The white-box tes-generator is also specification based in that the white box can be specified to be requirement assertions.

Harel Statecharts and Statechart Specifications

Harel statecharts have been described in numerous papers and books since first published by Harel [Ha] and later incorporated into the OMT methodology and eventually into the UML standard, (e.g. [Br, RB]). Statecharts extend finite state diagrams with hierarchy (state nesting), concurrence, and history states. Harel Statecharts are typically used for design analysis and modeling; for example, Brugge suggests using statecharts in the design analysis phase of an object oriented UML based design methodology [Br]. The tools described in this paper rely on an automata theoretic semantics for statecharts described in [D3].

The StateRover tool described in this paper is a code generator and visual debug animator for UML statecharts extended with features such as mixed flowcharts and statecharts, substatecharts, and critical regions. In addition, the StateRover supports run-time monitoring by providing a code generator for deterministic and non-deterministic Harel statecharts assertions as well as temporal logic assertions.

In [D3] Drusinsky described TLCharts, a hybrid of non-deterministic Harel statecharts and temporal logic conditions on statechart transitions and as statechart actions. The StateRover tool provides support for a subset of the TLCharts specification language where temporal logic assertions can only be specified in states and not as statechart transition guards.

Run-time Monitoring and Run-time Execution Recovery

Run-time Execution Monitoring (REM) is class of methods of tracking the temporal behavior of an underlying application. REM methods range from simple print-statement logging methods to run-time tracking of complex formal requirements (e.g., written in temporal logic) for verification purposes. Recently, NASA used REM for the verification of flight code for the Deep Impact project [DG]. The U.S. Missile Defense Agency (MDA) is currently applying REM to the verification of a new Ballistic Missile Defense System [Ca]. Published REM methods typically use temporal logic, Metric Temporal Logic (MTL), and regular expressions as a specification language [D1]. The StateRover tool described in this paper uses non-deterministic

0-7803-9197-7/05/$20.00 © 2005 IEEE.

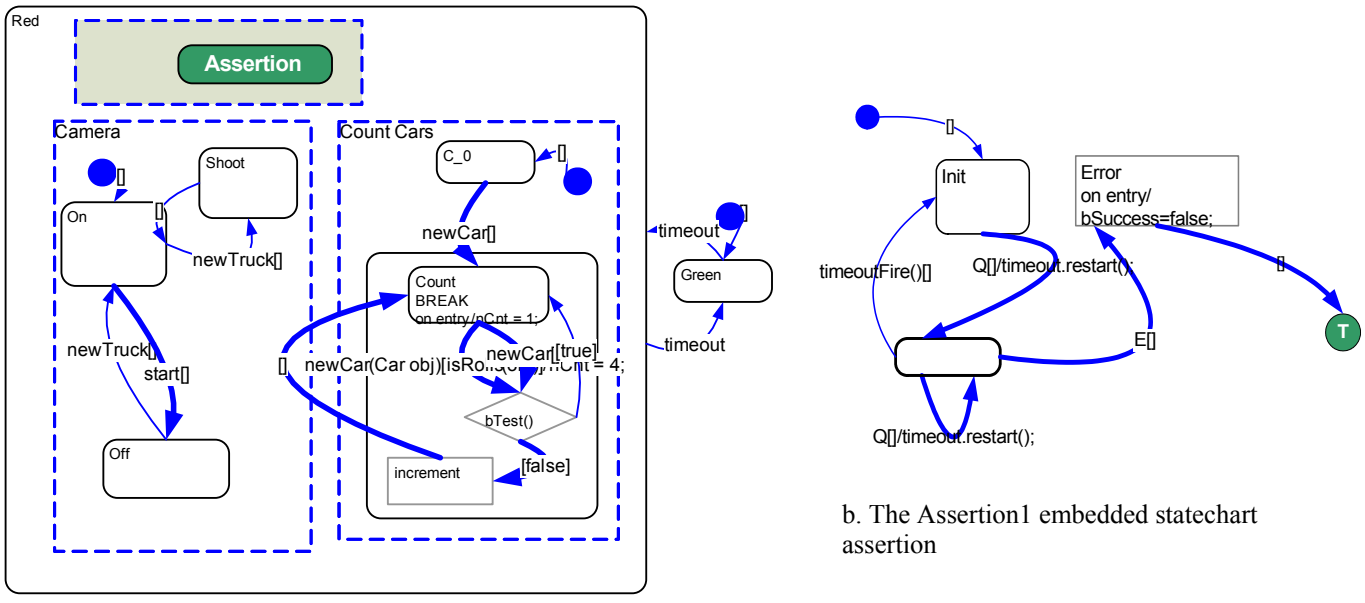
statecharts as a primary specification language but also enables the annotation of states with MTL assertions.

Run-time Execution Recovery (RER) from violations of formal requirement is a technique that integrates REM and the Monitored System (MS) in a closed loop so that the system, once notified of a formal specification violation, throws an exception and manages this exception in a predetermined manner. In [D2] Drusinsky describes a RER technique based on the heterogeneous coupling of REM and source code Java exception handling. This method uses a two layered approach where the REM tool manages specification and monitoring while recovery is performed using Java exception handling. In contrast, our technique uses statecharts as a single medium for specification, monitoring and recovery.

The white-box testing approach described in this paper combines automatic test generation with run-time monitoring of correctness properties described using deterministic and non-deterministic statechart assertions.

Statechart Assertions

A statechart assertion is a statechart embedded as a sub-statechart inside a primary statechart. For example, Fig. 1 illustrates an assertion statechart Assertion1 embedded within a primary Traffic Light Controller (TLC) statechart. Assertion1 of Fig. 1b asserts that *whenever event Q, then eventually, within time T, event E will occur*. While Assertion1 is generic, it is used inside the TLC as: *whenever newCar, then eventually, within time T, primary TLC will be in state CameraOn.* The StateRover automatically implements this mapping between the assertions event name space and the primary statechart event name space.



a. The primary (deterministic) TLC statechart Assertion1 embedded assertion

b. The Assertion1 embedded statechart assertion

Figure 1. A statechart TLC model with a statechart assertion

StateRover assertion statecharts always implement the (Java) *ITRAssertion* interface and primary statecharts with statechart assertions always implement the *ITRPrimary* interface. Using this interface mechanism, it is possible to refer to an unknown primary statechart within the assertion. For example, an assertion can assert about, and use, real-time measurements from an unknown primary statechart using the *primary.getTime*() method, where *getTime*() is part of the *ITRPrimary* interface. Similarly, a primary statechart can always refer to the assertion's *isSuccess*() method because this method is in the *ITRAssertion* contract interface. This method informs the primary about the success of the assertion. Moreover, the StateRover code generator will automatically construct an *isSuccess*() method for the primary; JUnit test suites for the primary can therefore assert about the outcome of the primary's *isSuccess*() method.

A statechart assertion has a natural scope defined by the primary statechart. Consider for example, *Assertion1* of Fig. 1. It is evident from the diagram that Assertion1 is only active while the primary statechart is in the *Red* state. Assertion1 has no effect when the primary transitions to the *Green* state and is reconstructed again when the primary returns to the *Red* state. Hence, statechart assertion are easily chopped and restarted, something that is not so easily done with other formalisms such as LTL [EFHL].

The StateRover tool provides a code generator for primary UML statecharts with embedded statechart assertions, such as the TLC of Fig. 1. This generated code, when executed, provides REM for the primary, namely, while the primary TLC statechart is executing its embedded assertions are also executing and monitoring its correctness.

The StateRover supports code generation for deterministic and non-deterministic statechart assertions. Non-deterministic statechart assertions are described in an accompanying paper [D4].

The following list summarizes the advantages of statechart assertions over temporal logic assertions:

- Visual, appeal, mostly familiar.
- More descriptive than LTL.
- Parameterized, reusable.
- Single layer exception handling available ([D4]).
- Single tool for UML statechart modeling and assertion development.
- Simple rule chaining technique [D4].
- Natural encapsulation of assertions within primary statecharts via Java.
- Simple communication between primary code and assertion code.

Model Checking and White Box Test Generation

Classical Model Checking (MC) is the process of automatically verifying the correctness of a correctness assertion in the context of a given model, or program. Classical MC techniques are typically algorithmic where the model is not actually executed, but rather analyzed in some manner.

With the advent of NASA's Java Path Finder (JFP) [HP], the definition of MC of software has been expanded to include the combination of automatic test generation with run-time monitoring.

JPF repeatedly executes the Java program under test, using a custom JVM. A user constructed test generation program, written in a JPF extension of Java, is used to specify the test suite. Hence, JPF can be viewed as an automatic black-box test generator. The debate whether this kind of MC is should be considered as MC rather than advanced testing is still on going.

The StateRover's automatic white box tester constructs a JUnit TestCase class from a given statechart model and associated embedded assertions, such as the TLC of Fig. 1. The JUnit test case executes a plurality of *test runs* of the statechart System Under Test (SUT). A typical white box test case consists of hundreds of thousands of runs of the SUT. Each test run consists of executing the SUT under a unique scenario developed by the white box tester, a scenario defined by a sequence of events, conditions, data objects, and simulated time delays, all driven from the white box statechart model and its associated assertions.

To illustrate the process consider the TLC of Fig 1a and its associated assertion of Fig. 1b. When the TLC is in its default state *Green*, it can only respond to one potential event namely, the *timeout* event. Note that at this time the assertion is not active because it is embedded in the *Red* state. Hence, the white box test generator emits the *timeout* event to the TLC SUT. When in the state configuration <On, C_0>, there are three events the SUT can react to: *newCar*, *newTruck*, and *timeout*. Meanwhile, the assertion, being in state *Init*, can only react to event Q which is mapped to *newCar*. The tester therefore generates one of the events *newCar*, *newTruck*, or *timeout*. It does so by using one of two algorithms: *stochastic* or *deterministic*. The stochastic algorithm determines the next event to emit using Java Random number generation. The deterministic algorithm determines the next event to emit in an orderly brute force manner, assuring exploration of all possibilities.

The underlying assumption behind this white box event generation technique is that events matter only if they are sensed by the SUT or sensed by an assertion. If an event does not affect the SUT or any assertion then it is considered redundant and not worth exploring. The only exception to this rule occurs when the SUT or the assertion have *Do* actions, i.e., actions that are performed as long as the statechart stutters in a given state. In this case the white box tester emits a special redundant event.

The white box tester explores a space induced by following SUT artifacts: events, transition guard conditions, data objects used by the SUT, and simulated time delays used by the built in *timeoutFire* event.

A well known issue with finite state machine and statechart testing involves the existence of loops. A statechart SUT that contains a loop has infinitely many potential tests. The StateRover test generator therefore requests that the user specify the max number of loop traversals permitted in a test. The user also specifies the max number of test runs within a white box test, and the maximal length of each test. Note that the number of potential tests is in general exponential with the permitted length of tests.

Each test run in a white box test is uniquely identified by an identification number (ID). This ID can be used later to automatically reconstruct a particular test without manually coding it. For example, once the user is informed that test-run #71 violates Assertion1 s/he can re-run that test-run for debugging purposes using the StateRover's debug animation feature. In addition, when in this single-test mode, the tester displays information about the particular settings for this test-run. This process is illustrated in Listing 1, which lists printout from a white box test for the TLC

(Fig. 1) SUT, specified to explore the SUT while avoiding loops, i.e., to explore all simple paths in the SUT, and Listing 2, which runs a particular test (ID 0_0_1_4_5) in a single-test mode:

```
13-failed tests (seed numbers):9, 10, 34, 40, 44, 50, 51, 54, 71, 80, 83, 89, 96
State visitation coverage:
All states were visited!

Failed assertions:
Assertion1

Number of test runs discovered:42

This covers ALL possible events sequences which induce paths with No. of loops:1
(simple paths), stutter=0
F
Time: 1.843
There was 1 failure:
1) testExecTReventDispatcher(TestPrimaryTLC6)junit.framework.AssertionFailedError
        at TestPrimaryTLC6.testExecTReventDispatcher(TestPrimaryTLC6.java:235)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at TestSuite1.main(TestSuite1.java:23)
FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

Listing 1. White box tester via JUnit output for the TLC SUT.

```
============= Seed: 71================
---> Cycle: 0
      Time increment = 7
      Event = start
---> Cycle: 1
      Time increment = 5
      Event = timeout
---> Cycle: 2
      Time increment = 2
      Event = newCar
---> Cycle: 3
      Time increment = 9
      Event = newTruck
---> Cycle: 4
      Time increment = 7
      Event = newCar
      Diamond Boolean value: nCnt > 3 = false
---> Cycle: 5
      Time increment = 7
      Event = eventTRfire()
F
Time: 0.141
There was 1 failure:
1) testExecTReventDispatcher(TestPrimaryTLC6)junit.framework.AssertionFailedError
        at TestPrimaryTLC6.testExecTReventDispatcher(TestPrimaryTLC6.java:328)
        at TestPrimaryTLC6.testExecTReventDispatcher(TestPrimaryTLC6.java:160)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
```

```
        at TestSuite1.main(TestSuite1.java:23)
FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

Listing 2. A single test mode execution of the tester for test ID 71.

The output from a white box test consists of the following information:

1. ID's of test runs that violated assertions.

2. ID's of test runs that reached certain, user specified, states.

3. States never reached during any test run.

4. The number of state configurations reached during all test runs vs. the number of state configurations in the SUT.

5. Estimated percentage of all possible tests (with a giving looping constant) covered by the current test.

References

[Br] B. Bruegge- Object-Oriented Software Engineering: Conquering Complex and Changing Systems, Prentice Hall, ISBN 0-13-489725-0.

[Ca] zzz the ISSRE paper with Butch et al

[DG] D. Drusinsky, G. Watney, Applying Run-Time Monitoring to the Deep-Impact Fault Protection Engine, 28'th IEEE/NASA Software Engineering Workshop, 2003.

[D1] D. Drusinsky - Monitoring Temporal Rules Combined with Time Series, Proc. 2003 Computer Aided Verification Conference (CAV), pp. 114-117.

[D2] D. Drusinsky - *Specs Can Handle Exceptions*. Embedded Developers Journal, November 2001, pp. 10-14. (http://eet.com/embedsub/archive.html).

[D3] D. Drusinsky - Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions, Fourth Workshop on RunTime Verification, ETAPS'04 Conference. Invited paper.

[D4] D. Drusinsky, Run-time Monitoring and Recovery of Harel Statecharts using Prioritized Non-deterministic Statechart Specifications, 48'th IEEE International Midwest Symposium on Circuits and Systems, Cincinnati OH, 2005 (MWSCAS 2005), accepted for publication.

[EFHL] C. Eisner, D. Fishman, J. Havlicek, Y. Lustig, A. McIsaac, D. Van Campenhout. "Reasoning with Temporal Logic on Truncated Paths", Proc. 2003 Computer Aided Verification Conference (CAV), pp. 27-39.

[Ha] D. Harel, A Visual Formalism for Complex Systems, Science of Computer Programming, 8, pp. 231-274, 1987.

[HP] K. Havelund, T. Pressburger, *"*Model Checking Java Programs Using Java PathFinder*",* International Journal on Software Tools for Technology Transfer, STTT, 2(4) April 2000.