2007-09

# A framework for computer-aided validation

Michael, James Bret; Shing, Man-Tak

Monterey, California. Naval Postgraduate School

NPS-CS-07-010


# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

## A Framework for Computer-aided Validation

By

D. Drusinsky, J. B. Michael, and M. Shing

September 2007



Prepared for: NASA IV&V Facility
100 University Drive
Fairmont, WV 26554

THIS PAGE INTENTIONALLY LEFT BLANK

**NAVAL POSTGRADUATE SCHOOL**
**Monterey, California 93943-5000**

Daniel T. Oliver
President

Leonard A. Ferrari
Provost

This report was prepared for and funded by the NASA IV&V Facility.



This report was prepared by:

_____
Man-Tak Shing
Associate Professor of Computer Science
Naval Postgraduate School

Reviewed by:

_____
Peter J. Denning, Chairman
Department of Computer Science

Released by:

_____
Dan C. Boger
Interim Associate Provost and
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

| REPORT DOCUMENTATION PAGE | | | Form Approved OMB No. 0704-0188 |
|---|---|---|---|
| Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503. | | | |
| **1. AGENCY USE ONLY** ***(Leave blank)*** | **2. REPORT DATE** September 2007 | **3. REPORT TYPE AND DATES COVERED** Technical Report | |
| **4. TITLE AND SUBTITLE**: Title (Mix case letters) **A Framework for Computer-aided Validation** | | | **5. FUNDING NUMBERS** NNG07LD01I |
| **6. AUTHOR(S)** D. Drusinsky, J.B. Michael, and M. Shing | | | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Naval Postgraduate School Monterey, CA 93943-5000 | | | **8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-07-010 |
| **9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)** NASA IV&V Facility, 100 University Drive, Fairmont, WV 26554 | | | **10. SPONSORING/MONITORING AGENCY REPORT NUMBER** |
| **11. SUPPLEMENTARY NOTES** The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. | | | |
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT** Approved for public release; distribution unlimited. | | | **12b. DISTRIBUTION CODE** |
| **13. ABSTRACT (maximum 200 words)** This paper presents a framework to incorporate computer-based validation techniques to the independent validation and verification (IV&V) of software systems. The framework allows the IV&V team to capture its own understanding of the problem and the expected behavior of any proposed system for solving the problem via an executable system reference model, which uses formal assertions to specify mission- and safety-critical behaviors. The framework uses execution-based model checking to validate the correctness of the assertions and to verify the correctness and adequacy of the system under test. | | | |
| **14. SUBJECT TERMS** Validation and verification, formal methods, model checking, runtime verification | | | **15. NUMBER OF PAGES** 22 |
| | | | **16. PRICE CODE** |
| **17. SECURITY CLASSIFICATION OF REPORT** Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified | **20. LIMITATION OF ABSTRACT** UL |

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

# 1. Introduction

According to the IEEE Std. 1012-2004 [1],

> the validation process provides evidence whether the software and its associated products and processes
> 1) Satisfy system requirements allocated to software at the end of each life cycle activity;
> 2) Solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions);
> 3) Satisfy intended use and user needs.

In short, validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose. However, the current IEEE Standard for Software V&V [1], the IEEE Guide for Developing System Requirements Specifications [2] and the IEEE Standard Glossary of Software Engineering Terminology [3] all define validation as "the process of evaluating a system or component during or at the end of the development process to determine whether a system or component satisfies specified requirements," and verification as "the process of evaluating a system or component to determine whether a system of a given development phase satisfies the conditions imposed at the start of that phase." These definitions give rise to a lot of computer-based validation and verification tools for checking the correctness of a target system or component against a formal model that is derived from the natural language requirements, and the consistency and completeness of the models, without ensuring that the developer understands the requirements and that the formal models correctly match the developer's cognitive intent of the requirements.

It is important for the independent validation and verification (IV&V) team to

> …formulate its own understanding of the problem and how the proposed system is solving the problem … [because] technical independence ("fresh viewpoint") is an important method to detect subtle errors overlooked by those too close to the solution. [1]

The IV&V team's independent requirements effort should develop a description of the necessary attributes, characteristics, and qualities of ***any*** system developed to solve the problem and satisfy the intended use and user needs. The IV&V team must ensure that their cognitive understanding of the problem and the requirements for any system solving the problem are correct before performing IV&V on developer-produced systems.

In order to use computer-based V&V technology, the IV&V team needs to develop formal, executable representations of the system properties. These properties can be expressed as a set of desired system behaviors, which in turn can be divided into the following two classes:

(1) Logical behavior - This class describes the cause and effect of a computation, typically represented as functional requirements of a system.

(2) Sequencing behavior – This class describes the behaviors that consist of sequences of events, conditions and constraints on data values, and timing. In its vanilla form, sequencing behavior specifies sets of legal (or illegal) sequences, such as the following automotive body-logic requirement:

> *Once engine is turned off, compartment lights must be on until driver door is opened.*

On top of pure sequencing, this kind of behavior can specify two types of constraints:

(a) Timing constraints – describe the timely start and/or termination of successful computations at a specific point of time, such as the deadline of a periodic computation or the maximum response time of an event handler.

(b) Time-series constraints – describe the timely execution of a sequence of data values within a specific duration of time. For example,

> *Whenever the track count (cnt) Average Arrival Rate (ART) exceeds 80% of the MAX_COUNT_PER_MIN, cnt ART must be reduced back to 50% of the MAX_COUNT_PER_MIN within 2 minutes and cnt ART must remain below 60% of the MAX_COUNT_PER_MIN for at least 10 minutes.*

This paper presents a framework to incorporate advanced computer-aided validation techniques to the IV&V of software systems. The framework allows the IV&V team to capture its own understanding of the problem and the expected behavior of any proposed system for solving the problem via an executable system reference model. For the rest of this paper, we shall use the term "developer-generated requirements" to mean the requirements artifacts produced by the developer of a system (which include both functional and non-functional requirements), and use the term "System Reference Model" to denote the artifacts developed by the IV&V team's own requirements effort.

## 2. Creation and Validation of the System Reference Models

In this paper, we advocate the use of a system reference model (SRM) to capture the IV&V team's understanding of the problem. A SRM is made up of a set of use cases, Unified Modeling Language (UML) artifacts (e.g., activity diagrams, sequence diagrams, and object class diagrams), and a set of formal assertions to describe precisely the necessary behaviors to satisfy system goals (i.e., to solve the problem) with respect to: (a) what the system should do, (b) what the system should not do, and (c) how the system should respond under non-nominal circumstances.

## 2.1 The Use Cases and UML Artifacts of the System Reference Model

The starting point of both understanding and documenting system behaviors is to identify the high-level use cases (and use case scenarios) from the stakeholder's input, which could be in the form of mission statements, user expectations, and operation concepts (and other concept-level documents). The use cases help the system analysts understand the problems to be solved and the objectives to be accomplished by the perceived system(s). The high-level use cases are goal-oriented (instead of function-oriented), and typically are used to describe the workflow of a business (or operation) process instead of interactions among system components. Mapping the scenarios of the use cases to activity diagrams helps both highlight the assignment of responsibilities and the interdependencies among the different components (of an organization or system).

For the purpose of the IV&V of software systems, the high-level use cases must be reified into mission threads (i.e., detailed use cases) that capture the interactions among the component systems (or sub-systems). Mapping the detailed use cases to sequence diagrams helps highlight the system events and the corresponding responses to be exhibited by the system. In addition to capturing interactions, the analysts need to record all relevant system attributes and constraints that they discover as they refine the use cases. A use case typically describes what the system should do. However, the analysts may need to develop misuse cases [4] to capture what the system should not do.

Concurrent to the development of use cases (and their scenarios), and activity and sequence diagrams, the analysts must also develop a conceptual model (in the form of an object class diagram) to capture the essential concepts and manage the namespace of the problem.

## 2.2 The Formal Assertions of the System Reference Model

IV&V traditionally relies on manual examination of software requirements and design artifacts, manual and tool-based code analysis, and the systematic or random independent testing of target code. Most of these techniques are ineffective for validating the correctness of the developer's cognitive understanding of the requirements. Moreover, as software-intensive systems become increasingly complex, manual IV&V techniques are inadequate for locating the subtle errors in the software. For example, there are intricate and abstruse sequencing behaviors that are only observable at runtime and at such a fine level of granularity of time that human intervention at runtime is not practical. Software automation holds the key to the validation and verification of these types of system behaviors, and formal specification of system behaviors is the enabling factor for software automation.

In [5], we classify formal behavioral specifications into two categories – *assertion-* and *model-oriented specifications*.

With *assertion-oriented* specifications, high-level requirements are decomposed into more precise requirements that are mapped one-to-one to formal assertions. For example, we may start with a high-level requirement

> *R1. The track processing system can only handle a workload not exceeding 80% of its maximum load capacity at runtime.*

and derive the lower level requirement

> *R1.1 Whenever the track count (cnt) Average Arrival Rate (ART) exceeds 80% of the MAX_COUNT_PER_MIN, cnt ART must be reduced back to 50% of the MAX_COUNT_PER_MIN within 2 minutes and cnt ART must remain below 60% of the MAX_COUNT_PER_MIN for at least 10 minutes.*

The requirement R1.1 will, in turn, be mapped to a formal assertion expressed as a Statechart assertion A1 shown in Figure 1, which is made up of a combination of UML statecharts and flowcharts. The statechart assertions are written from the standpoint of an observer and can be used for runtime monitoring of the target application [6]. (Readers can refer to Section 7.1 for an explanation of the Statechart assertion A1.)

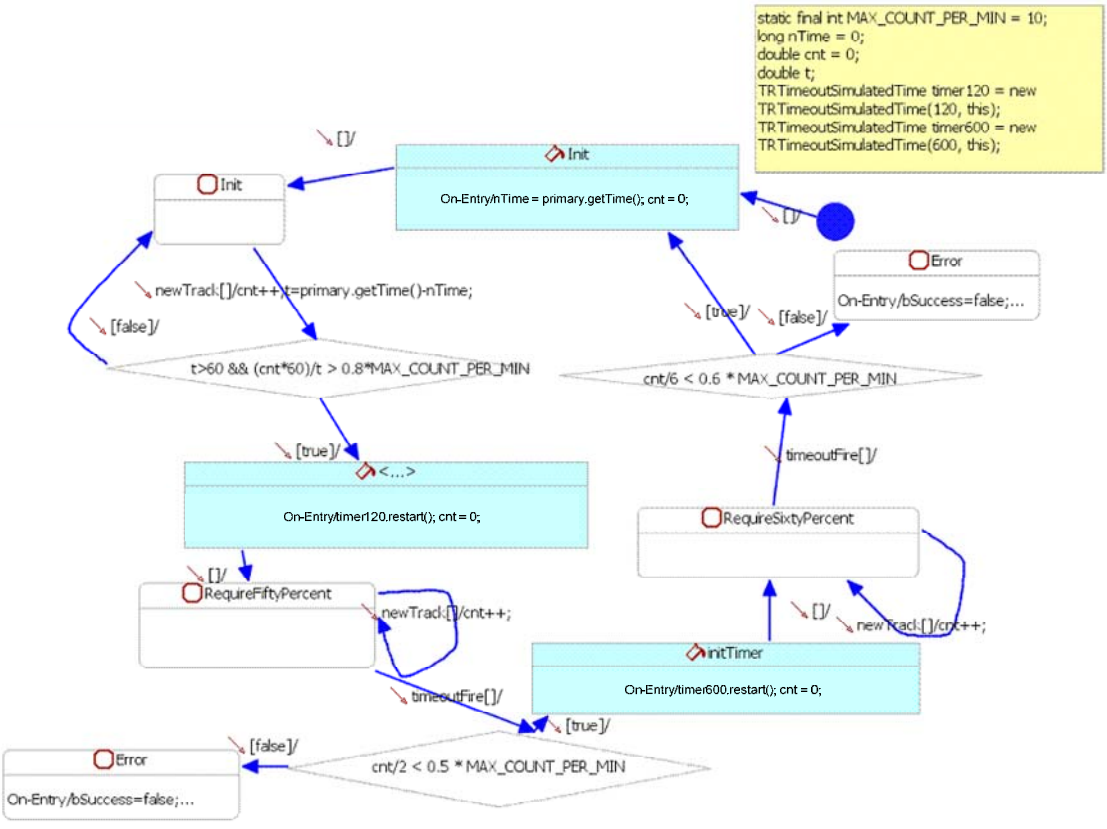


Figure 1. A sample Statechart assertion A1

With *model-oriented* behavioral specifications, a *single* monolithic formal model (either as a state- or an algebraic-based system) captures the combined expected behavior described by the lower level specifications of behavior. Note that this formal model describes the expected behavior of a conceptualized system from the IV&V team's understanding of the problem space. It may differ significantly from the system design models created by the developers in their design space.

We favor the assertion-oriented specification approach due to its following advantages over the model-oriented specification approach:

(1) Requirements are written by humans and need to be traceable in the formal specification. Requirements are indeed traceable in the assertion-oriented formal specification approach because they are represented, one-to-one, by assertions (acting as watchdogs for the requirements).

    A monolithic model specification on the other hand is the sum of all concerns. Hence, on detecting a violation of the formal specification, it is difficult to map that violation to a specific human-driven requirement.

(2) When a requirement changes, it is harder to adjust the monolithic model without affecting the behavior related to other requirements. Hence, assertion-oriented specifications have a lower maintenance cost in this regard than the model-oriented counterpart.

(3) Particular assertions can be constructed to represent illegal behaviors, whereas the monolithic model typically only represents "good behavior."

(4) It is much easier to trace the expected and actual behaviors of the target system to the required behaviors in the requirements space with assertion-oriented specifications than with the model-oriented specifications. The formal assertions can be used directly as input to the verifiers in the verification dimension.

(5) The conjunction of all the assertions becomes a "single" formal model of a conceptualized system from the requirement space, and can be used to check for inconsistencies and other gaps in the specifications with the help of computer-aided tools.

### 2.3 Validation of the Formal Assertions

We argue that the formal assertions must be *executable* to allow the modelers to visualize the true meaning of the assertions via scenario simulations. For example, the Software Cost Reduction (SCR) Toolset contains a simulator for use in executing a series of scenarios against the executable model to determine whether the specification captures

the intended behavior [7]. In [8], we presented an iterative process that allows the modeler to write formal specifications using Statechart assertions, and then validate the correctness of the assertions via simulated test scenarios within the JUnit test-framework (Figure 2).

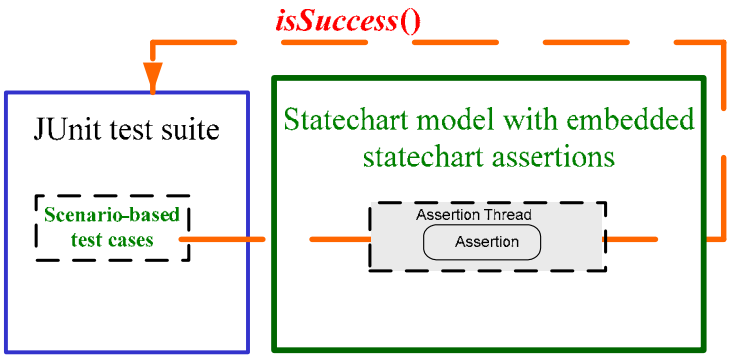


Figure 2. Validation of statechart assertion via scenario-based testing

For example, the IV&V team can test the Statechart assertion A1 with a scenario in which the system receives more than eight newTracks in one minute, then successfully reduces the workload to fewer than five per minute in the next two minutes followed by fewer than six per minute in the following ten-minute period, resulting in a successful test outcome. The IV&V team may choose to exercise the Statechart assertion on other scenarios to increase their confidence that the assertion is correct. For example, they may test the Statechart assertion with another scenario in which the system receives more than eight newTracks in one minute, then attempts recovery (fewer than five per minute in the next two minutes), but fails at the end because there are more than six newTracks per minute in the following ten-minute period. (Readers can refer to Sections 7.2 and 7.3 for the Java source code of the two scenarios.)

### 2.4 A Process for Formal-specification and Computer-aided Validation of Complex System Behavior

Using the executable SRM and the execution-based validation technique, the IV&V team can formally capture its understanding of the problem and the requirements for any system solving the problem, and validate the correctness of their cognitive understanding with the process shown in Figure 3. First, individual assertions are tested using the scenario-based test cases, like those shown in Sections 7.2 and 7.3, to validate the correctness of the logical and temporal meaning of the assertions (circuit #1 in Figure 3). Then, the assertions are tested using the scenario-based test cases subjected to the constraints imposed by the objects in the SRM conceptual model (circuit #2 in Figure 3). For example, the conceptual model may impose a limit on the number of vehicles the system has to monitor during operation. Finally, the IV&V team can use the white-box automatic tester to exercise all assertions together to detect any conflicts in the formal specification (circuit #3 in Figure 3).

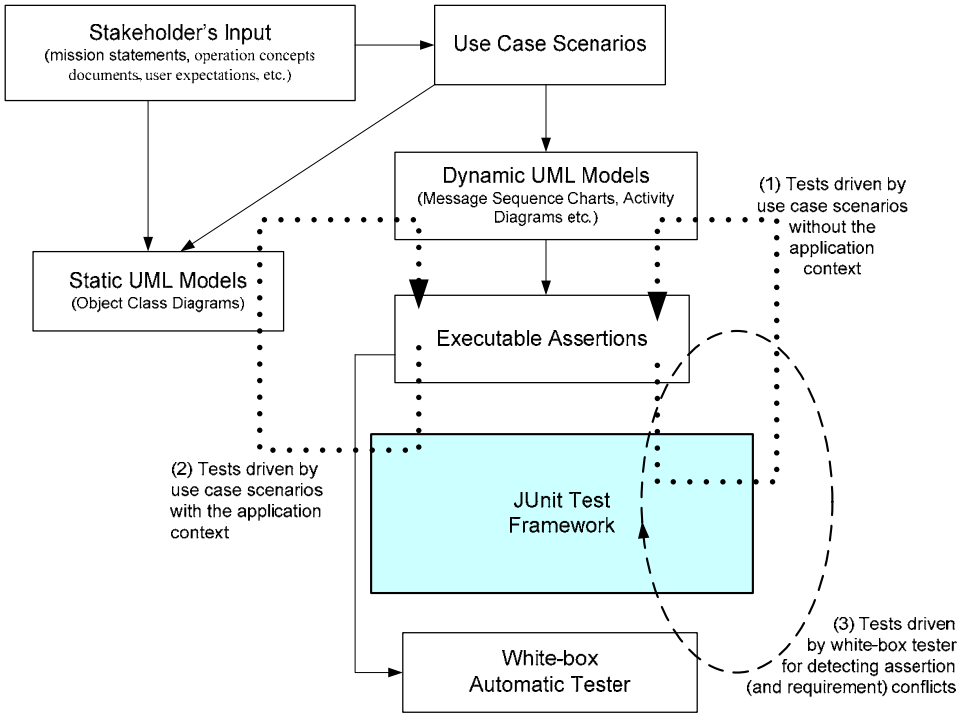


Figure 3. A process for formal specification and computer-aided validation

## 3. Application of the System Reference Models

One major benefit of using an executable SRM is its support for conducting runtime verification of the software produced by the developer. *Runtime Verification* (RV) is a verification technique that monitors the runtime execution of a system and checks the observed runtime behavior against the system's formal specification. Hence, RV serves as an automated observer of the program's behavior and compares it with the expected behavior per the formal specification. To use RV, the software artifacts produced by the developer needs to be instrumented, with the degree of instrumentation being dependent on the software methodologies used by the developer.

In the following two sections, we illustrate the application of RV in two different scenarios. Section 3.1 describes a scenario where state-based design models are available as part of the developer's development process, while Section 3.2 describes a different scenario where only executable code is available to the IV&V team.

### 3.1 Verification of State-based Design Models

In the event that the state-based design models are available to the IV&V team, the IV&V team can apply *Execution-based Model Checking* (EMC) to verify the state-based models against the SRM. EMC is a combination of RV and Automatic Test Generation (ATG). Some ATG tools that, when combined with RV tools, create an EMC technique are the StateRover's white-box automatic test-generator [9] and NASA's Java Path Finder (JPF) [10]. With EMC, a large volume of automatically generated tests are used to exercise the program or system under test, using RV on the other end to check the SUT's conformance to the formal specification.

With this approach, the IV&V team will need to re-enter the state-based design models as StateRover statecharts (called the primary statecharts) and embed the statechart assertions of the SRM as sub-statecharts of the resultant statechart model. The IV&V team then uses the StateRover code generator to create an executable model from the instrumented statecharts, and test the model with the white-box tester (Figure 4).

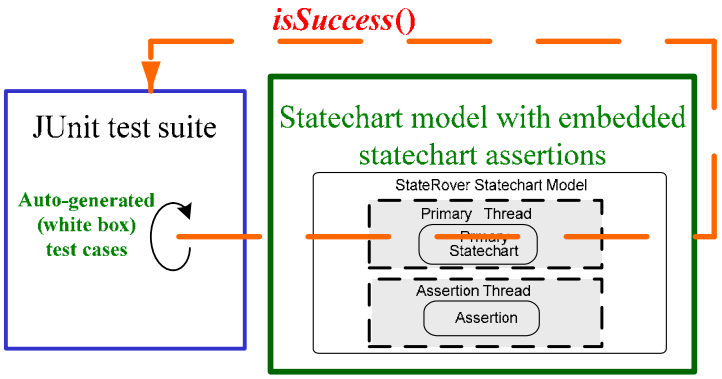


Figure 4. Execution-based model checking of state-based design models

The StateRover's automatic white-box tester constructs a JUnit TestCase class from a given statechart model and the associated embedded assertions. A typical JUnit white-box test case consists of hundreds of thousands of runs of the statechart under test (SUT). The auto-generated tests are used in three ways:

(1) To search for severe programming errors, of the kind that induces a JUnit error status, such as *NullPointerException*.

(2) To identify test cases which violate temporal assertions.[1]

[1] To help statechart designers pinpoint specific errors, each failed test run is reported with an identification number. The causes of failure for a specific run can be investigated in detail by running the automatic white-box tester in *single test/run mode*. Such a mechanism helps developers to eliminate errors in their design in an efficient manner.

(3) To identify input sequences that lead the SUT to particular states of interest.

The StateRover generated *WBTestCase* creates sequences of events and conditions for the SUT. The WBTestCase is nontrivial in the following regard: it creates only sequences consisting of events that the SUT or some assertion is sensitive to, by repeatedly observing all events that potentially affect the SUT when it is in a given configuration state, selects one of those events and fires the SUT using this event. The WBTestCase auto-generates three artifacts:

(1) Events, as described above.

(2) Time-advance increments, for the correct generation of timeoutFire events.

(3) External data objects of the type that the statechart prototype refers to.

The above procedure describes the model-based aspect of the StateRover's White-Box Automatic Test Generator (WBATG). However, the WBATG actually observes all entities, namely, the SUT and all embedded assertions. It collects all possible events from all of those entities, thus creating a hybrid model- and specification-based WBATG.

### 3.2 Verification of Target Code

In the event that only executable code is available, the IV&V team can use the StateRover white-box tester in tandem with the executable assertions of the SRM to automate the testing of the target code produced by the developer using the architecture shown in Figure 5.

The white-box tester acquires the set of all possible "next" events from the statechart assertions, and selects one of those events and sends the event to the SUT and to the assertion statecharts. The white-box tester also maintains a timer that controls the tempo of the test. The white-box tester advances the timer to the next meaningful value whenever a timeoutFire event is selected.

The statechart assertions of the SRM have the following responsibilities in the proposed test architecture: (i) keeping track of the set of possible next events to drive the SUT, and (ii) serving as the observer for the RV during the test.

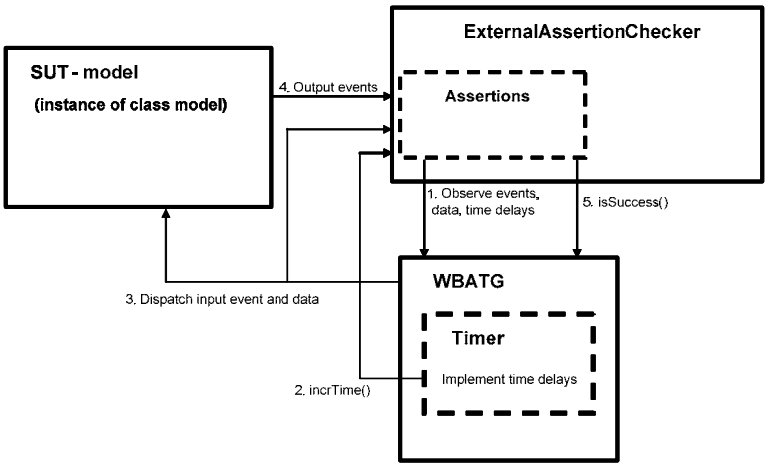


Figure 5. Automated testing using the system reference model

### 3.3 Manual Examination of the Developer Generated Requirements

Although not as effective as execution-based model checking, the IV&V team can also use the SRM to validate the textual descriptions of the requirements produced by the developer. The IV&V team will start by associating the developer-generated requirements with the use cases. This will provide the context for assessing the requirements. Next, the IV&V team can trace the developer-generated requirements to the other artifacts. For example, tracing the requirements to the activity and sequence diagrams can help the analyst identify the subsystems or components responsible for the system requirements and trace the developer-generated requirements to the domain model to identify the correct naming of the objects and events. These requirement traces may also help in identifying the critical components of the target system for more thorough testing.

## 4. Conclusion

In this paper, we discussed the importance for the IV&V team to capture its own understanding of the problem to be solved and the expected behavior of any system for solving the problem, using a SRM. We argued that complex system sequencing behaviors can mainly be understood and their formal specifications can most effectively be validated via execution-based techniques, and advocate the use of assertion-oriented specification over the model-oriented specification for the SRM. We presented a framework for incorporating computer-aided validation into the IV&V of complex reactive systems, and showed how the SRM can be used to automate the testing of the software artifacts produced by the developer of the system.

## 5. Acknowledgement

We thank the members of the NASA IV&V Facility's Core Modeling Group for reviewing this report. The research was funded in part by a grant from the National Aeronautics and Space Administration. The views and conclusions in this talk are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

## 6. References

[1] *IEEE Std. 1012-2004, IEEE Standard for Software Verification and Validation*, IEEE, 2004.

[2] *IEEE Std. 1233-1998, IEEE Guide for Developing System Requirements Specifications*, IEEE, 1998.

[3] *IEEE Std. 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

[4] I. Alexander, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software*, 20, 1 (2003), pp. 58-66.

[5] D. Drusinsky, B. Michael and M. Shing, *The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors*, Tech. Rpt. NPS-CS-07-008, Dept. of Computer Science, Naval Postgraduate School, Monterey, Calif., August 2007.

[6] D. Drusinsky, *Modeling and Verification Using UML Statecharts - A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, Elsevier, 2006.

[7] C. Heitmeyer, "Formal Methods for Specifying, Validating, and Verifying Requirements," *J. Universal Computer Science*, 13, 5 (2007), pp. 607-618.

[8] D. Drusinsky, M. Shing, and K. Demir, "Creating and Validating Embedded Assertion Statecharts," *IEEE Distributed Systems Online*, 8, 5 (2007), art. no. 0705-o5003.

[9] D. Drusinsky, *Modeling and Verification Using UML Statecharts - A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, Elsevier, 2006.

[10] K. Havelund and T. Pressburger, “Model Checking Java Programs using Java PathFinder,” *Int’l J. Software Tools for Technology Transfer*, 2, 4 (2000), pp. 366-381.

# 7. Appendix

## 7.1 Description of the Statchart Assertion A1

The statechart assertion A1 realizes the natural language requirement R1.1 as follows. After initializing the local variables *nTime* to the current time and *cnt* to zero, the startchart assertion enters the *Init* state to observe the arrival of the *newTrack* events. With the arrival of each newTrack event, it updates the variables *cnt* and *t* and evaluates the condition in the first decision box to see if *track count (cnt) Average Arrival Rate (ART) exceeds 80% of the MAX_COUNT_PER_MIN*. The statechart assertion will reset *cnt* to zero, start the 2-minute timer (*timer120*), and enter the *RequireFiftyPercent* state if the condition becomes true. The statechart assertion stays in the *RequireFiftyPercent* state and keeps tracks of the number of *newTrack* events for two minutes. When the *timer120* fires, it evaluates the condition in the second decision box to see if *cnt ART falls below 50% of the MAX_COUNT_PER_MIN*. It will enter the *Error* state and sets *bSuccess* to false, indicating the violation of the assertion, if the condition is false. Otherwise, the statechart assertion will reset *cnt* to zero, start the 10-minute timer (*time600*), and enter the *RequireSixtyPercent* state. The statechart assertion keeps tracks of the number of *newTrack* events for ten minutes in the *RequireSixtyPercent* state, and, when the *timer600* fires, it evaluates the condition in the third decision box to see if *cnt ART remains below 60% of the MAX_COUNT_PER_MIN*. It will enter the *Error* state and sets *bSuccess* to false, indicating the violation of the assertion, if the condition is false. Otherwise, it will reset *nTime* to the current time and *cnt* to zero, and returns to the *Init* state.

Note that the statechart assertion A1 represents one of the many possible interpretations of the natural language requirement R1.1. A different analyst from the one who constructed A1 could have a separate interpretation of the meaning of the *track count (cnt) Average Arrival Rate (ART)*. This highlights the importance of expressing natural language requirements as formal assertions to gain a deeper understanding of the system behavior being specified, and to uncover inconsistencies, ambiguities and incompletenesses in behavior specifications of the behavior of the system.

## 7.2 Test Scenario 1

**Here is the Java source code of scenario 1.**

```
import junit.framework.*;

public class TestVVFrameworkExample1 extends TestCase {
    private VVFrameworkExample assertion = null;
    private MockupPrimary mockupPrimary = null;

    protected void setUp() throws Exception {
        super.setUp();
        /**@todo verify the constructors*/
        assertion = new VVFrameworkExample(false);
        mockupPrimary = new MockupPrimary(assertion);
        // mock the relationship primary <-> assertion
        assertion.setTRPrimary(mockupPrimary);
    }

    protected void tearDown() throws Exception {
        assertion = null;
        mockupPrimary = null;
        super.tearDown();
    }

    // Test scenario 1
    // More than 8 newTracks in 1 min, then recovery (fewer than
    // 5 per min in 2 min followed by fewer than 6 per min in 10
    // min period)
    public void testExecTReventDispatcher() {
         mockupPrimary.setTime(0); //start time
         assertion.newTrack(); // 1
         mockupPrimary.setTime(10);
         assertion.newTrack(); // 2
         mockupPrimary.setTime(20);
         assertion.newTrack(); // 3
         mockupPrimary.setTime(30);
         assertion.newTrack(); // 4
         mockupPrimary.setTime(35);
         assertion.newTrack(); // 5
         mockupPrimary.setTime(40);
         assertion.newTrack(); // 6
         mockupPrimary.setTime(45);
         assertion.newTrack(); // 7
         mockupPrimary.setTime(50);
         assertion.newTrack(); // 8
         assertTrue(assertion.isState("Init"));
         mockupPrimary.setTime(62);
         assertion.newTrack(); // 9 -- more than 8
         assertTrue(assertion.isState("RequireFiftyPercent"));

         // now fewer than 5 per min for 2 min
         assertion.newTrack(); // 1
         mockupPrimary.setTime(65);
         assertion.newTrack(); // 2
```

```
            mockupPrimary.setTime(70);
            assertion.newTrack(); // 3
            mockupPrimary.setTime(71);
            assertion.newTrack(); // 4
            mockupPrimary.setTime(75);
            assertion.newTrack(); // 5
            mockupPrimary.setTime(115);
            assertion.newTrack(); // 6
            mockupPrimary.setTime(120);
            assertion.newTrack(); // 7
            mockupPrimary.setTime(125);
            assertion.newTrack(); // 8
            assertTrue(assertion.isState("RequireFiftyPercent"));
            mockupPrimary.setTime(200); // by now 2 min have elapsed
            assertTrue(assertion.isState("RequireSixtyPercent"));
            assertTrue(assertion.isSuccess());

            // now fewer than 6 per min for 10 min
            assertion.newTrack(); // 1
            mockupPrimary.setTime(300);
            assertion.newTrack(); // 2
            mockupPrimary.setTime(400);
            assertion.newTrack(); // 3
            mockupPrimary.setTime(900); // trigger second timer

            assertTrue(assertion.isSuccess());
        }
    }
```

### 7.3 Test Scenario 2

**Here is the Java source code of scenario 2.**

```
    // Test scenario 2:
    public void testExecTReventDispatcher() {
        mockupPrimary.setTime(0); //start time
        assertion.newTrack(); // 1
        mockupPrimary.setTime(10);
        assertion.newTrack(); // 2
        mockupPrimary.setTime(20);
        assertion.newTrack(); // 3
        mockupPrimary.setTime(30);
        assertion.newTrack(); // 4
        mockupPrimary.setTime(35);
        assertion.newTrack(); // 5
        mockupPrimary.setTime(40);
        assertion.newTrack(); // 6
        mockupPrimary.setTime(45);
        assertion.newTrack(); // 7
        mockupPrimary.setTime(50);
        assertion.newTrack(); // 8
        assertTrue(assertion.isState("Init"));
        mockupPrimary.setTime(62);
```

```
        assertion.newTrack(); // 9 -- more than 8
        assertTrue(assertion.isState("RequireFiftyPercent"));

        // now fewer than 5 per min for 2 min
        assertion.newTrack(); // 1
        mockupPrimary.setTime(65);
        assertion.newTrack(); // 2
        mockupPrimary.setTime(70);
        assertion.newTrack(); // 3
        mockupPrimary.setTime(71);
        assertion.newTrack(); // 4
        mockupPrimary.setTime(75);
        assertion.newTrack(); // 5
        mockupPrimary.setTime(115);
        assertion.newTrack(); // 6
        mockupPrimary.setTime(120);
        assertion.newTrack(); // 7
        mockupPrimary.setTime(125);
        assertion.newTrack(); // 8
        assertTrue(assertion.isState("RequireFiftyPercent"));
        mockupPrimary.setTime(200); // by now 2 min have elapsed
        assertTrue(assertion.isState("RequireSixtyPercent"));
        assertTrue(assertion.isSuccess());


        // now more than 6 per min for 10 min
        assertion.newTrack(); // 1
        mockupPrimary.setTime(300);
        assertion.newTrack(); // 2
        mockupPrimary.setTime(400);
        assertion.newTrack(); // 3
        mockupPrimary.setTime(400);
        assertion.newTrack(); // 3
        for (int i = 0; i < 97; i++) {
            mockupPrimary.setTime(500+i);
            assertion.newTrack();
        }
        mockupPrimary.setTime(600);
        assertion.newTrack();
        mockupPrimary.setTime(620);
        assertion.newTrack();
        mockupPrimary.setTime(900); // trigger second timer

        assertFalse(assertion.isSuccess());
    }
```

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52
   Naval Postgraduate School
   Monterey, CA

3. Research Office, Code 09
   Naval Postgraduate School
   Monterey, CA

4. Dr. Butch Caffall
   NASA IV&V Facility
   Fairmont, WV

5. LTC Thomas Cook
   Naval Postgraduate School
   Monterey, CA

6. Dr. Doron Drusinsky
   Naval Postgraduate School
   Monterey, CA

7. Dr. Bret Michael
   Naval Postgraduate School
   Monterey, CA

8. Dr. Man-Tak Shing
   Naval Postgraduate School
   Monterey, CA

9. Dr. Rudy Panholzer
   Naval Postgraduate School
   Monterey, CA