Validating UML Statechart-Based Assertions Libraries for Improved Reliability and Assurance[1]

Doron Drusinsky[2], James Bret Michael, Thomas W. Otani, Man-Tak Shing
Department of Computer Science
Naval Postgraduate School
1411 Cunningham Road, Monterey, CA 93943, USA
{ddrusins, bmichael, twotani, shing }@nps.edu

Abstract

In this paper we present a new approach for developing libraries of temporal formal specifications. Our approach is novel in its use of UML statechart-based assertions for formal specifications and its emphasis on validation testing, including an emphasis on the inclusion of validation test scenarios as an integral part of a formal specification library. Validation test scenarios are needed to ensure a robust validation process and to improve the reliability and assurance of the specification and resulting software.

1. Introduction

In our research effort, we have advocated the use of a system reference model (SRM) to capture the modeler's understanding of the problem [1]. The framework incorporates computer-based validation techniques as part of independent validation and verification (IV&V) of software systems. In addition to use cases and standard Unified Modeling Language (UML) artifacts, the SRM contains a set of formal assertions that precisely model the required behavior of the system. The framework allows an IV&V team to capture its own understanding of the problem domain and associated behavioral constraints via an executable SRM using formal assertions to specify mission- and safety-critical behaviors. The framework uses testing to validate the correctness of the assertions with respect to the developer's cognitive expectation as manifested by the natural language requirements.

The effectiveness of executable assertions depends on the ability of the modelers to specify correct assertions. However, correctly defining assertions that describe the system behavior precisely and completely is an error-prone task. No matter how powerful executable assertions may be, their effectiveness is diminished if faulty assertions are used (i.e., the Garbage In Garbage Out - GIGO principle). To reduce the burden on the modelers and to improve the reliability and assurance of the software, we advocate the use of libraries of assertions. Rather than having the modelers always specify assertions from scratch, they can reuse the assertions from such a library. Clearly, effective reuse will help reduce the number of errors modelers introduce when specifying assertions. Moreover, we advocate that assertion-libraries include, in addition to the formal specification assertions, the validation test scenarios originally used to validate each assertion.

The paper is organized as follows. In Section 2 and 3, we describe statechart assertions and their validation process. Section 4 presents the case for including validation test scenarios in the assertions library. We discuss some of the reasons causing modelers to err in their specification of assertions and discuss how the presence of validation test scenarios in the assertions library will help the modelers avoid or detect and rectify such errors. We provide a list of patterns for validation test scenarios in Section 5. Section 6 concludes with a discussion of the future direction of our research on reuse of executable assertions.

[1] The research reported in this article was funded in part by a grant from the National Aeronautics and Space Administration. The views and conclusions contained herein 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. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

[2] Also with Time-Rover Software Inc. www.time-rover.com

978-0-7695-3266-0 2008
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/SSIRI.2008.54

2. Statechart Assertions

In this section, we describe the UML statechart-based temporal assertions for formal specifications. Harel [3] proposed the use of statechart diagrams as a visual approach to modeling the behavior of complex reactive systems such as those found in avionics applications. Statecharts are now one of the standard diagrams in the UML. Drusinsky [2] extended the use of statechart diagrams to specify formal assertions. We differentiate the two by calling the original a *modeling statechart* and the extended version an *assertion statechart*.

Assertion statecharts extend modeling statecharts in two ways: (i) it includes a built-in boolean flag *bSuccess* (and a corresponding *isSuccess* method) that specifies the Boolean status of the assertion (*true* if the assertion succeeds and *false* otherwise) and (ii) an assertion statechart can be nondeterministic. The addition of the *bSuccess* flag makes the assertion statechart suitable as a formal specification, similar to a formal logic such as linear-time temporal logic [5]. Since assertions can be expressed in different ways (e.g., in natural language, formal logic), we use the term *statechart assertions* to differentiate those expressed in statecharts from other types of assertions.

Fig. 1 illustrates a statechart assertion for the following requirement, denoted R1: *When the engine is on, the ignition switch must not be turned on before it is turned off.*

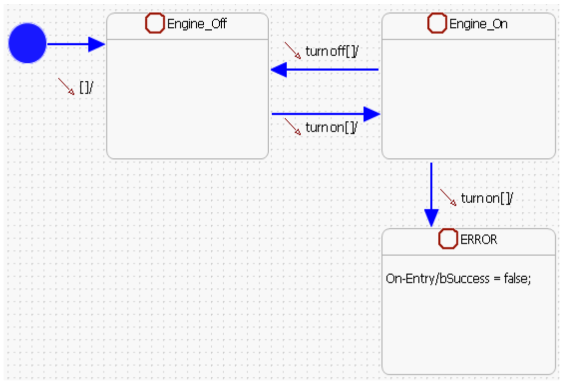


Figure 1. A statechart assertion for requirement R1

By default, an assertion statechart begins with the result flag *bSuccess* being true. It behaves like modeling statechart as described in [3]. When the sample assertion enters the ERROR state (upon the *turnon* event) the on-entry action assigns *bSuccess = false*.

3. Validating Statechart Assertions

To ensure that the statechart assertion correctly represents the intended behavior we run *validation test scenarios* against it. Listing 1 describes two validation test scenarios for the statechart assertion of Fig. 1 (*sa1* is the implementation object for the statechart assertion):

```
sa1.turnon(); //test 1
sa1.turnoff();
assertTrue(sa1.isSuccess());

sa1.turnon(); //test 2
sa1.turnon();
assertFalse(sa1.isSuccess());
```

Listing 1. Two validation scenarios for the statechart assertion of Fig. 1

These two scenarios are implemented as JUnit test cases using the Eclipse-based StateRover tool [6]. Clearly, we would like to ascertain the correctness of this assertion with an adequate set of validation scenarios. We will address this issue in the next section.

In many practical applications, requirements are concerned with real-time constraints. For example, consider the following requirement, denoted R2: *The engine must be turned-off within 10 minutes of being turned-on*.

Fig. 2 illustrates a statechart assertion for requirement R2:

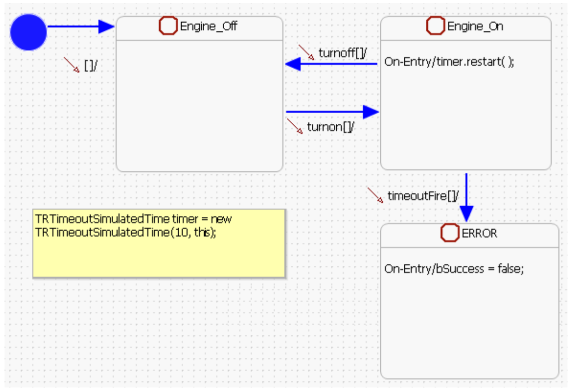

Engine_Off
Engine_On
turnoff[]/
On-Entry/timer.restart( );
[]/
turnon[]/
timeoutFire[]/
TRTimeoutSimulatedTime timer = new TRTimeoutSimulatedTime(10, this);
ERROR
On-Entry/bSuccess = false;

Figure 2. A statechart assertion for requirement R2

The variable declaration box declares a variable named *timer* of type *TRTimeoutSimulatedTime* and set to 10 time units.

The timer is (re)started by the *timer.restart()* action whenever the statechart enters the *Engine_On* state. The timeout event is triggered when the set time limit expires. The timeout event implies a failure because the time limit has expired with no intermediate *turnoff* event.

Listing 2 contains two simple test scenarios for the assertion of Fig. 2, one for testing the good behavior (i.e., `isSuccess() == true`) and the other for the bad behavior (`isSuccess() == false`). *sa2* is the implementation object for the statechart assertion.

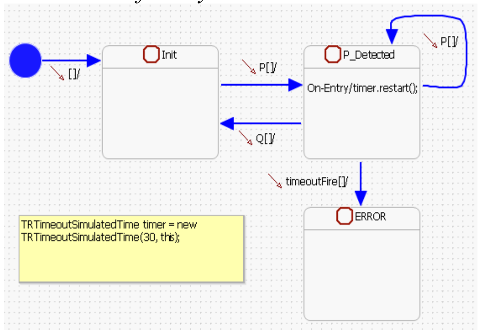
```
sa2.turnon(); //test 1
sa2.incrTime(8);
sa2.turnoff();
assertTrue(sa2.isSuccess());

sa2.turnon(); //test 2
sa2.incrTime(15);
sa2.turnoff(); // too late
assertFalse(sa2.isSuccess());
```

Listing 2. Two validation scenarios for the assertion of Fig. 2

The *incrTime*() method increments the timer for the designated time units.

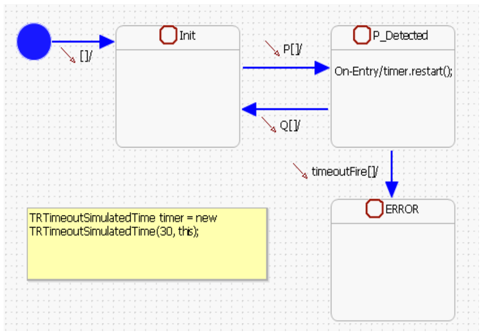
4. Significance of Validation Test Scenarios

Consider the two candidate statechart assertions shown in Fig. 3 for the following natural language requirement, denoted R3*: An event Q must occur within 30 seconds of every event P.*



```
TRTimeoutSimulatedTime timer = new
TRTimeoutSimulatedTime(30, this);
```

a. Statechart assertion A



```
TRTimeoutSimulatedTime timer = new
TRTimeoutSimulatedTime(30, this);
```

b. Statechart assertion B

Figure 3. Two candidate statechart assertions for requirement R3

The two statechart assertions of Fig. 3 are similar. The only difference is the existence of the self-loop in the first statechart assertion from the *P-Detected* state back to itself. A good validation test-suite should be capable of distinguishing between the two statechart assertions and identifying the correct statechart assertion. Clearly, we should execute meaningful tests so to ensure that the assertion is correct. The test scenario of Listing 3 does not distinguish between the statechart assertions of Fig. 3 because it succeeds on both:

```
//test 1 - same result
sa.P();
sa.incrTime(20);
sa.Q();
assertTrue(sa.isSuccess());
```

Listing 3. A validation scenario that does not distinguish between the statechart assertions of Fig. 3

The test scenario of Listing 4, however, does distinguish between those two statechart assertions:

```
//test 2 - different results
sa.P();
sa.incrTime(20);
sa.P();
sa.incrTime(20);
sa.Q(); // too late for first P
assertFalse(sa.isSuccess());
```

Listing 4. A distinguishing validation scenario for the statechart assertions of Fig. 3

We expect a correct statechart assertion to fail this test because it violates R3: the Q event occurs 40 time-units after the first P event, which is too late per R3. At the end of the test, *bSuccess* is false in Assertion B, while it is true in Assertion A. Hence, this test scenario managed to identify an error in assertion A.

We identify two kinds of errors in statechart assertions: (i) implementation errors resulting from mistakes in the statechart assertion, as the case for Assertion A, and (ii) errors or ambiguities in the natural language statements. For example, should the scenario "*R occurs 10 time-units after P without any Q in between*" violate the requirement: *After P occurs, Q must occur once every 30 time-units until R occurs*? Regardless of the type of error, running the test scenarios provides a means to detect errors [4].

5. Patterns for Test Scenarios

This section describes patterns, or types of necessary validation scenarios. The purpose of this section is to highlight types of patterns that must be part of every validation test-suite. The list is not presented as a complete list; rather, it highlights types of scenarios that are often overlooked during validation testing. Moreover, the patterns are not mutually exclusive and are often combined to create additional scenarios.

1. *Obvious success*. For example, the scenario of Listing 3 trivially conforms to requirement R3; clearly, we should expect the statechart assertion being validated to succeed on such a test.

2. *Obvious failure*. For example, consider a variation of the scenario of Listing 3 where the delay (*incrTime*) is of 35 time units. Clearly, this scenario fails requirement R3; we should therefore expect a statechart assertion being validated to fail on such a test.

3. *Event repetitions*. A common error found in temporal assertions is the failure in handling event repetitions properly. For example, consider the following requirement, denoted R4: *An event Q must occur between an event P and an event R*. A validation scenario with event repetitions is depicted in Fig. 4. Note how it fails requirement R4 because Q does not occur between the third P event and the R event. By using such a validation test pattern we assure that an assertion is not written in a manner that only observes the first P in a sequence of P's.

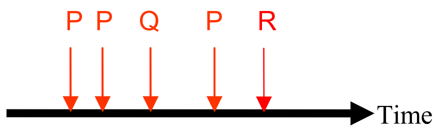


Figure 4. An event-repetition scenario

Note that, depending on the actual assertions, this pattern may need to include scenarios with repeating Q's or R's.

The next two patterns address the presence of timing constraints in temporal assertions.

4. *Multiple time intervals.* Another common assertion modeling error is the failure to handle multiple time intervals or scenarios. For example, the scenario depicted in Fig. 5 consists of two 30 time-unit intervals. This scenario fails requirement R3 because Q does not occur within the second interval as required by R3. By using such a validation test pattern we assure that an assertion is not written in a manner that observes only a single time interval or a single scenario.

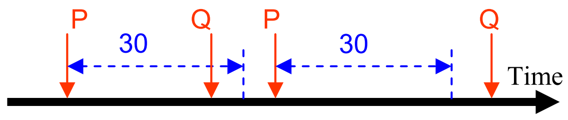


Figure 5. A time-interval repetition

5. *Overlapping time intervals*. Another common assertion modeling error is the failure to handle overlapping time intervals within a scenario. For example, consider the following requirement, denoted R5: *An event Q may not occur within 30 seconds of any event P*. The scenario depicted in Fig. 6 consists of two overlapping 30 time-unit intervals. This scenario fails requirement R5 because Q occurs within the second interval.

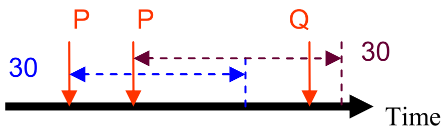


Figure 6. A time-interval-with-overlap scenario

Note that this list is not comprehensive and the patterns can be combined to create additional scenarios. For example, consider the scenario depicted in Fig. 7; it superimposes a triggering-event-repetitions scenario on an overlapping-time-intervals scenario of Fig. 6.

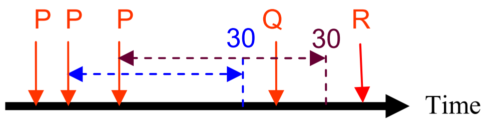


Figure 7. A combination scenario

6. Conclusion

In this paper, we presented our initial results pertaining to the construction of libraries of reusable formal specification assertions. Key to our approach is the inclusion of validation test scenarios as an integral part of a specification library, for use in ensuring the correctness of statechart assertions. We argue that formal specification assertions must always be accompanied by a set of validation test scenarios. Without such a test suite it is difficult to interpret the original developer's intent, thus undermining the ability to reuse the assertions in the library. A good set of test scenarios also helps us understand how the developer resolved ambiguities in the natural language specification. This concept adheres to the tenet of test-driven software development of providing a test suite along with the software.

In addition, we provided necessary types of validation test scenarios. Our long-term goal is the development of an assertion library that includes a rich set of validation test scenarios.

7. References

[1] D. Drusinsky, J. B. Michael, and M. Shing, "A Framework for Computer-Aided Validation," to appear in the journal *Innovations in System and Software Engineering*, Springer, London, ISSN 1614-5046.

[2] D. Drusinsky, *Modeling and Verification Using UML Statecharts - A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, Elsevier, 2006, ISBN 0-7506-7949-2.

[3] D. Harel, "Statecharts: A Visual Approach to Complex Systems", *Science of Computer Programming*, Vol. 8, No. 3, 1987, pp. 231-274.

[4] D. Drusinsky, M. Shing and K. Demir, "Creation and Validation of Embedded Assertion Statecharts", *Proc. 15th IEEE International Workshop in Rapid Systems Prototyping*, Chania, Greece, 14-16 June 2006, pp. 17-23.

[5] A. Pnueli, "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. on Foundations of Computer Science*, Providence, Rhode Island, 31 October-2 November 1977, pp. 46-57.

[6] The StateRover, Timer-Rover, Inc. http://www.time-rover.com