2006

Creation and Validation of Embedded Assertion Statecharts

Drusinsky, Doron; Shing, Man-Tak; Demir, Kadir Alpaslan

Drusinsky, Doron, M-T. Shing, and Kadir Alpaslan Demir. "Creation and validation of embedded assertion statecharts." Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on. IEEE, 2006.
https://hdl.handle.net/10945/60582

Creation and Validation of Embedded Assertion Statecharts

Doron Drusinsky, Man-Tak Shing and Kadir Alpaslan Demir
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943, USA
{ddrusins, shing, kdemir}@nps.edu

Abstract

This paper addresses the need to integrate formal assertions into the modeling, implementation, and testing of statechart based designs. The paper describes an iterative process for the development and verification of statechart prototype models augmented with statechart assertions using the StateRover tool. The novel aspects of the proposed process include (1) writing formal specifications using statechart assertions, (2) JUnit-based simulation and validation of statechart assertions, (3) JUnit-based simulation and testing of statechart prototype models augmented with statechart assertions, (4) automatic, JUnit-based, white-box testing of statechart prototypes augmented with statechart assertions, and (5) spiral adjustment of model and specification using the test results. We demonstrate the proposed process with a prototype of a safety-critical computer assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump.

1 Introduction

Harel Statecharts are commonly used for design analysis and implementation; for example, Brugge suggests using statecharts in the design analysis phase of an object oriented UML based design methodology to specify dynamic behavior of complex reactive systems [3]. Studies have suggested that the process of formally specifying requirements enables developers to gain a deeper understanding of the system being specified, and to uncover requirements flaws, inconsistencies, ambiguities and incompletenesses [6]. In order to improve the clarity and precision of the system requirements, system designers often incorporate formal assertions into statechart design by augmenting statechart models with other formalisms such as process algebra [11], symbolic timing diagrams [9] and temporal logic [7], and demonstrate the correctness of the statechart design with formal methods (e.g. theorem prover, static model-checker, or execution-based model checker) on the corresponding assertions. In [4], Drusinsky presented a new formalism that combines UML-based prototyping, UML-based formal specifications, run-time monitoring, and execution-based model checking. The approach is supported by the StateRover, a design entry, code generation, and visual debug animation tool for UML statecharts combined with flowcharts. The new formalism and tool allow system designers to embed deterministic and non-deterministic statechart requirement assertions in statechart designs and to execute the assertions in tandem with their primary UML statechart to provide run-time monitoring and run-time recovery from assertion failures.

This paper is concerned with the correct development and early use of statechart assertions in rapid system prototyping. We shall illustrate the process with a statechart design of the safety-critical computer assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump. The rest of the paper is organized as follows. Section 2 presents a statechart design of the CARA software. Section 3 describes the process for the development and validation of statechart assertions. Section 4 presents the prototype that combines the statechart design and the statechart assertions, and the testing of the prototype using the StateRover tool. Section 5 presents a discussion on the approach and draws some conclusions.

2 The CARA Statechart Design

CARA is a safety-critical software developed by the Walter Reed Army Institute of Research to improve life support for trauma cases and military casualties; it has been used as a case study by several software engineering research groups [1, 5, 10]. CARA's mission is to monitor a patient's blood pressure and to automatically administer intravenous (IV) fluids via computer-controlled pump at levels required to restore intravascular volume and blood pressure.

2.1 The CARA Statechart Prototype

Figure 1 shows the top-level statechart of the CARA software. It consists of three concurrent threads, *Main,*

Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)
0-7695-2580-6/06 $20.00 © 2006 IEEE

MonitorPlugIn and *MonitorOcclusion.* The top-level statechart of the *Main* thread is made up of two states, the *Manual* state and a composite state named *SoftwareControl,* which consists of three sub-states: *AutoReady*, *AutoControl* and *AutoFail*.

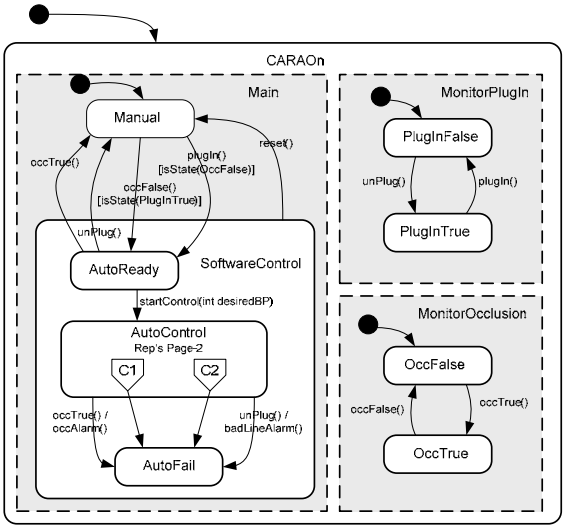

Figure 1. Top level page of the CARA statechart

CARA monitors the pump connector on the Life Support for Trauma and Transport (LSTAT) unit to determine when a pump is powered on and plugged in. When CARA determines that (i) the pump is plugged in and (ii) the occlusion line is clear, it transitions into the *AutoReady* state.

CARA will transition back to the *Manual* state if it receives the *unplug* or the *occTrue* signals from the environment while it is in the *AutoReady* state. If CARA receives the *start_control(desired_bp)* message from the environment while it is in the *AutoReady* state, it will enter the *AutoControl* state, which is made up of three concurrent threads shown in Figure 2.

While in the *AutoControl* sub-state, CARA continuously checks for continuity on all wires going to the pump, and will sound a level-one alarm and enter the *AutoFail* state if it receives the *unplug* or the *occTrue* signals from the environment. Note that refined details of this state are available in another portion (page) of the diagram file, depicted in Figure 2.

A clock interrupt triggers an event at precise five-second intervals signaling the necessity to check the back electromotive force (EMF) voltage as shown in the *MonitorEMF* component in Figure 2. It will sound a level-one alarm and transition to the *AutoFail* state whenever the back EMF reading is zero or cannot be obtained.

CARA will attempt to use blood pressure from various sources as the input for the CARA algorithm to control the pump. For simplicity, we use only one blood pressure source in this version of the prototype - an arterial line sensor, which is an active device with a sampling rate of 1Hz. CARA adjusts the patient's blood pressure by regulating the voltage to drive the pump after each blood pressure reading. CARA will signal a *lowBPAlarm* if the blood pressure reading is below a pre-set minimum critical value, and will maintain a *keep-vein-open* rate at or above the threshold of 4 ml/min when *desiredBP* is reached. While under the arterial line pressure control, if the arterial line signal is lost for more than 1 minute, CARA sounds a level-one alarm and enters the *LoseBP* state of the *ControlPump* component shown in Figure 2. If the arterial line signal is lost for another 2 minutes, a level-two alarm sounds and CARA transitions to the *AutoFail* state.

While in *SoftwareControl*, a "*reset*" event from the external environment will cause CARA to reset its alarms and transition back to the *Manual* state.

2.2 Generation and Testing of the Target Code

The StateRover's code generator generates one Java controller class for each statechart file. In our case study, we have one statechart diagram file consisting of two pages, with the top-level statechart in the first page (Figure 1) and the *AutoControl* sub-statechart in the second page (Figure 2). The StateRover's code generator automatically connects the two statecharts into a single statechart and generates a single *CARA* class for the executable prototype. The controller class consists of a set of event handlers (one per transition event), the central event dispatcher *execTReventDispatcher*, and the source code for local variable declarations and methods supplied by the users via the dialog boxes of StateRover's statechart editor. In addition, the code generator also generates a Java interface, named *CARAIF*, to allow the test drivers or other systems from the external environment to interact with the CARA prototype.

The StateRover's vanilla code generator implements statechart orthogonality using a fixed schedule. For example, the three orthogonal *occTrue()* transitions shown in Figure 1, two in the *Main* thread and one in the *MonitorOcclusion* thread, will be realized as three if-blocks within the *occTrue()* event handler. The order of these if-blocks induces a fixed firing schedule for corresponding transitions. In addition to the vanilla code generator described above, the StateRover has a *concurrent code generator* that generates multi-threaded Java code for statecharts with Harel-concurrence.

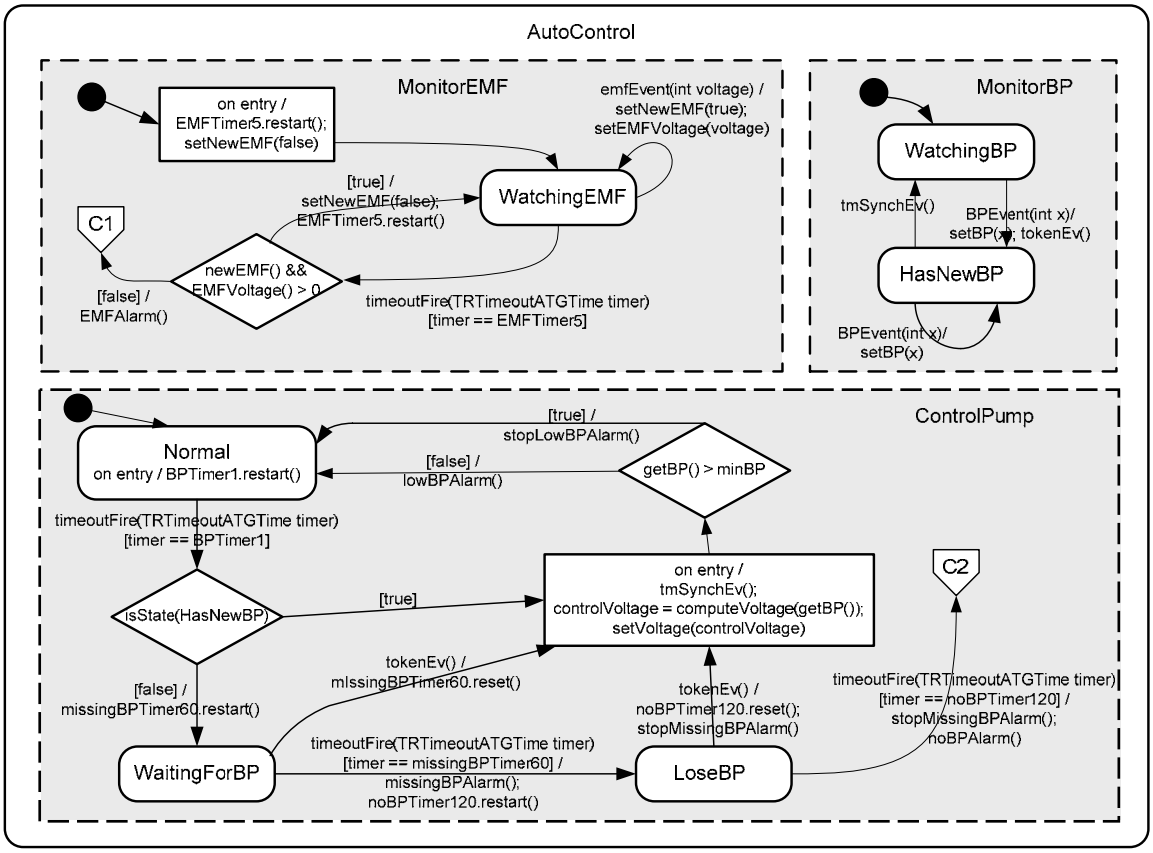

Figure 2. Detailed AutoControl sub-state of the CARA statechart

The generated code is designed to work with the JUnit Test Framework [2]. Use Case scenarios used by the system designers to identify user needs and system requirements are hand-coded as JUnit test cases and exercised against the generated statechart code. For example, the following test case describes a scenario in which CARA enters the *AutoControl* state after receiving the events *plugIn()*, *occFalse()*, *startControl*(), and eventually ends up in the *AutoFail* state after receiving the events *BPEvent()*, *BPEvent*() and *occTrue*() :

```
import junit.framework.*;

public class TestCARA1 extends TestCase {
  private CARA cara = null;

  public TestCara1(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    cara = new CARA();
  }

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

  // Test Scenario:
  public void testExecTReventDispatcher() {
    cara.plugIn();
    cara.occFalse();
    cara.setTime(30);  //advance clock to 30s
    this.assertTrue(cara.isState("AutoReady"));
    cara.startControl(70);
    cara.incrTime(70); //advance clock to 100s
    cara.BPEvent(50);
    cara.incrTime(50); //advance clock to 150s
    cara.BPEvent(52);
    cara.incrTime(50); //advance clock to 200s
    cara.occFalse();
    this.assertTrue(cara.isState("AutoFail"));
  }
}
```

3 Development and Validation of the Statechart Assertion Correctness

Typically, formal specifications are created from a conceptual requirement as understood by the primary modeler. Regardless of what formal notations or formal methods were used, the system modelers always start their requirements discovery process based on some scenarios involving the system and its environment, and express their understanding of the expected behavior or properties of the system informally with natural languages. For example, we may come across a scenario

where there is a need for the CARA software to keep the IV line open while under its control. We first express the requirements in English

> "*Whenever CARA receives the startControl() event, it must generate a control voltage that is greater than or equal to the Keep-Vein-Open (KVO) voltage within one minute. The voltage level condition will be examined once every second and should be sustained until the reset() event is received.*"

and then translate the assertion into the statechart shown in Figure 3.

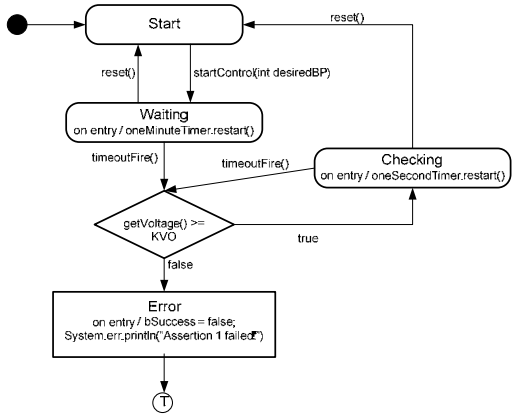

Figure 3. The assertion statechart

It is important to validate the correctness of the assertions early in the software development process. Unfortunately, users often discover, late in the development process, that their assertions are incorrect and do not work as intended. Possible reasons for incorrect assertions are:

1. Incorrect translation of the natural language specification to a formal specification.
2. Incorrect translation of the requirement, as understood by the modeler, to natural language.
3. Incorrect cognitive understanding of the requirement. This situation typically occurs when the requirement was driven from the use case's main success scenario, with insufficient investigation of other scenarios.

Hence, we propose the following iterative process for assertion development (Figure 4).

We will first test the behavior of the assertion described by the statechart shown in Figure 3 with the following scenario.

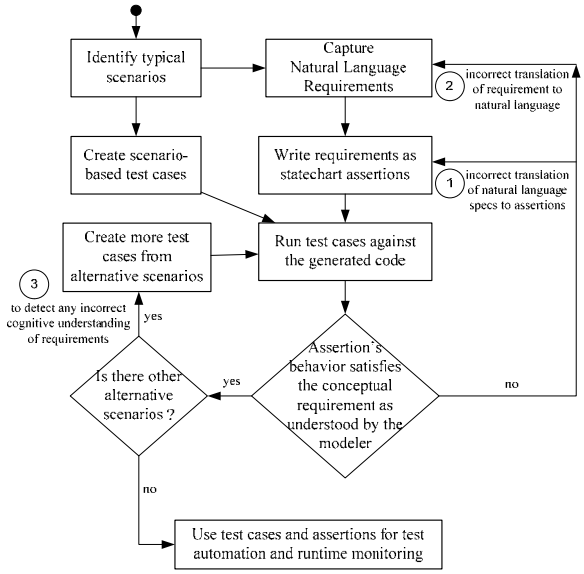

Figure 4. Iterative process for assertion development

```
import junit.framework.*;

public class TestAssertion extends TestCase {
  private Assertion1 assert1 = null;

  public TestAssertion(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    assert1 = new Assertion ();
  }

  protected void tearDown() throws Exception {
    assert1 = null;
    super.tearDown();
  }

  // Assertion 1, Test Scenario 1
  public void testExecTReventDispatcher() {
    assert1.startControl(70);
    assert1.incrTime(30); //advance clock to 30s
    assert1.setVoltage(KVO);
    assert1.incrTime(30); //advance clock to 60s
    assert1.incrTime(1);  //advance clock to 61s
    assert1.setVoltage(KVO + 1);
    assert1.incrTime(1);  //advance clock to 62s
    assert1.reset();
    this.assertTrue(assert1.isSuccess());
  }
}
```

Scenario 1 represents a typical case where the control voltage is set at a level greater than or equal to the KVO voltage within 1 minute after the arrival of the *startControl()* event and remain greater than or equal to the KVO voltage until the *reset()* event is received, resulting in a successful test outcome.

In order to make sure that the assertion works as intended, we create two more scenarios by replacing the body of the *testExecTReventDispatcher()* method with the following code.

```
// Assertion 1, Test Scenario 2
public void testExecTReventDispatcher() {
  assert1.startControl(70);
  assert1.incrTime(30); //advance clock to 30s
  assert1.setVoltage(KVO);
  assert1.incrTime(30); //advance clock to 60s
  assert1.incrTime(1);  //advance clock to 61s
  assert1.setVoltage(0);
  assert1.incrTime(1);  //advance clock to 62s
  assert1.reset();
  this.assertTrue(assert1.isSuccess());
}

// Assertion 1, Test Scenario 3
public void testExecTReventDispatcher() {
  assert1.startControl(70);
  assert1.incrTime(30); //advance clock to 30s
  assert1.setvoltage(0);
  assert1.reset();
  this.assertTrue(assert1.isSuccess());
}
```

Scenario 2 represents the case where the control voltage is set at a level greater than or equal to the KVO voltage within 1 minute after the arrival of the *startControl()* event, but fails to sustain the voltage level condition before the *reset()* event is received. The entry action in the *Error* flowchart box in Figure 3 sets the variable *bSuccess* to false, which in turn causes *assert1.isSuccess()* to return false and *this.assertTrue()* to fail.

Scenario 3 presents an interesting case. Although the control voltage is kept below KVO voltage all the time, the test outcome still turns out to be successful. This behavior relates directly to the process described in Figure 4 as follows: initially, the developer of the assertion was surprised by the assertion's success for this scenario. He then followed the process illustrated in Figure 4 and checked whether this behavior represents (1) an incorrect assertion realization of the natural language requirement, (2) incorrect or ambiguous natural language requirement, or (3) incorrect or ambiguous cognitive expectation. Finally our developer decided that this is an acceptable behavior. Other developers might have concluded otherwise and would need to adjust their natural language requirement and assertion statechart accordingly.

This example highlights the subtleties in creating correct formal assertions and the value of testing executable formal assertions via JUnit-based simulations. In fact, we argue that the test suite for an assertion is an integral part of the assertion's deliverables.

4 Integrating Assertions to the Statechart Design

Figures 5 and 6 shows the combined CARA statechart with embedded statechart assertion, where the *Assertion* statechart shown in Figure 3 now becomes a sub-statechart of the *AutoControl* sub-state shown in Figure 6. In addition, an unlabeled transition from the *Assertion* sub-statechart (Figure 6) to the *Manual* state in the top-level statechart (Figure 5) is added to enable run-time recovery. Whenever the assertion fails, it reaches the terminal state *T* (in Figure 3) and will therefore cause the unlabeled transition out of the Assertion sub-statechart to fire, forcing CARA to leave *AutoControl* and returns to the *Manual* state.

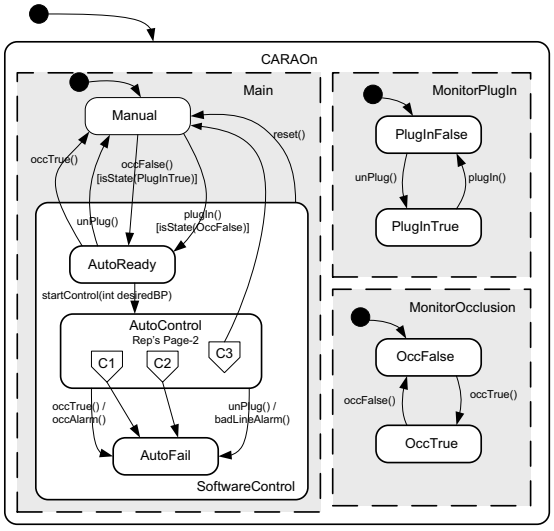

Figure 5. Top level page of the combined CARA statechart

4.1 Testing the Combined Prototype

Although the *TestCARA1* test case described in Section 2.2 resulted in a successful test outcome for the prototype generated from the statecharts shown in Figures 1 and 2, running the same *TestCARA1* test case against the prototype generated from the CARA statecharts shown in Figures 5 and 6 resulted in an unsuccessful test outcome due to failure of the *Assertion* sub-statechart. A close inspection of the execution trace reveals that the assertion was violated because CARA does not generate any control voltage until it receives the first *BPEvent()* 70 seconds after the *startControl()* event in the *TestCARA1* test case scenario. To fix the problem, an entry action "*setVoltage(KVO)*" is added to the *AutoControl* super-state to make sure that the control voltage is set to the KVO voltage once CARA receives the *startControl()* event.

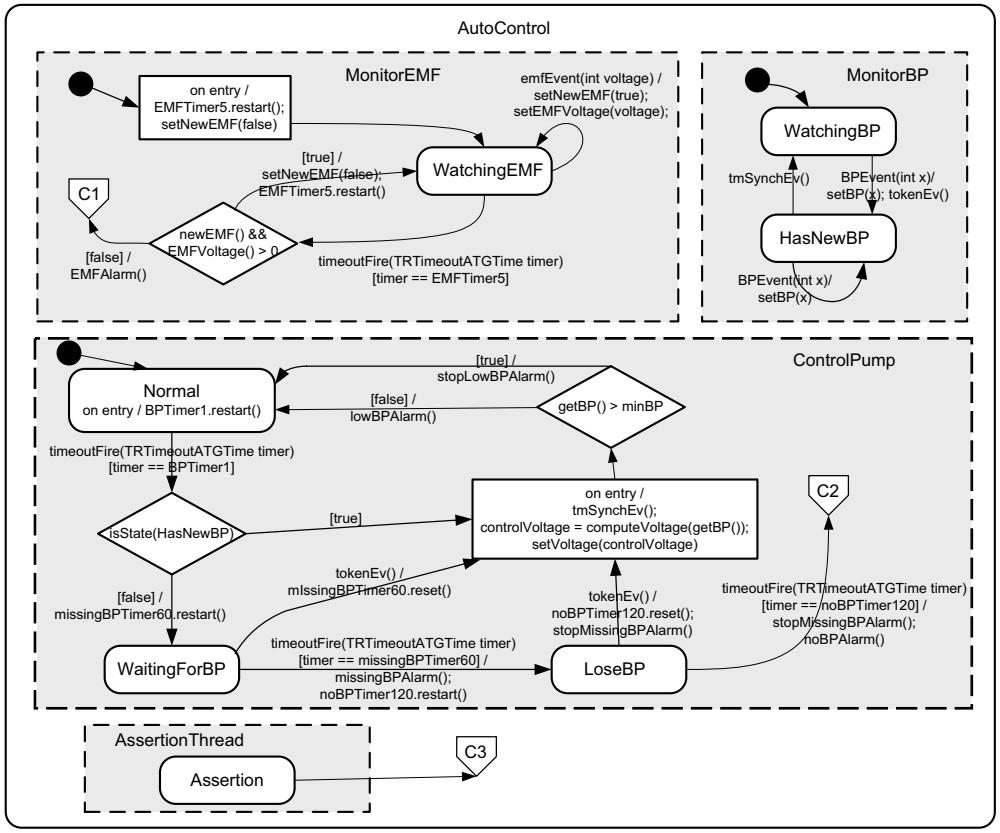

Figure 6. The AutoControl sub-state of the combined CARA statechart with embedded statechart assertion

4.2 Automatic White Box Testing of the Combined Prototype

The StateRover's automatic white box tester constructs a JUnit TestCase class from a given statechart model and associated embedded assertions. The JUnit test case executes a large volume of *test runs* of the statechart-under-test (SUT). A typical white box test case consists of hundreds of thousands of runs of the SUT. The availability of the executable statechart assertions makes the automatic checking of test results possible and cost-effective.

In order 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 mechanism helps developers to efficiently eliminate errors in their design.

5 Discussions and Conclusion

In this paper, we presented an iterative process for developing statechart assertions and using the assertions to verify statechart prototype designs early in the software development process. The proposed process has the following novelties:

(1) Writing formal specifications using statechart assertions. It is easier for system designers to create and understand statechart assertions than text-based temporal assertions because statechart assertions are visual, intuitive, and resemble statechart design models. For example, statechart assertions are event driven just like statechart models, while temporal logic is purely propositional. Moreover, statechart assertions are Turing equivalent and are therefore significantly more expressive than temporal logic

(2) JUnit-based simulation and validation of statechart assertions. The ability to test the statechart assertions independent of the prototype design ensures that system designers truly understand the required system behavior without being tainted by any pre-conceived solutions. With the help of StateRover's code generator, we can create a library of executable assertion patterns consisting of generic statechart assertions and the accompanying scenario-based test cases. The use of pre-tested generic statechart assertions will lessen the development time and improve the quality of the statechart assertions in rapid prototyping.

(3) JUnit-based simulation and testing of statechart prototype models augmented with statechart assertions. Quite often, subtle timing properties can only be studied with simulation and runtime execution monitoring. The availability of the StateRover code generator and the JUnit test framework makes the rapid prototyping and testing of the statechart prototype augmented with statechart assertions possible and cost-effective.

(4) Automatic, JUnit-based, white-Box testing of statechart prototypes augmented with statechart assertions. The white-box tester is both model-based and specification-based because it uses information from the SUT as well as embedded assertions for test generation. The StateRover white-box test generator is intelligent; it generates only test scenarios that actually affect the statechart SUT or one of its embedded assertions.

(5) Spiral adjustment of model and specification using the test results. This paper points out the subtleties in creating correct formal assertions and the value of testing executable formal assertions via JUnit-based simulations. The proposed iterative process helps ensure the correctness of formal requirements per the modeler's expectations early in the development process.

The StateRover is commercially available and is being used by the U.S. Ballistic Missile Defense System project to design and verify the new BMDS battle manager because of its ability to scale, and its support for temporal assertions that include real-time and time series constraints. The StateRover's automatic white-box tester has been extended to generate code for NASA's Java Path Finder (JFP) [8], which uses a customized Java Virtual Machine to detect the presence of concurrency error under varying firing schedules of concurrent transitions and actions.

Acknowledgements

The authors would like to thank the anonymous reviewers for their very helpful comments. The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. 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.

References

[1] R. Alur, D. Arney, E. Gunter, I. Lee, W. Nam and J. Zhou, "Formal Specifications and Analysis of the Computer Assisted Resuscitation Algorithm (CARA) Infusion Pump Control System", *Proc. Integrated Design and Process Technology (IDPT)*, 2002.

[2] K. Beck and E. Gamma, "Test infected: Programmers love writing tests", *Java Report*, 3(7), pp. 37-50, 1998.

[3] B. Bruegge, *Object-Oriented Software Engineering: Using UML, Patterns, and Java (2nd ed.)*, Prentice Hall, 2004, ISBN 0-13-0471100.

[4] 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.

[5] D. Drusinsky and M. Shing, "TLCharts: Armor-plating Harel Statecharts with Temporal Logic Conditions", *Proceedings of the 15th IEEE International Workshop in Rapid Systems Prototyping*, Geneva, Switzerland, pp. 29-36, June 28-30, 2004.

[6] S. Easterbrook, R. Lutz, R. Covington, J. Kely, Y. Ampo and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling", *IEEE Transactions on Software Engineering*, 24(1), pp. 4-11, Jan 1998.

[7] G. Graw, P. Herrmann, and H. Krumm, "Verification of UML-Based Real-Time System Design by Means of cTLA", *Proc. 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pp.86-95, 15-17 March 2000.

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

[9] K. Lüth, J. Niehaus and T. Peikenkamp, "HW/SW Co-synthesis using Statecharts and Symbolic Timing Diagrams", *Proc. 9th International Workshop on Rapid System Prototyping*, pp.212-217, 3-5 June 1998.

[10] Luqi, M. Shing, J. Puett, V. Berzins, Z. Guan, Y. Qiao, L. Zhang, N. Chaki, X. Liang, W. Ray, M. Brown, and D. Floodeen, "Comparative Rapid Prototyping, A Case Study", Proc. *14th IEEE International Workshop in Rapid Systems Prototyping*, pp. 210-217, 9-11 June 2003.

[11] M.H. Park, K.S. Bang, J.Y. Choi and I. Kang, "Equivalence Checking of Two Statechart Specifications", *Proc. 11th International Workshop on Rapid System Prototyping*, pp.46-51, 21-23 June 2000.