Innovations Syst Softw Eng (2008) 4:161–168
DOI 10.1007/s11334-008-0047-2

ORIGINAL PAPER

A framework for computer-aided validation

Doron Drusinsky · James Bret Michael · Man-Tak Shing

Received: 24 October 2007 / Accepted: 11 February 2008 / Published online: 11 March 2008

Abstract This paper presents a framework for augmenting independent validation and verification (IV&V) of software systems with computer-based IV&V techniques. The framework allows an IV&V team to capture its own understanding of the application as well as the expected behavior of any proposed system for solving the underlying problem by using 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.

Keywords Validation and verification · Formal methods · Model checking · Runtime verification

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

D. Drusinsky · J. B. Michael · M. Shing (✉)
Department of Computer Science, Naval Postgraduate School, 1411 Cunningham Road, Monterey, CA 93943, USA
e-mail: shing@nps.edu

D. Drusinsky
e-mail: ddrusins@nps.edu

J. B. Michael
e-mail: bmichael@nps.edu

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 many 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 to have an *independent* validation and verification (IV&V) team because it can formulate its own understanding of the problem and the manner in which the proposed system solves the problem. This is because technical independence ("fresh viewpoint") has a greater chance of detecting subtle errors overlooked by the developer, who is often too close to the solution.

The IV&V team's independent requirements effort should yield a description of the necessary attributes, characteristics, and qualities of any system that solves the problem and satisfies the intent. The IV&V team must, therefore, ensure that their cognitive understanding of the problem and corresponding requirements are correct before performing IV&V on developer-produced systems. This process is the notion of validation referred to in this paper.

In order to use computer-based validation, the IV&V team needs to develop formal, executable representations of the system's correctness properties. These properties are expressed as a set of desired system behaviors, which in turn are 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. For example, given two positive numbers x and e, the output of the square root function sqrt(x) must satisfy the requirement: $|x - \text{sqrt}(x) * \text{sqrt}(x)| < e$.
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 lighting requirement:

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

   Sequencing behavior has two types of possible 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. For example, the sqrt() function must complete its computation and return an answer within 200 ms from the time it is called.
   (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 min and cnt ART must remain below 60% of the MAX_COUNT_PER_MIN for at least 10 min.

This paper presents a framework for incorporating computer-aided validation techniques for the purpose of 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" (SRM) 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 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 and 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

Use cases help system analysts understand the underlying problems to be solved by—and the objectives to be accomplished by—the perceived system(s). High-level use cases are typically goal-oriented (instead of function-oriented), and used to describe the workflow of a process instead of interactions among system components. Mapping use-case scenarios to activity diagrams helps in visualizing this process as well as highlighting responsibilities and interdependencies within the collection of system components.

With the end-goal of software-system IV&V in mind, high-level use cases should be reified into detailed use cases in the form of mission threads; the threads capture interactions among member components and sub-systems. Mapping detailed use cases to sequence diagrams helps highlight system events and corresponding expected system responses. Note that while a use case typically describes what the system should do, analysts should also develop misuse cases [4] to capture what the system should *not do*.

Concurrent to the development of use cases, activity diagrams and sequence diagrams, the analysts must also develop a conceptual model (in the form of a class diagram) to capture the important concepts (as object classes) and constraints [as Object Constraint Language (OCL) expressions] of the underlying 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

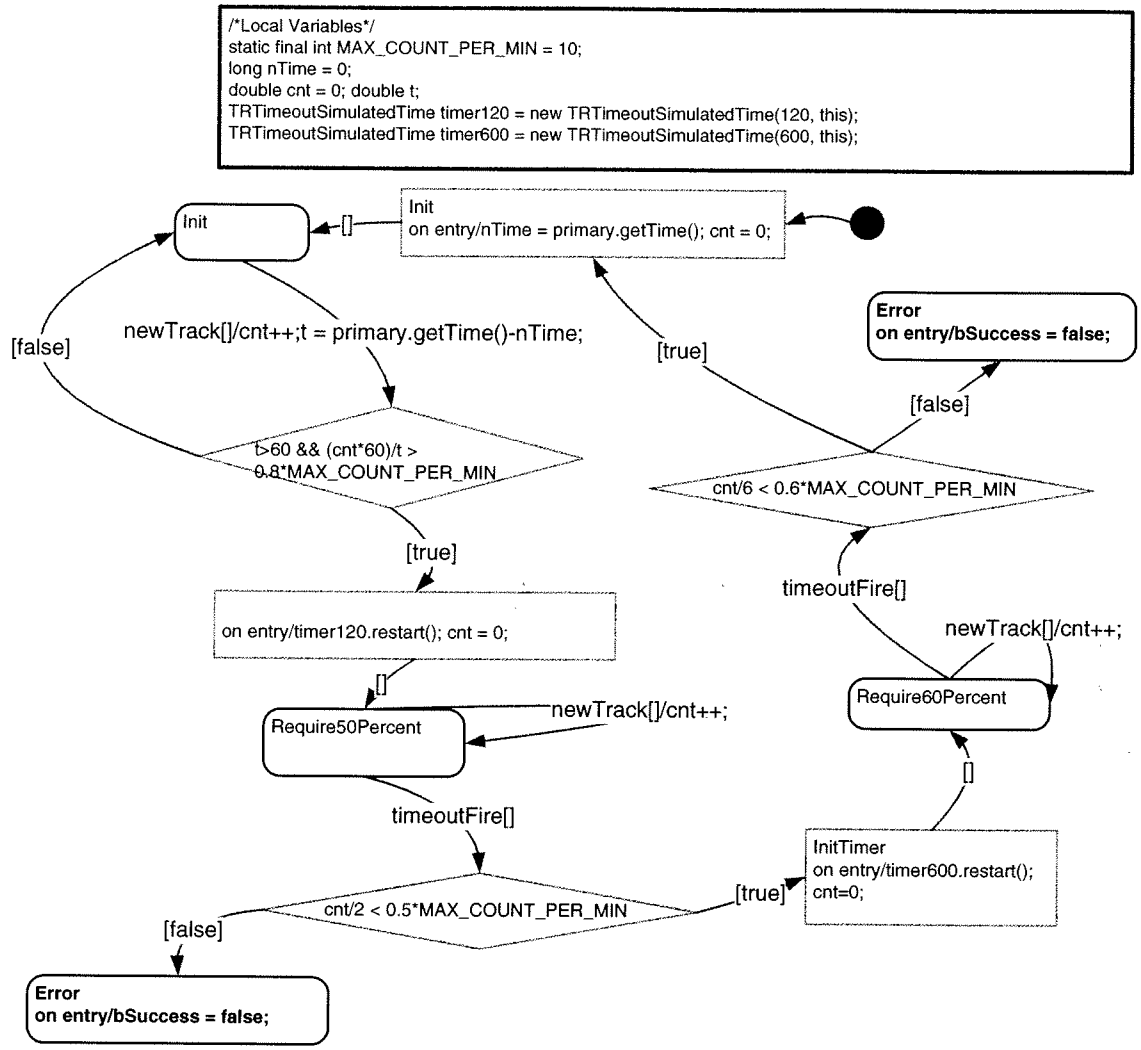

Fig. 1 A sample statechart assertion A1

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 cnt 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 min and cnt ART must remain below 60% of the MAX_COUNT_PER_MIN for at least 10 min.

The requirement R1.1 will, in turn, be mapped to a formal assertion expressed as a statechart assertion A1 shown in Fig. 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 Appendix A1 for an explanation of the 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 their 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.

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 (Fig. 2).

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 1 min, then successfully reduces the workload to fewer than five per minute in the next 2 min followed by fewer than six per minute in the following 10-min

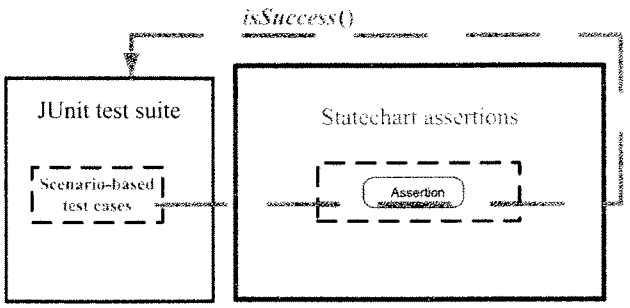

Fig. 2 Validation of statechart assertion via scenario-based testing

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, the team 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 2 min), but fails at the end because there are more than six newTracks per minute in the following 10-min period. (Readers can refer to Appendices A2 and A3 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 underlying problem and the requirements for any system solving the problem, and validate the correctness of their cognitive understanding with the process shown in Fig. 3. First, individual assertions are tested using the scenario-based test cases, like those shown in Appendices A2 and A3, to validate the correctness of the logical and temporal meaning of the assertions (circuit #1 in Fig. 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 Fig. 3). For example, the conceptual model may impose a limit on the number of objects 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 Fig. 3).

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 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

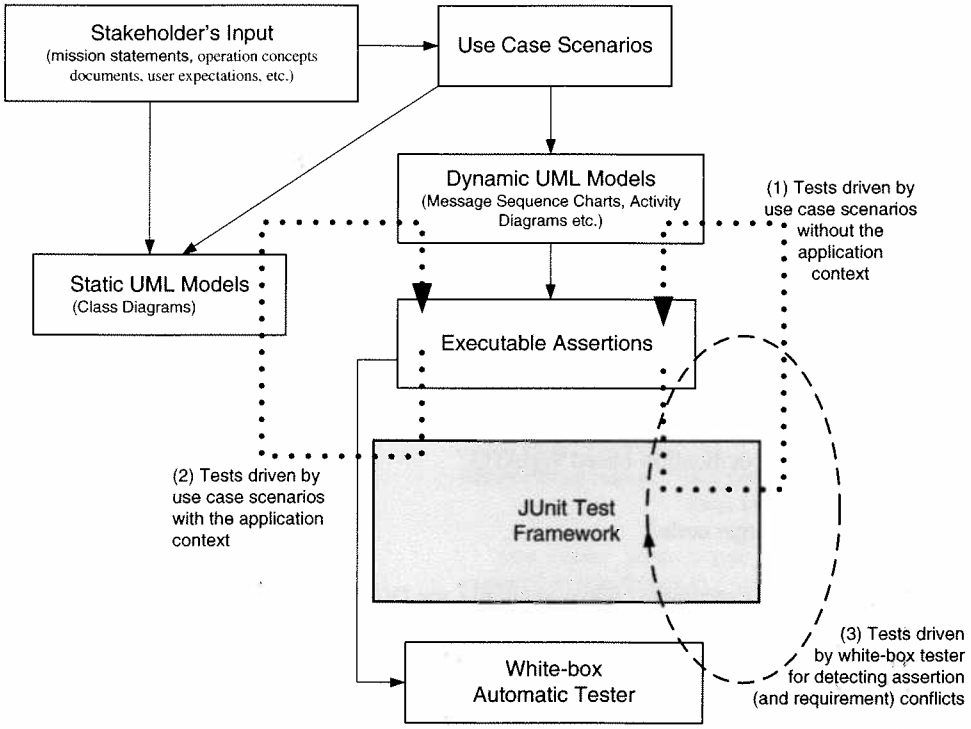

Fig. 3 A process for formal specification and computer-aided validation

of the system's behavior and compares it with the expected behavior per the formal specification. To use RV, the software artifacts produced by the developer need to be instrumented, with the degree of instrumentation being dependent on the software methodologies used by the developer.

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 [6] and NASA's Java Path Finder (JPF) [9]. 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 system's conformance to the formal specification.

With this approach, the IV&V team converts the state-based design models into StateRover statechart (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 (Fig. 4).

The StateRover's automatic white-box tester constructs a JUnit TestCase class from a given statechart model and/or

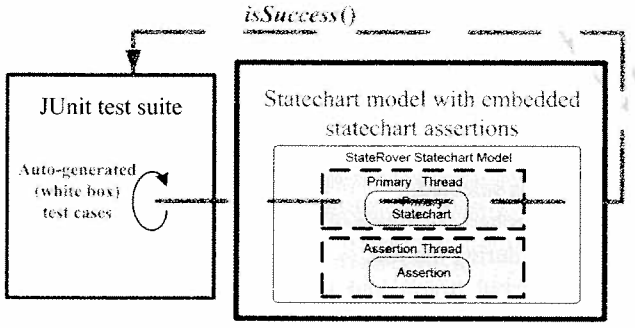

Fig. 4 Execution-based model checking of state-based design models

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

The StateRover generated *WBTestCase* class creates sequences of events and conditions for the SUT. The *WBTestCase* class 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* class 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 Fig. 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 serve as the observers for the RV during the test.

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 tracing the developer-generated requirements to the domain model helps identify the correct naming of the objects and events. These traces of requirements may also help in identifying the critical components of the target system for more thorough testing.

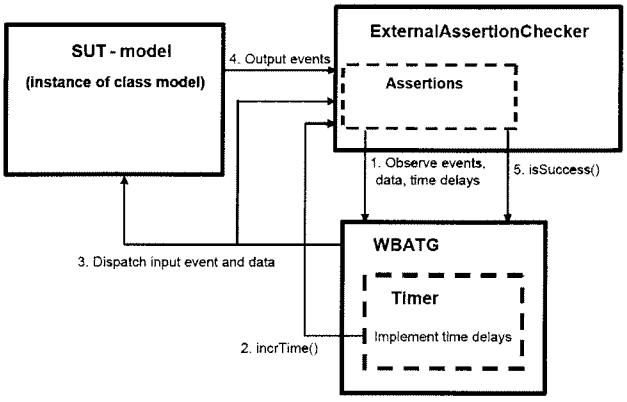

Fig. 5 Automated testing using the system reference model

4 Conclusions

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 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.

Acknowledgments The research was funded in part by grants from the National Aeronautics and Space Administration. The views and conclusions in this article are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Government.

A Appendix

Appendix A.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 cnt ART exceeds 80% of the MAX_COUNT_PER_MIN. The statechart assertion will reset cnt to zero, start the 2-min timer (timer120), and enter the RequireFiftyPercent state if the condition becomes true. The statechart assertion stays in the RequireFiftyPercent state and keeps track of the number of newTrack events for

2 min. 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-min 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. Another analyst could have a different interpretation of the meaning of the track cnt 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 the specifications of system behaviors.

Appendix A.2: Test 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());
    }
}
```

Appendix A.3: Test 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());
}
```

References

1. IEEE (2004) IEEE Std. 1012-2004, IEEE standard for software verification and validation
2. IEEE (1998) IEEE Std. 1233-1998, IEEE guide for developing system requirements specifications
3. IEEE (1990) IEEE Std. 610.12-1990, IEEE standard glossary of software engineering terminology
4. Alexander I (2003) Misuse cases: use cases with hostile intent. IEEE Softw 20(1):58–66
5. Drusinsky D, Michael B, Shing M (2007) 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, August 2007
6. Drusinsky D (2006) Modeling and verification using UML statecharts—a working guide to reactive system design, runtime monitoring and execution-based model checking. Elsevier, Burlington
7. Heitmeyer C (2007) Formal methods for specifying, validating, and verifying requirements. J Univ Comput Sci 13(5):607–618
8. Drusinsky D, Shing M, Demir K (2007) Creating and validating embedded assertion statecharts. IEEE Distrib Syst Online 8(5) art. no. 0705-o5003
9. Havelund K, Pressburger T (2000) Model checking Java programs using Java PathFinder. Intl J Softw Tools Technol Transf 2(4):366–381