Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering
Los Angeles, CA, USA - April 2006

Quality Assurance of the Timing Properties of Real-time, Reactive System-of-Systems

Man-Tak Shing
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
shing@nps.edu

Doron Drusinsky
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
and Time Rover, Inc.
Cupertino, CA, USA
ddrusins@nps.edu, www.time-rover.com

Thomas S. Cook
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
tscoo1@nps.edu

Abstract - *The paper concerns the quality assurance of the timing properties of complex, real-time, reactive system-of-systems. It builds upon our previous work on run-time model checking of timing properties and the automatic white-box testing based on run-time assertion checking, and brings together several technologies to improve the predictability of the system-of-systems' logical and timing behavior. The paper presents a microkernel architecture for evolvable system-of-systems to isolate the computations that are likely to change with time from the basic control logics that are invariant in the application domain, and a testing methodology that is based on formal statechart assertions. We demonstrate the approach with a conceptual design of a ballistic missile defense battle management software.*

Keywords: White-box testing, design by contract, statechart assertions, formal specification, run-time execution monitoring.

1 Introduction

Large systems-of-systems (SoSs), like the network-centric C4 systems and the global ballistic missile defense systems (BMDS), are typically large, heterogeneous and distributed. Each SoS is made up of a federation of existing and developing systems to provide an enhanced capability greater than that of any of the individual systems within the system-of-systems. The individual systems making up of a SoS (i.e., the component systems) are often developed for a different context and subjected to a different set of constraints than those of the system-of-systems. When assembled together as a SoS, these component systems are expected to work together to provide additional services beyond what they were originally designed to do.

Many of the SoSs are real-time, reactive systems. These systems are very complex and yet have to be highly dependable. Quality assurance is a dominant issue for these real-time, mission critical SoSs. Some of the component systems have to continuously interact with their environment under tight timing constraints. Both the inputs and outputs of these component systems must satisfy new timing constraints imposed by the SoS requirements, which may not be present in the original functional requirements of the component systems. The consequences of these component systems, like the ballistic missile defense battle management software, missing their timing requirements (i.e. not performing its intended function in a timely fashion) could be the loss of thousands or perhaps tens of thousands of innocent lives. Clearly, such system must be of the highest quality. Hence, in addition to the typical interoperability problem that may exist among the legacy components, the SoS developers now have the difficult task of verifying the correctness of the SoS's timing behavior. Moreover, the individual component systems and the SoS as a whole are likely to constantly evolve with time. Traditional testing methods that rely on hand-crafted test codes are too tedious and time-consuming to cope with the frequent changes in SoS code.

This paper addresses the need to verify the timing properties of real-time, reactive SoSs. It presents a testing methodology that builds upon our previous work on run-time model checking of timing properties and the automatic white-box testing based on run-time assertion checking [4]. Run-time Execution Monitoring of formal specification assertions (REM) is a class of methods for tracking the temporal behavior, often in the form of formal specification assertions, of an underlying application. REM methods range from simple print-statement logging methods to run-time tracking of complex formal requirements (e.g., written in temporal logic or as statechart assertions) for verification purposes. NASA used REM for the verification of flight code for the Deep Impact project [8]. In [7], we showed that the use of run-time monitoring and verification of temporal assertions, in tandem with rapid prototyping, helps debug the requirements and identify errors earlier in the design process. Recently, REM has been adopted by the

1-4244-0188-7/06/$20.00 ©2006 IEEE

U.S. Ballistic Missile Defense System project as the primary verification method for 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 [2].

The rest of the paper is organized as follows. Section 2 provides an overview of the StateRover statechart assertion formalism [6]. Section 3 presents a microkernel architecture for the real-time, reactive SoS and describes the process for testing such systems. Section 4 presents a discussion on the approach and draws some conclusions.

2 The StateRover Statechart Assertions

Harel Statecharts [10] are commonly used in the design analysis phase of an object oriented UML based design methodology to specify the dynamic behavior of complex reactive systems. In [5, 6], Drusinsky presented a new formalism that combines UML-based prototyping, UML-based formal specifications, run-time monitoring, and execution-based model checking. The new formalism is supported by StateRover, a commercially available tool from the Time Rover Inc. StateRover provides support for design entry, code generation, and visual debug animation for UML statecharts combined with flowcharts. The new formalism and tool allow system designers to embed deterministic and non-deterministic statechart assertions in statechart designs and execute the assertions in tandem with their primary UML statechart to provide run-time monitoring and run-time recovery from assertion failures.

2.1 A statechart example

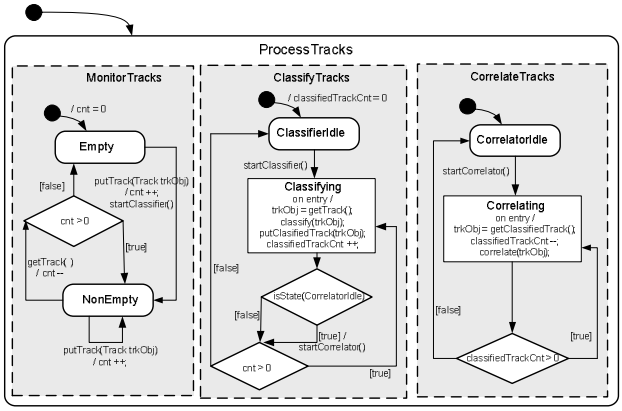

Figure 1. Track processing module statechart

For example, Figure 1 shows a design of the simplified track processing module of a missile defense battle manager. It consists of three concurrent threads, *MonitorTracks*, *ClassifyTracks* and *CorrelateTracks*. The *MonitorTracks* thread places all incoming track data into the Track Data Store, and forwards them to the *ClassifyTracks* thread for classification. The *ClassifyTracks* thread retrieves the track data from the Track Data Store one at a time, invokes the *classify()* method of a Classifier component to determine the track's classification, and places the classified track in the Classified Track Data Store for further processing. The *CorrelateTracks* thread retrieves the track data from the Classified Track Data Store one at a time and invokes the *correlate()* method of a Correlator component to update the Correlated Track database.

2.2 Statechart assertions

Studies have suggested that the process of specifying requirements formally enables developers to gain a deeper understanding of the system being specified, and to uncover requirements flaws, inconsistencies, ambiguities and incompletenesses [9]. StateRover uses deterministic and non-deterministic statecharts as assertions for timing properties as well as correct ordering and sequencing of events and responses. For example, Figures 2 and 3 show the statechart assertions for the following requirements.

Assertion 1:

The time spent for classifying a track must not exceed 1 second.

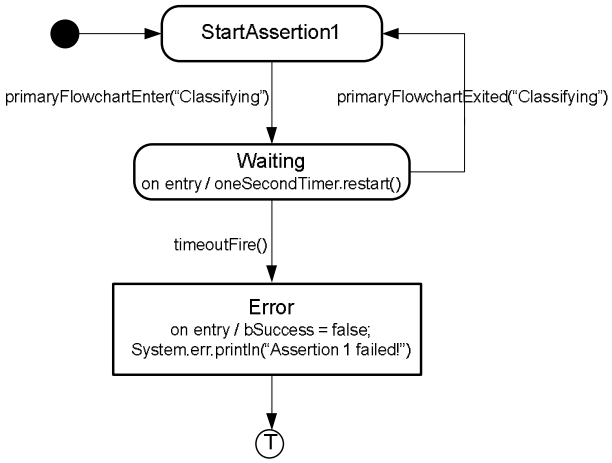

Figure 2. Assertion1 Statechart

Assertion 2:

Whenever the track count (*cnt*) exceeds 75% of the MaxCount, *cnt* must be reduced back to 50% of the MaxCount within 1 minute and must remain at or below 50% of the MaxCount for at least 10 minutes.

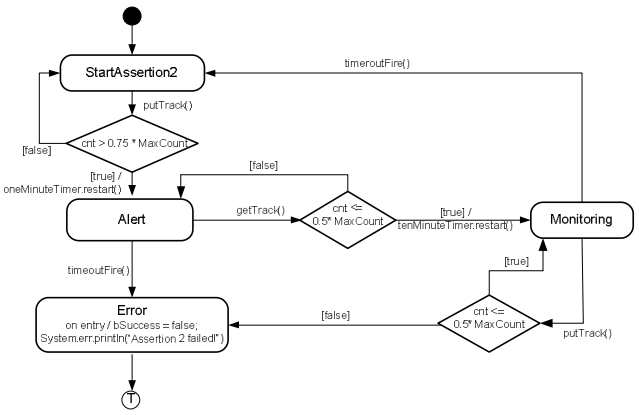

Figure 3. Assertion2 Statechart

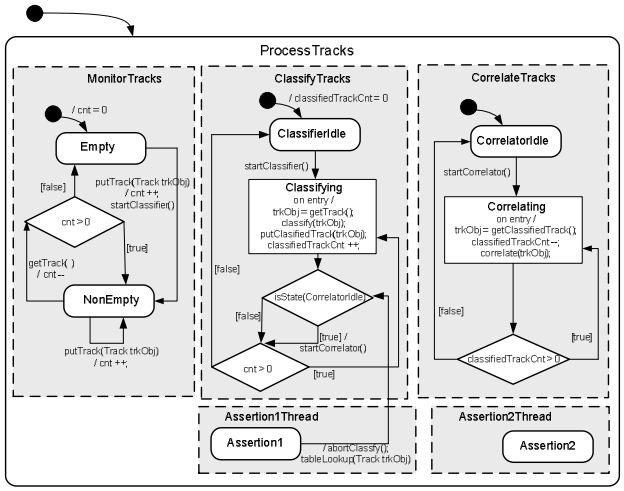

Figure 4. Track processing module statechart with embedded statechart assertions

Figure 4 shows the combined ProcessTracks statechart with embedded statechart assertions, where the Assertion statecharts shown in Figures 2 and 3 now becomes sub-statecharts of the ProcessTracks statechart. In addition, an unlabeled transition from the *Assertion1* substatechart to the decision box "isState(CorrelatorIdel)" in the *ClassifyTracks* thread is added to enable run-time recovery. Whenever the assertion 1 fails because the *classify()* method exceeds its execution time limit, the *Assertion1* substatechart reaches the terminal state (T) and will therefore cause the unlabeled transition out of *Assertion1* to fire, forcing the *ClassifyTracks* thread to abort the on-going *classify()* operation and use a simple table-lookup to get a rough classification of the *trkObj*.

2.3 Target code generation

The StateRover's code generator generates one Java controller class for each statechart file. In our example, we have three statechart diagram files, with the ProcessTracks statechart in the first file and the *Assertion1* and *Assertion2* substatecharts in the second and third files. The StateRover's code generator automatically connects the three statecharts objects resulting in an executable track processing module. 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 ProcessTracksIF, to allow the test drivers or other systems from the external environment to interact with the track processing module.

2.4 Testing of generated code

The generated code is designed to work with the JUnit Test Framework. 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 ProcessTracks receives one track per second for 100 seconds.

```
import junit.framework.*;

public class TestProcessTracks1 extends TestCase {
  private ProcessTracks proc = null;

  public TestProcessTracks1(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    proc = new ProcessTracks ();
  }


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

  // Test Scenario:
  public void testExecTReventDiapatcher() {
    proc.setTime(0); //set clock to 0 sec
    for (i=1;i<=100;i++) {
      //generates a random track
      Track obj = genTrack();

      //send track to ProcessTracks
      proc.putTrack(obj);

      //advance clock by 1 sec
      proc.incrTime(1);
    }
    this.assertTrue(proc.isSuccess());
  }

  Track genTrack(){
    // code for generating a random track
    …
  }
}
```

The StateRover provides an automatic white-box test generator. It generates a JUnit TestCase class. Unlike hand written JUnit TestCase classes, this TestCase (denoted WBTestCase) does not capture a single scenario. Rather, it creates a plurality of tests for a Statechart Under Test (SUT). The WBTestCase create tests that consist of

sequences of events, timing information, and external objects for the SUT.

Automatically generated tests are used in three primary ways: (1) to search for severe programming errors of the kind that induce a JUnit errors, such as NullPointerException, (2) to identify tests that violate embedded statechart assertions -- such failed assertions should be captured by a JUnit assertFalse() statement, using the isSuccess() feedback loop depicted in Figures 2 and 3, and (3) to identify input sequences that lead the SUT-statechart to particular states of interest.

3 Micro-kernel Architecture For Real-time Reactive SoS

In [1], Caffall observed that in many real-time, reactive SoSs, the basic functions of the control software do not change from system to system and from year to year. Almost exclusively changes are to the component systems that provide information to the control software and the component systems that carry out the tasks assigned by the control software. For example, the five basic functions of the battle manager of a global ballistic missile defense systems (detect, track, assign weapon, engage, assess kill) are the same for any ballistic missile defense systems, while the sensors used to collect information for the warfighters, the weapons used to engage threat targets, and the rules of engagement (ROEs) established in both the planning and the C2 functions are constantly changing. Specific features within the battle-management software will also change over time (e.g., discrimination algorithms, correlation algorithms, feature-aided tracking). Furthermore, Caffall proposed to use a microkernel architecture to isolate those features in software components that can be interchanged when developers are prepared to introduce new components (e.g., new types of weapon systems) into the battle management software.

A battle management kernel is similar in purpose to an operating system (OS) kernel in that both kernels manage resources shared by competing entities. The components in the kernel are expected to be stable compared to the other components in the system-of-systems. For instance, device drivers tend to be updated frequently and therefore in principle should not be included in the OS kernel. Figure 5 shows a conceptual view of the micro-kernel proposed by Caffall. The battle management kernel consists of a set of processes (the control software for the five basic functions), a set of interfaces for the battle manager to interact with other component systems, and a set of software components to do predefined work for the kernel software. The software components are software units of composition with contractually specified interfaces and explicit context dependencies. They contain the algorithms required to perform the computations of the BMDS Battle Manager. For example, the track processing will call upon the discrimination component to discriminate various benign objects from the threat ballistic missile.

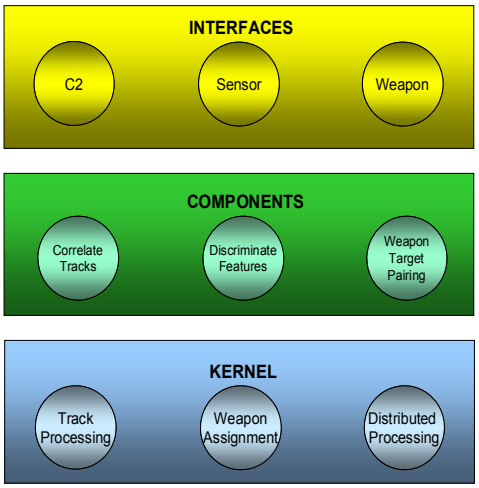

Figure 5. Micro-kernel architecture of the battle manager software (From [1])

To increase the predictability of the dynamic behavior of the battle management software, Caffall proposed to employ the concepts of design-by-contract in the specification and design of the component interfaces, and to use temporal logic and run-time model checking to assure the correct behavior of the control software.

3.1 Qualify assurance of the software components

We propose to use an approach similar to the one proposed by Cheon [3] to create test oracles from the formal behavior specifications of the component interfaces and to employ the runtime assertion checker as the test oracle engine. The only difference is that almost all the previous work on design-by-contract focused on the logical behavior of the software, while we are extending the concept to formally specify and test timing behaviors. To do that, we need to introduce temporal logic and statechart assertions to the pre- and post-conditions of the component interface and to the in-line assertions of the component code. The temporal assertions are translated into target code snippets by tools like the TemporalRover [4] and StateRover [6] code generators and embedded in the component code, so that the assertion evaluator can listen for messages from the component software and evaluate corresponding temporal assertions during test executions.

3.2 Quality assurance of the control software

We propose to express the design of the control software as StateRover statecharts and express the formal specifications of the functional and temporal properties of the control software as embedded statechart assertions. The statecharts are converted by the StateRover's code generator to target control software code, together with the

code that communicate with the StateRover run-time monitor for timing property verification.

We use the StateRover's automatic white box tester to construct a JUnit TestCase class from the statechart model and associated embedded assertions. The white-box test generator is intelligent; it generates only test scenarios that actually affect the statechart SUT or one of its embedded assertions. It is both model-based and specification-based because it uses information from the SUT as well as embedded assertions for test generation.

3.3 Runtime Monitoring of the battle management kernel

To further enhance the robustness of the battle manager software, we can selectively turn on the mission-critical and safety-critical assertions and continuously monitor them using the TemporalRover and StateRover run-time monitor during the execution of the control software. Any violation of the assertions will be caught and run-time recovery code will be executed per specification of the statechart assertions.

4 Discussion and Conclusion

The paper concerns the quality assurance of the timing properties of complex, real-time, reactive system-of-systems. It brings together several technologies to improve the predictability of the system-of-systems' logical and timing behavior.

We increase the maintainability of a SoS with a microkernel architecture that isolates those computations that are likely to change with time from the basic control logics that are invariant in the application domain, and use formal behavior interface specifications to ensure that the software components will perform the intended computation correctly.

We introduce the statecharts with embedded assertions formalism to allow system designers to embed the timing properties into the design model itself. System designers can now reason at a higher level of abstraction (at the state machine level instead of the programming language level), thus improving the understandability of the SoS design.

The target code generated by StateRover is designed to work with the JUnit Test Framework. Use Case scenarios used by the system designers to identify user needs and system requirements can be hand-coded as JUnit test cases and exercised against the generated statechart code. Formal specifications are best understood when presented with multiple views. In our approach, the statechart assertion provides a graphical description of the formal behavior, while the generated code and the scenario-based test cases together provide an executable view of the expected behavior.

The fact that individual software components all satisfy their contracts is not enough to guarantee that the integrated SoS kernel software will satisfy their timing constraints. The availability of StateRover's automatic white box tester is essential in checking the correctness of the timing behavior of the integrated software.

The JUnit test case executes a large volume of test runs of the statechart System Under Test (SUT). A typical white box test case consists of hundreds of thousands of runs of the SUT. The availability of the executable statechart assertions via run-time execution monitoring makes the automatic checking of test results possible and cost-effective.

Finally, the ability to use the statechart assertions to trap exceptions during system execution and invoke run-time recovery is another big step towards making the SoS more dependable. Using the methodology presented in this paper, a mechanism for assuring software quality assurance is built into the component interfaces. If for any reason a component fails to meet its timing requirements assertions, the exception handlers in the control code fire and put the system back into an appropriate state. In essence this methodology provides a continuous Software Quality Assurance (SQA) audit of the system-of-systems.

Acknowledgement

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] D.S. Caffall, *Developing Dependable Software For A System-Of-Systems*, Doctoral Dissertation, Naval Postgraduate School, Monterey, CA, March 2005.

[2] D. Caffall, T. Cook, D. Drusinsky, B. Michael, M. Shing and N. Sklavounos, *Formal Specification and Run-time Monitoring within the Ballistic Missile Defense Project*, Tech. Report NPS-CS-05-007, Naval Postgraduate School, Monterey, California, June 2005.

[3] Y. Cheon, *A Runtime Assertion Checker for the Java Modeling Language*, Tech Report #03-09, Department of

Computer Science, Iowa State University, Ames, Iowa, April 2003.

[4] D. Drusinsky, "The Temporal Rover and ATG Rover", *Lecture Notes in Computer Science. 1885* (Proc. Spin2000 Workshop), pp. 323-329, Berlin: Springer-Verlag, 2000.

[5] D. Drusinsky, "Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions", *Proc. 4th Runtime Verification Workshop (RV'04)*, 2004, Invited paper.

[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, ISBN 0-7506-7949-2.

[7] D. Drusinsky and M. Shing, "Verification of Timing Properties in Rapid System Prototyping", *Proc.14th IEEE International Workshop in Rapid Systems Prototyping*, pp. 47-53, 9-11 June 2003.

[8] D. Drusinsky and G. Watney, "Applying run-time monitoring to the Deep-Impact Fault Protection Engine", *Proc. 28th NASA Goddard Software Engineering Workshop*, IEEE, pp. 127-133, Dec. 2003.

[9] 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.

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