FEATURE: **REQUIREMENTS ANALYSIS**

Verification and Validation for Trustworthy Software Systems

James Bret Michael, Doron Drusinsky, Thomas W. Otani, and Man-Tak Shing, Naval Postgraduate School

// Upfront validation is essential to provide the high level of assurance required of today's trustworthy systems, from consumer products to those involved with the critical infrastructure. //

THE CLOSE INTERACTION between high-integrity systems and their operating environments places a high priority on understanding and satisfying both functional requirements (what the software must do) and safety requirements (what the system must not do). However, traditional validation methods that test the delivered system's behavior against customer expectations are ineffective (and too late) to assure requirement correctness.

Validating requirements early in the system life cycle is increasingly important to organizations that implement capability-based acquisition. For instance, government organizations such as the US Department of Defense (DoD) now play the role of smart buyers whose job is to acquire a set of capabilities.[1] This makes the task of assuring that the system developers correctly translate capabilities into system specifications even more vital. Without such assurance, the DoD can't reasonably expect successful development of trustworthy software-intensive systems.

The US Food and Drug Administration (FDA), on the other hand, plays the role of regulator with the responsibility of approving public use of, say, safety-critical medical devices and investigating the cause of mishaps involving these devices. The FDA must ensure that the device behaves as the manufacturer specifies and that the manufacturer acts with due diligence in assessing its products' trustworthiness—without source code or other detailed information about the systems' implementation.

These examples highlight the need for the continuous and proactive verification and validation (V&V) of complex and safety-critical software systems. This article presents a continuous, computer-aided process that uses statechart assertions, runtime execution monitoring, and scenario-based testing to specify and validate complex system requirements.

Verification & Validation

Merriam-Webster's Dictionary defines *trustworthy* as "deserving of trust or confidence; dependable; reliable."[2] In a system context, *dependability* is "the trustworthiness of a computer system such that reliance can justifiably be placed on the service;" the dependability of a system can be defined by a set of attributes that include availability, reliability, safety, security (confidentiality and integrity), and maintainability.[3]

The *Guide to the Software Engineering Body of Knowledge* defines V&V as the process for determining "whether or not products of a given development or maintenance activity conform to the requirement of that activity, and whether the final software product fulfills its intended purpose and meets user requirements."[4] *Verification* refers to activities

0740-7459/11/$26.00 © 2011 IEEE

that ensure the product is built correctly by assessing whether it meets its specifications. *Validation* refers to activities that ensure the right product is built by determining whether it meets customer expectations and fulfills specific user-defined intended purposes.

Software engineers have become competent at verification: we can build portions of systems to their applicable specifications with relative success. However, we still build systems that don't meet customers' expectations and requirements. This is because people mistakenly combine V&V into one element, treating validation as the user's operational evaluation of the system, resulting in the discovery of requirement errors late in the development process, when it's costly, if not impossible, to fix those errors and produce the right product.

Figure 1 illustrates how the validation process should be proactive and continuous—enacted prior to development and verification, with closure at the end of each phase. Validation is required whenever a requirements-derivation process (that is, a translation of requirements from one domain to another) occurs.

Typically, the requirements-discovery process begins with constructing scenarios involving the system and its environment. From these scenarios, analysts informally express their understanding of the system's expected behavior or properties using natural language and then translate them into a specification.

Specifications based on natural language statements can be ambiguous. For example, consider the following requirement for a project management system: the system shall generate a project status report once every month. Will the system meet the customer's expectation if it generates one report each calendar month? Does it matter if the system generates one report in the last week of May and another in the first week of June? What happens if a project lasts only 15 days? Does the system have to generate at least one report for such a project? Because only the customer who supplied the requirements can answer these questions, the analyst must validate his or her own cognitive understanding of the requirements with the customer to ensure that the specification is correct.

Research has shown that formal specifications and methods help improve the clarity and precision of requirements specifications (for example, see the work of Steve Easterbrook and his colleagues).[5] However, formal specifications are useful only if they match the true intent of the customer's requirements. Let's assume that the customer expects the system to generate a report at the end of each calendar month, even if the project ends earlier. A system that satisfies the formal specification **always (project_active $\wedge$ last_day_of_month => report_generation)** might still fail to meet the customer's requirement regarding reports for projects that end before the last day of a month.

Possible reasons for an incorrect assertion include

- incorrect translation of the natural language specification to a formal specification,
- incorrect translation of the requirement, as understood by the analyst, to natural language, and
- incorrect cognitive understanding of the requirement.

These situations typically occur when the requirement was driven from the use case's main success scenario, with insufficient investigation into other scenarios. Consequently, we propose the iterative process for assertion validation shown in Figure 2. This process encodes requirements as Unified Modeling Language (UML) statecharts augmented with Java action statements and validates the assertions by executing a series of scenarios against the statechart-generated executable code to determine whether the specification captures the intended behavior. This approach helps provide evidence for building assurance cases as proposed by Robin E. Bloomfield and his colleagues.[7]

Assertion-Oriented Specification

Instead of using a single monolithic formal model (either as a state- or an algebraic-based system) to capture the combined expected behavior of a system meeting all requirements, our assertion-oriented approach maps user requirements one-to-one to formal assertions, allowing explicit traceability between formal assertions and user

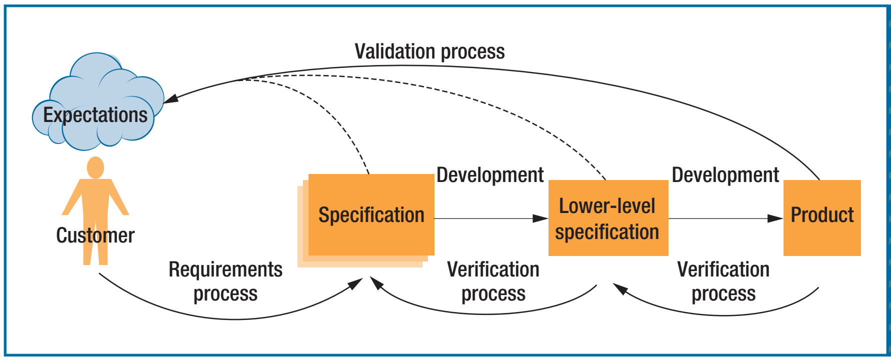


FIGURE 1. A continuous validation and verification process. Validation ensures the requirements correctly capture the users' and stakeholders' expectations and should be performed whenever a translation of requirements from one domain to another occurs.

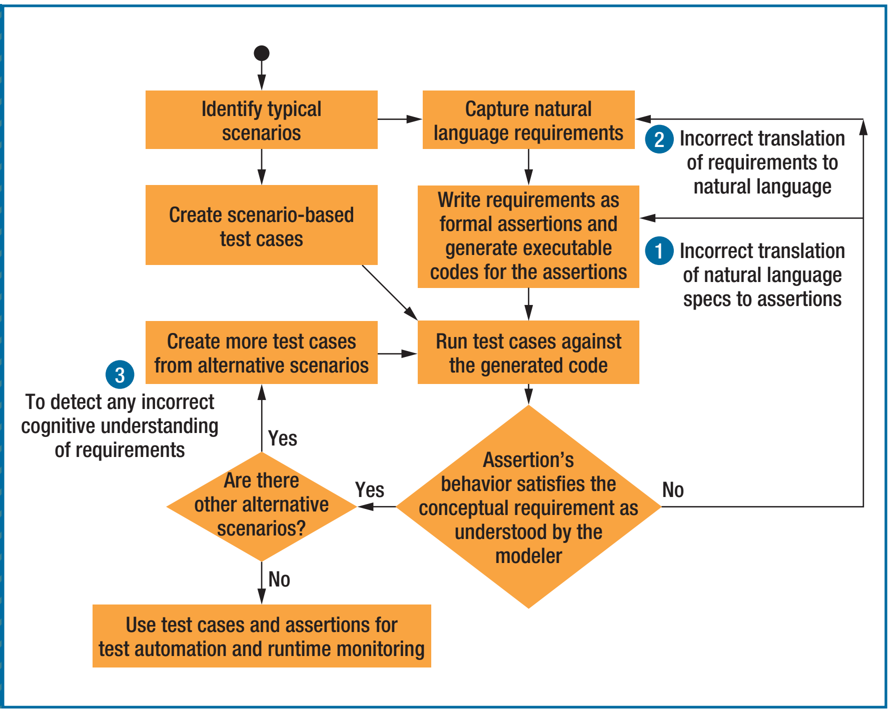

Identify typical scenarios
Capture natural language requirements
Incorrect translation of requirements to natural language
Create scenario-based test cases
Write requirements as formal assertions and generate executable codes for the assertions
Incorrect translation of natural language specs to assertions
Create more test cases from alternative scenarios
Run test cases against the generated code
To detect any incorrect cognitive understanding of requirements
Yes
Are there other alternative scenarios?
Yes
Assertion's behavior satisfies the conceptual requirement as understood by the modeler
No
No
Use test cases and assertions for test automation and runtime monitoring

FIGURE 2. Iterative process for assertion validation.[6] By encoding the system requirements as statechart assertions and testing the generated code against different use case scenarios, analysts can validate the formal specifications and make necessary modifications early in the development process.

requirements and ease of requirement maintenance.

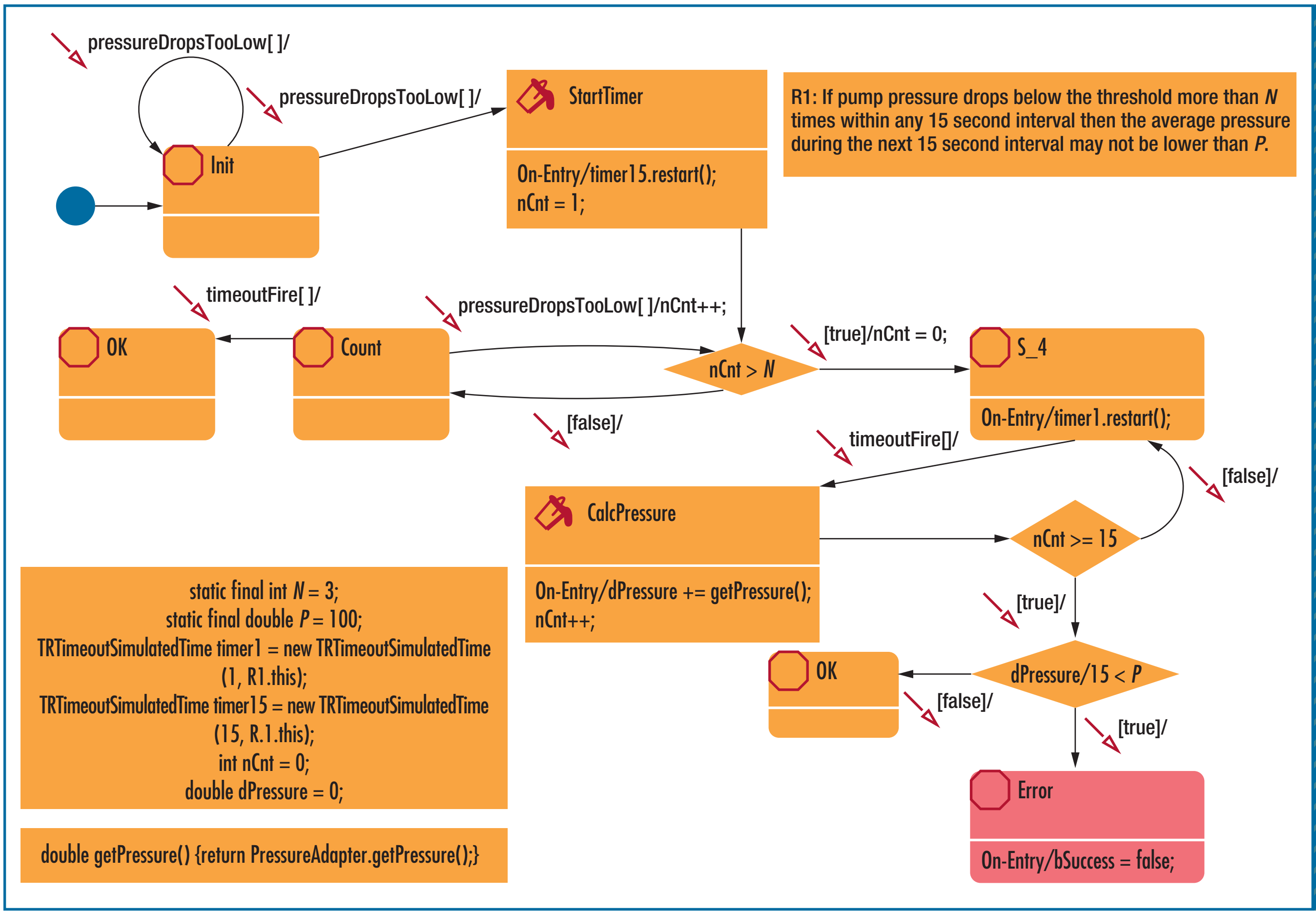
Many methods and languages exist for formally specifying and analyzing system behaviors, but they differ in their ability to describe complex system behavior, their ease of use to produce correct specifications, and their effectiveness in verifying the target code. Our previous work presented a visual space for engineers to discuss the various trade-offs between different formal verification and validation (FV&V) techniques.[8] Today's safety-critical systems often involve complex sequencing behaviors with time-series constraints. Consider the infusion pump controller requirement *R1*: if pump pressure drops below the threshold more than *N* times within any 15-second interval, then the average pressure during the next 15-second interval must not be lower than *P*. Formally specifying *R1* requires the ability to express counting constraints, sliding time window constraints, and time-series data constraints (for example, the average of pressure readings over a 15-second interval), which are lacking in most existing formal V&V techniques. Figure 3 shows a statechart assertion for *R1*, where the second 15-second interval starts immediately at the detection of the fourth **pressureDropsTooLow** event within a 15-second interval, according to the analyst's interpretation of the natural language requirement.

A statechart assertion is a UML statechart-based formal specification for use in prototyping, runtime monitoring, and execution-based model checking.[9] It extends the Harel statechart formalism[10] and is supported by the StateRover plug-in (www.time-rover.com/staterover.pdf).

Statechart assertions are easier to create and understand than text-based temporal assertions: they're visual, resemble statechart design models, and provide an intuitive way to express various constraints. In addition, statechart assertions are executable to let developers and customers visualize the true meaning of the assertions via scenario-based simulations. The assertion-oriented approach produces a collection of statecharts that are far less complex than those in design models composed of UML state diagrams, thus avoiding the automated code generation problems in typical software development.

Unlike Harel statecharts, statechart assertions are written from an external observer's viewpoint. They extend Harel statecharts in two ways: they include a built-in Boolean flag **bSuccess** (and a corresponding **isSuccess** method) that specifies the assertion's Boolean status (true if the assertion succeeds and false otherwise), and they can be nondeterministic. The addition of the **bSuccess** flag makes the statechart assertion suitable as a formal specification, similar to a formal logic such as linear-time temporal logic.[11] By default, the statechart assertion in Figure 3 begins with the result flag **bSuccess** being true. It behaves like a Harel statechart. When the statechart assertion enters the **Error** state (because the average pressure over the last 15-second interval is less than *P*), the on-entry action assigns **bSuccess** to false, signaling assertion violation.

Although deterministic statechart assertions suffice for specifying many requirements, theoretical results show that nondeterministic statecharts are exponentially more succinct than deterministic Harel statecharts.[12] As Figure 3 illustrates, an apparent next-state conflict exists when the event **pressureDropsTooLow** is sensed while the statechart assertion is in the **Init** state. StateRover uses a special code generator to create a plurality of state-configuration objects for nondeterministic statechart assertions, one per



pressureDropsTooLow[ ]/
pressureDropsTooLow[ ]/
StartTimer
R1: If pump pressure drops below the threshold more than N times within any 15 second interval then the average pressure during the next 15 second interval may not be lower than P.
Init
On-Entry/timer15.restart();
nCnt = 1;
timeoutFire[ ]/
pressureDropsTooLow[ ]/nCnt++;
[true]/nCnt = 0;
OK
Count
nCnt > N
S_4
[false]/
timeoutFire[]/
On-Entry/timer1.restart();
[false]/
CalcPressure
nCnt >= 15
On-Entry/dPressure += getPressure();
nCnt++;
[true]/
OK
dPressure/15 < P
[false]/
[true]/
Error
On-Entry/bSuccess = false;

```
static final int N = 3;
static final double P = 100;
TRTimeoutSimulatedTime timer1 = new TRTimeoutSimulatedTime
(1, R1.this);
TRTimeoutSimulatedTime timer15 = new TRTimeoutSimulatedTime
(15, R.1.this);
int nCnt = 0;
double dPressure = 0;
```

```
double getPressure() {return PressureAdapter.getPressure();}
```

FIGURE 3. A statechart assertion for requirement R1. A statechart assertion describing formally the requirement that if pump pressure drops below the threshold more than *N* times within any 15-second interval, then the average pressure during the next 15-second interval must not be lower than *P*.

possible computation in the assertion statechart. Nondeterministic statechart assertions use an existential definition of the isSuccess method, where if at least one state configuration detects an error (assigns bSuccess=false) then the isSuccess method for the entire nondeterministic assertion returns false. The StateRover also has a power-user priority mechanism to change or limit the existential default definitions of the isSuccess method and the terminal state.

Figure 4 shows the timing diagrams of two different test scenarios applied to the statechart assertion of Figure 3. Scenario 1 corresponds to the typical case in which the infusion pump successfully maintains the pressure to no more than three pressureDropsTooLow events within any 15-second interval. Scenario 2 corresponds to a potential failure scenario in which timestep 21 observes four pressureDropsTooLow events within an interval of 11 seconds. The statechart assertion enters the S_4 state at timestep 21 and starts summing up the pressure reading once every second for the next 15 seconds. It computes the average pressure (= 1,585/15 > 100) at timestep 36, checking that the average exceeds the threshold of 100, and enters the OK state. Because the bSuccess flag remains true from timestep 36 onward, the isSuccess method returns true at timestep 40. Now, suppose the customer meant to have the second 15-second interval start after the completion of the first 15 seconds—that is, at timestep 25 instead of timestep 21. Then, the customer would expect the statechart assertion to fail for the second scenario because the average pressure from timesteps 25 through 40 (=1,497/15) is less than 100, thus revealing that the customer and analyst have different cognitive understandings of the requirement.

The formal assertions (particularly those for time-series constraints) must be executable to allow analysts to visualize the actual meaning of the assertions via scenario-based simulations.

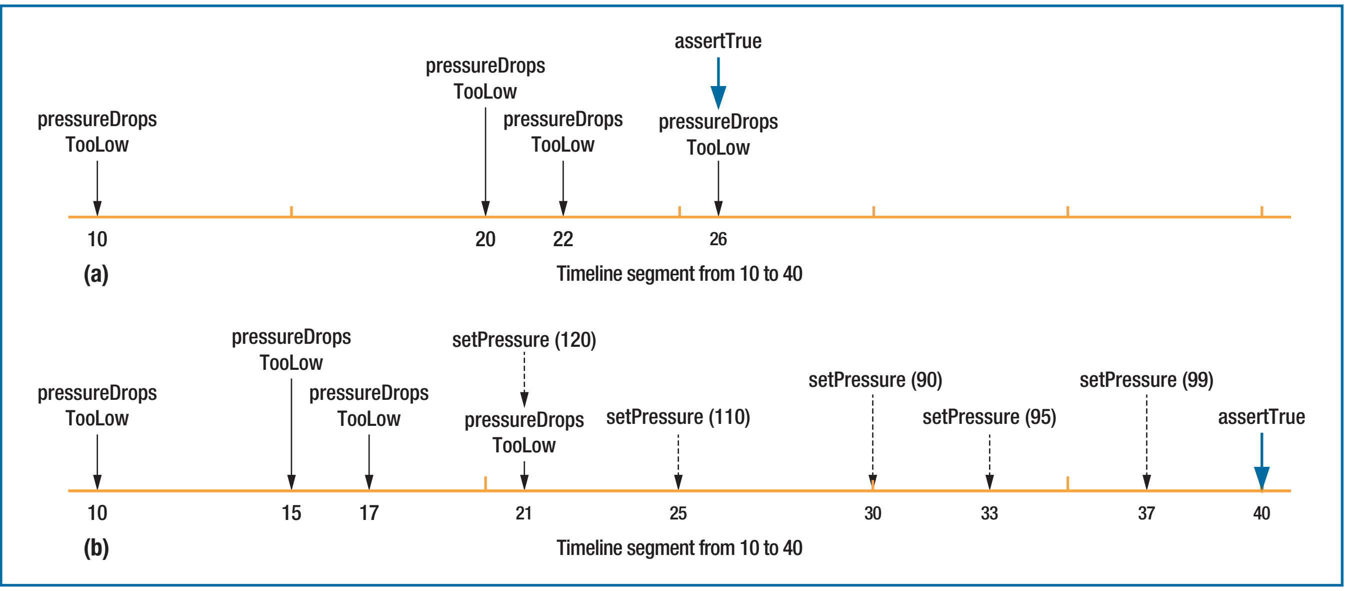


FIGURE 4. Timing diagram of the two test scenarios for R1. Multiple test scenarios are used to validate the statechart assertion to make sure that the assertion matches the user's expectation in its acceptance and rejection of each scenario. (a) Scenario 1: no more than three **pressureDropsTooLow** events within a 15-second interval. (b) Scenario 2: four **pressureDropsTooLow** events within a 15-second interval and the average pressure 15 seconds afterwards is above *P*.

To ensure that the statechart assertion correctly represents the intended behavior, we use StateRover to generate executable Java code from the statechart assertion and then run validation test scenarios against it within the JUnit test framework (see Figure 5).[13]

A test suite that includes both success and failure scenarios should accompany each statechart assertion. Without such a test suite it's difficult to interpret the original developer's intent. A good set of test scenarios also helps stakeholders 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.

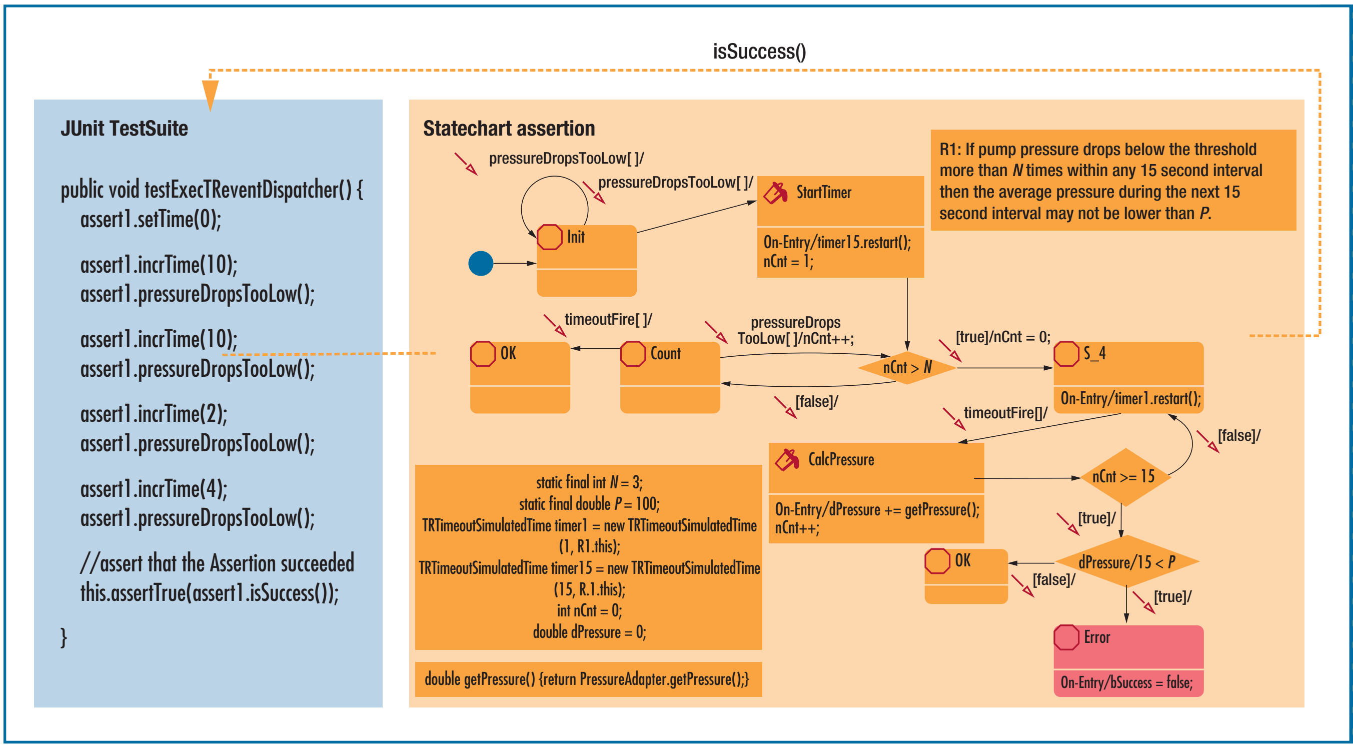
Our previous work provides a list of commonly used assertion patterns and their accompanying validation test scenarios.[14] We advocate the use of libraries of assertions to reduce the requirement analysts' workload. Rather than having analysts always specify assertions from scratch, they can reuse the assertions or patterns from such a library. Clearly, effective reuse will help reduce the number of errors analysts introduce when specifying assertions. The assertion libraries should also include the test scenarios originally used to validate each assertion.

Traditional testing techniques rely on manually written test inputs and manual checking of the observed behavior against natural language specifications. Such an approach isn't useful in verifying today's trustworthy systems because it's too labor intensive and error-prone for a human to check the correctness of the complex sequencing behaviors based on the observed output and his or her understanding of the natural language specification. Instead, we recommend using runtime execution monitoring, which uses a set of validated executable assertions to verify the correctness of the system under test.

SRM Process for IV&V

We expect a trustworthy system to perform correctly according to its stated requirements. Developers run a suite of tests and other forms of verification to ensure their systems meet customer expectations. Despite this effort however, systems might fail with serious consequences more frequently than customers can tolerate. To increase stakeholder confidence in a system's correctness, an independent verification and validation (IV&V) process should exist. With this process, a team of engineers who aren't members of the development team conduct V&V. We consider IV&V essential for detecting subtle, yet critical, errors that developers often overlook.

IV&V has been a mandatory practice in the aeronautics, space, railway, and defense industries, where reliability, security, and safety are of paramount importance, but so far, it hasn't attained its full potential, in part due to a lack of appropriate tools and methodologies. In the past, the IV&V team relied primarily on the manual examination of software requirements and design artifacts along with analysis of the source code, either manually


isSuccess()
JUnit TestSuite
Statechart assertion
pressureDropsTooLow[ ]/
pressureDropsTooLow[ ]/
StartTimer
Init
On-Entry/timer15.restart();
nCnt = 1;
R1: If pump pressure drops below the threshold more than N times within any 15 second interval then the average pressure during the next 15 second interval may not be lower than P.
timeoutFire[ ]/
pressureDrops TooLow[ ]/nCnt++;
[true]/nCnt = 0;
OK
Count
nCnt > N
S_4
[false]/
On-Entry/timer1.restart();
timeoutFire[]/
[false]/
CalcPressure
nCnt >= 15
On-Entry/dPressure += getPressure();
nCnt++;
[true]/
OK
dPressure/15 < P
[false]/
[true]/
Error
On-Entry/bSuccess = false;

```
public void testExecTReventDispatcher() {
    assert1.setTime(0);

    assert1.incrTime(10);
    assert1.pressureDropsTooLow();

    assert1.incrTime(10);
    assert1.pressureDropsTooLow();

    assert1.incrTime(2);
    assert1.pressureDropsTooLow();

    assert1.incrTime(4);
    assert1.pressureDropsTooLow();

    //assert that the Assertion succeeded
    this.assertTrue(assert1.isSuccess());

}
```

```
static final int N = 3;
static final double P = 100;
TRTimeoutSimulatedTime timer1 = new TRTimeoutSimulatedTime
(1, R1.this);
TRTimeoutSimulatedTime timer15 = new TRTimeoutSimulatedTime
(15, R.1.this);
int nCnt = 0;
double dPressure = 0;
```

```
double getPressure() {return PressureAdapter.getPressure();}
```

FIGURE 5. Validating statechart assertion via scenario-based testing. The scenarios are encoded as JUnit test cases and run against the code generated automatically from the statechart assertions within the JUnit test framework.

or by employing a software analyzer. As an added measure, the team sometimes performs systematic or random independent testing of target code. Although this type of IV&V has merit, it falls short of what an ideal IV&V can accomplish.

Our previous work introduced the System Reference Model (SRM) framework for IV&V.[15] The SRM is composed of high-level, goal-oriented use cases, UML artifacts, and formal assertions. Collectively, the SRM's components specify the desired behavior of the system under test. Through these components, the IV&V team can answer the following questions:

- What should the system do?
- What should the system not do?
- How should the system respond to abnormal conditions?

For the SRM framework to effectively establish a clear and correct understanding of the problem space, it must be executable. Instead of just marveling at beautiful "wallpaper" UML diagrams, we should be able to execute the model through scenario-based simulations to confirm the correctness of the independent team's understanding of the problem space. Statechart assertions are integral to making the SRM executable and effective.

The starting point of both understanding and documenting system behaviors is to identify the high-level use cases (and use case scenarios) from the stakeholder's input, which could come in the form of mission statements, user expectations, and operation concepts (and other concept-level documents). For the purpose of IV&V of software systems, the high-level use cases must be reified into mission threads (that is, detailed use cases) that capture the interactions among component systems (or subsystems). Concurrent to the development of use cases (and their scenarios) and activity and sequence diagrams, analysts must also develop a conceptual model (an object class diagram) to capture the essential concepts and manage the problem's namespace. Safety-critical and mission-essential behaviors are then expressed as statechart assertions, which are validated via scenario-based testing for correctness.

When validation is complete, we have a set of validated formal assertions that capture the requirements unambiguously and precisely without any conflicts. An example of such a process, applied to the NASA Grail mission, is available in a previous work.[16] After validating the requirements, we move into the verification phase, where we test run the software using the executable statechart assertions as monitors and verify that the software doesn't violate the validated formal specifications.

ABOUT THE AUTHORS

JAMES BRET MICHAEL is a professor in the Departments of Computer Science and Electrical & Computer Engineering at the Naval Postgraduate School. His research interests include mobile computing, cloud computing, and computer-aided formal verification and validation of distributed systems. Michael has a PhD in information technology from George Mason University. Contact him at bmichael@nps.edu.

DORON DRUSINSKY is an associate professor in the Department of Computer Science at the Naval Postgraduate School. His research interests include computer-aided specification, validation, and verification of mission-critical systems. Drusinsky has a PhD in computer science from the Weizmann Institute of Science. Contact him at ddrusins@nps.edu.

THOMAS W. OTANI is an associate professor in the Department of Computer Science at the Naval Postgraduate School. His research interests include object-oriented modeling and design, cloud computing, and user interface design. Otani has a PhD in computer science from the University of California. Contact him at twotani@nps.edu.

MAN-TAK SHING is an associate professor in the Department of Computer Science at the Naval Postgraduate School. His research interests include service-oriented and cloud computing, modeling and design of real-time and distributed systems, and the specification, validation, and use of temporal assertions. Shing has a PhD in computer science from the University of California, San Diego. Contact him at shing@nps.edu.

Our approach is a powerful addition to the V&V practices in use today. We applied it to the distributed V&V of a set of critical time-constrained requirements for the Brazilian Satellite Launcher flight software.[17] The statechart assertions were created in California, as was log-file based runtime verification by running the event trace extracted from the log files against the statechart asserions. Instrumented target code execution and log file creation were performed in Brazil. The results of the case study indicate that the process and its computer-aided environment were both technically and managerially effective, and can scale well to cater for V&V of larger mission-critical systems. The case study also demonstrated the feasibility of conducting distributed verification.

References

1. P. Charles and P. Turner, "Capabilities Based Acquisition: From Theory to Reality," *CHIPS*, vol. 22, no 3, 2004, pp. 38–39.
2. *Merriam-Webster's New Universal Unabridged Dictionary*, 2nd ed., Simon & Schuster, 1983.
3. J.C. Laprie, A. Avizienis, and H. Kopetz, eds., *Dependability: Basic Concepts and Terminology*, Springer, 1992.
4. P. Bourque and R. Dupuis, eds., *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE CS Press, 2004; http://ieeexplore.ieee.org/servlet/opac?punumber=4425811.
5. S. Easterbrook et al., "Experiences Using Lightweight Formal Methods for Requirements Modeling," *IEEE Trans. Software Eng.*, vol. 24, no. 1, 1998, pp. 4–11; doi:10.1109/32.663994.
6. D. Drusinsky, M. Shing, and K. Demir, "Creating and Validating Embedded Assertion Statecharts," *IEEE Distributed Systems Online*, vol. 8, no. 5, 2007, p. 3; doi:10.1109/MDSO.2007.25.
7. R. Bloomfield et al., "International Working Group on Assurance Cases (for Security)," *IEEE Security & Privacy*, vol. 4, no. 3, 2006, pp. 66–68; doi:10.1109/MSP.2006.73.
8. D. Drusinsky, J.B. Michael, and M. Shing, "A Visual Tradeoff Space for Formal Verification and Validation Techniques," *IEEE Systems Journal*, vol. 2, no. 4, 2008, pp. 513–519; doi:10.1109/JSYST.2008.2009190.
9. D. Drusinsky, *Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-Based Model Checking*, Elsevier, 2006.
10. D. Harel, "Statecharts: A Visual Approach to Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, 1987, pp. 231–274.
11. A. Pnueli, "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. Foundations of Computer Science*, IEEE CS Press, 1977, pp. 46–57.
12. D. Drusinsky and D. Harel, "On the Power of Bounded Concurrency I: Finite Automata," *J. ACM*, vol. 41, no. 3, 1994, pp. 517–539; doi:10.1145/176584.176587.
13. K. Beck and E. Gamma, "Test Infected: Programmers Love Writing Tests," *Java Report*, vol. 3, no. 7, 1998, pp. 37–50.
14. D. Drusinsky et al., "Validating UML Statechart-based Assertions Libraries for Improved Reliability and Assurance," *Proc. 2nd Int'l Conf. Secure System Integration and Reliability Improvement*, IEEE CS Press, 2008, pp. 47–51; doi:10.1109/SSIRI.2008.54.
15. D. Drusinsky, J.B. Michael, and M. Shing, "A Framework for Computer-Aided Validation," *Innovations in Systems and Software Eng.*, vol. 4, no. 2, 2008, pp. 161–168; doi:10.1007/s11334-008-0047-2.
16. D. Drusinsky and S. Raque, *Specification and Validation of Space System Behaviors*, tech. report NPS-CS-10-002, Dept. of Computer Science, Naval Postgraduate School, 2010.
17. M. Alves et al., "Formal Validation and Verification of Space Flight Software Using Statechart-Assertions and Runtime Execution Monitoring," *Proc. 6th IEEE Int'l System of Systems Eng. Conf.* (SoSE), IEEE CS Press, pp. 155–160; doi:10.1109/SYSOSE.2011.5966564.