NPS-AM-13-084

ACQUISITION RESEARCH PROGRAM SPONSORED REPORT SERIES

Computer-Aided Process and Tools for Mobile Software Acquisition

30 July 2013

LT Christopher Bonine, USN,

Dr. Man-Tak Shing, Associate Professor, and

Dr. Thomas W. Otani, Associate Professor

Graduate School of Operational & Information Sciences

Naval Postgraduate School

Approved for public release; distribution is unlimited.

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

Abstract

Mobile devices have, in many ways, replaced traditional desktops in usability, usefulness, and availability. Many companies are scrambling to develop enterprise strategies to provide mobile devices and application support for their employees, and the Department of Defense (DoD) is taking the point in the federal government's campaign to deploy mobile devices. A successful DoD mobile software acquisition program requires efficient and effective means to ensure the proper functioning of the applications. As the majority of future mobile apps will be developed by small companies (or crowdsourcing individuals) and have relatively short development cycles, the traditional software verification process that relies on testing of source code is not effective for vetting mobile apps. This paper presents a new approach for vetting mobile software. It allows subject-matter experts to specify desirable and undesirable behaviors of the mobile apps as executable statecharts and verify the target software by running the automatically generated statechart code against the execution trace of the mobile apps using logfile-based runtime verification. A case study of formally specifying, validating, and verifying a set of requirements for an iPhone application that tracks the movement of the iPhone user is used to demonstrate the new approach.

Keywords: Mobile apps, formal specification, statechart assertion, requirements validation, logfile-based runtime verification

THIS PAGE INTENTIONALLY LEFT BLANK

About the Authors

Christopher Bonine is a lieutenant in the United States Navy. He is currently assigned to the Navy Cyber Defense Operations Command in Norfolk, VA. He has served as information warfare officer onboard the USS *Sampson* and as N51 division officer at the Navy Cyber Warfare Development Group. His current interests are in development and implementation of cyber security policy. Bonine has a master's in computer science from the Naval Postgraduate School.

Graduate School of Business & Public Policy
Naval Postgraduate School
Monterey, CA 93943-5000
Fax: (831) 656-3407
E-mail: cbbonine@nps.edu

Dr. Man-Tak Shing is an associate professor at the Naval Postgraduate School. His research interests include the engineering of software intensive systems. He is on the program committees of several software engineering conferences. He was the program co-chair of the Rapid System Prototyping Workshop in 2004 prior to being the general co-chair for the symposium in 2008. He also served as the program co-chair of the IEEE System of Systems Engineering Conference in 2010 and 2011. He received his PhD in computer science from the University of California–San Diego and is a senior member of IEEE.

Graduate School of Business & Public Policy
Naval Postgraduate School
Monterey, CA 93943-5000
Tel: (831) 656-2634
Fax: (831) 656-3407
E-mail: shing@nps.edu

Dr. Thomas W. Otani is an associate professor of computer science at the Naval Postgraduate School. His main research interests include object-oriented programming, mobile and web application development, and database design. He received his PhD in computer science from the University of California–San Diego in 1983.

Graduate School of Business & Public Policy
Naval Postgraduate School
Monterey, CA 93943-5000
Tel: (831) 656-3391
Fax: (831) 656-3407
E-mail: twotani@nps.edu

THIS PAGE INTENTIONALLY LEFT BLANK

NPS-AM-13-084

ACQUISITION RESEARCH PROGRAM SPONSORED REPORT SERIES

Computer-Aided Process and Tools for Mobile Software Acquisition

30 July 2013

LT Christopher Bonine, USN,

Dr. Man-Tak Shing, Associate Professor, and

Dr. Thomas W. Otani, Associate Professor

Graduate School of Business and Public Policy

Naval Postgraduate School

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Computer-Aided Process and Tools for Mobile Software Acquisition[1]

Introduction

In an April 23, 2012, blog post, analyst Frank E. Gillett of Forrester Research predicted that "tablets will become our primary computing device" in the near future, with "global tablet sales to reach 375 million units, with one-third purchased by businesses and two-fifths (or 40 percent) by emerging markets" by 2016. Many companies are scrambling to develop enterprise strategies to provide mobile devices and application support for their employees, and the Department of Defense (DoD) "is taking the point in the federal government's campaign to deploy mobile devices" (Kenyon, 2012a). The Defense Information Systems Agency (DISA) has opened a program office and issued a request for information to solicit ideas from industry for ways to provide the mobile device management (MDM) services and to run an applications store (Kenyon 2012b), and the Army has established the Army Software Marketplace, a prototype online storefront for Army-wide distribution of mobile software.

As the DoD is charging forward with its mobile programs, it must find ways to address its concerns in security, authentication, and the logistics in managing and deploying the rapidly growing number of mobile applications and devices with varying degrees of access across the DoD enterprise. The Space and Naval Warfare (SPAWAR) Atlantic System Center is working with the DISA and the National Institute of Standards and Technology (NIST) to provide warfighters with access to unclassified information from their handheld devices via the cloud-based mobility-as-a-service, and the adoption of a hardened kernel for the Android mobile operating system is another major step towards providing a secure base for the development of trustworthy mobile software. Moreover, the DoD needs an efficient and effective process to ensure the proper functioning of the mobile software (commonly referred to as mobile apps) and to ensure that the software do what it promises to do without hidden or emergent malicious behaviors.

Mobile apps shrink the software programs that were once only available on a desktop computer, making them usable on smart phones and mobile devices. The app market has been growing at an unprecedented rate. The app world, which consisted of 8,000 Apple titles in 2008, has reached 1 million titles in 2011

[1] This work was supported in part by the NPS Acquisition Research Program—OUSD_13 (Project #:F13-010, JON: R8G59). The views and conclusions in this paper 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.

(Freierman, 2011). As the majority of mobile apps is developed by small companies (or crowdsourcing individuals) and has relatively short development cycles, the traditional software verification process that relies on testing of source code is not effective for vetting mobile software. The DoD needs better means to ensure proper functioning of mobile apps without source code or other detailed information about the software's implementation.

This paper presents a new approach for vetting mobile software. It allows subject-matter experts to specify desirable and undesirable behaviors of the mobile apps as executable statecharts and to verify the target software by running the automatically generated statechart code against the execution trace of the mobile apps using logfile-based runtime verification.

The rest of the paper is organized as follows. The section titled The V&V of Mobile Apps provides a summary of the current state of verification and validation (V&V) of mobile apps. The Formal Specification and Validation of Mobile Apps section presents an overview of statechart assertions, the formal specification language of choice, and the proposed computer-aided process for the V&V of mobile apps. The Case Study section presents a case study involving the formal specification, validation, and verification of a set of requirements for an iPhone application that tracks the movement of the iPhone user. The final section provides a summary and draws some conclusions.

The V&V of Mobile Apps

V&V is a software evaluation process that ensures proper and expected operation. As stated by Michael, Drusinsky, Otani, and Shing (2011),

> verification refers to activities 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. (pp. 86–87)

Simply stated, the purpose of V&V is to ensure the software does what it is required to do, and nothing more.

Difficulties in Testing Mobile Apps

New mobile devices, especially phones, have such short development times that the devices have been on the market barely long enough to work out existing bugs before a new device with new software is ready to be released. As an example, Apple releases a new iPhone model every year and has developed six generations of iOS. The Android operating system had eight versions in three years. This high turnover of mobile devices is created not only by demand and competition, but also by capability increases of computing power, battery life, and screen size. As new

capabilities are added to the devices and applications in each development cycle, new automated V&V techniques are needed to keep up with the fast pace of mobile application development.

Additional difficulties in testing mobile applications are due to the limitations of the hardware. At this time, other than operating system tasks, iPhone can run only a single application at a single point in time. The purpose is to conserve the limited computing power of the device as well as reduce power consumption. The negative aspect is that there is little or no application interaction on a single device. This prevents useful testing applications from running on mobile devices to analyze the real-time behavior of application. Even if such ability were possible, the small screen size would create difficulties in analyzing the data while on the device. Android devices have the ability for third party developers to create multiprocessing applications, which could allow analytics to be conducted directly on the device, but the same screen size limitation would impede analysis of the data. (Readers can refer to Muccini, Francesco, and Esposito [2012] for a detailed discussion of the challenges in testing mobile apps.)

These limitations make testing done off the device more amenable. There are two possible options: use device-specific emulators, or use specially altered software code to allow offloading of real data from the device onto a computer for analysis. Although the emulators do a good job of creating a proper environment to test an application, that method has the limitation of being stuck in place and does not recreate the ever-changing environment where mobile devices exist. The other method could potentially include such a robust environment. However, currently existing techniques require tethering the mobile device to an immobile computer with a cable connection, thus significantly reducing the mobility of the device under test.

Current Solutions to V&V of Mobile Apps

Monkeyrunner enables the writing of unit tests to test software at a functional level (“Monkeyrunner Tool,” n.d.). Monkeyrunner uses Python to run testing code on one or more devices, or an emulator. It can send commands and keystrokes, and record screenshots. Monkeyrunner allows for repetition of test results, but element location in the recorded screenshots is the basis for comparing two test results, thus limiting comparisons to a single screen size.

Android Robotium is a Java-based tool for writing unit tests (“User Scenario Testing for Android,” n.d.). Similar to Monkeyrunner, it is designed to run as a black-box testing tool and can run as an emulator as well as run on the actual device, although it is limited to a single device. Robotium allows for testing of pre-install software as well. The big difference between Robotium and Monkeyrunner is that Robotium has a more robust test result comparison. Rather than using a location-

based method, Robotium uses identifiers to recognize elements, which allows devices of different types and sizes to be compared to ensure consistency.

Lesspainful.com provides a way for customers to run software and unit tests on physical devices without the cost of owning the devices ("Lesspainful Device Lab," n.d.). The customers use the programming language Cucumber to write an English description of the test they would like to run on their software. Once the devices to be tested on are chosen, the tests are automated in a cloud-like system with results from each mobile device presented to the customer to allow for easy comparison.

Testquest 10 is a software suite, created by BSquare, which enables unit tests in a device emulator and enables collaboration of geographically dispersed teams ("Testquest Automated Testing," 2003). It utilizes extensive use of image recognition to determine device state as well as the location of applications and features on the screen. An interesting feature is that if the graphical user interface (GUI) design is changed and an application or feature is moved from one location to another, then this suite is able to locate and use the feature.

Bo, Xiang, and Xiaopeng (2007) introduced an approach for testing a device and software by using what they called sensitive-events. Their approach reduces the need for screenshot comparisons by capturing these events, such as inbox full, to determine state change. The software then evaluates these state changes, and if the events indicate the desired conditions, then the tests continue.

All of the aforementioned software tools are for testing an application to ensure proper functionality and operations. What they are missing is the ability to map the operation of the phone directly to a set of requirements. The above tools all require some form of script writing, which can lead to missing software test cases; when writing scripts to cover unit tests, the programmer must understand the requirements and determine boundary (edge) cases in order to properly test for them. The tools are also limited in their ability to handle context-aware features. Another limitation is that, due to the limitation of the hardware and the software testing suites, only one application at a time can be tested.

Delamaro, Vincenzi, and Maldonado (2006) introduced JaBUTi/ME, which extends the Java byte code analysis tool JaBUTi by adding the ability to run instrumented code on a mobile device to create trace data and then pass the trace data to a desktop computer for analysis. By using a method of creating trace data, this solution is conceptually similar to the idea presented in this paper. However, this method still requires test cases to be manually written to evaluate the resulting trace file. Additionally, as the authors stated, the code instrumentation would vary based on the hardware device the code is being tested on, due to the potential differences

in network connectivity needed to transmit the trace data back to the desktop computer.

Formal Specification and Validation of Mobile Apps

Michael et al. (2011) pointed out that

> 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. (p. 87)

Hence, first and for most, we need a means for analysts to describe the desirable and undesirable behaviors of the mobile apps. 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 software: The software shall generate a project status report once every month. Will the software meet the customer's expectation if it generates one report each calendar month? Does it matter if the software generates one report in the last week of May and another in the first week of June? What happens if a project ends before the report generation day? Does the software have to generate a report for such a project?

Research has shown that formal specifications and methods help improve the clarity and precision of requirements specifications (Easterbrook, Lutz, Covington, Ampo, & Hamilton, 1998). However, formal specifications are useful only if they match the true intent of the customer's requirements. Because only the subject-matter expert who supplied the requirements can answer these questions, the analyst must validate his or her own cognitive understanding of the requirements with the subject-matter expert to ensure that the specification is correct. For example, consider the security requirement R1: *If there are more than two invalid login attempts within any 15-second interval, then the mobile device will remain unavailable for 10 minutes.* Whether the scenario shown in Figure 1 violates R1 depends on the interpretation of the starting time of the 10-minute timeout interval.

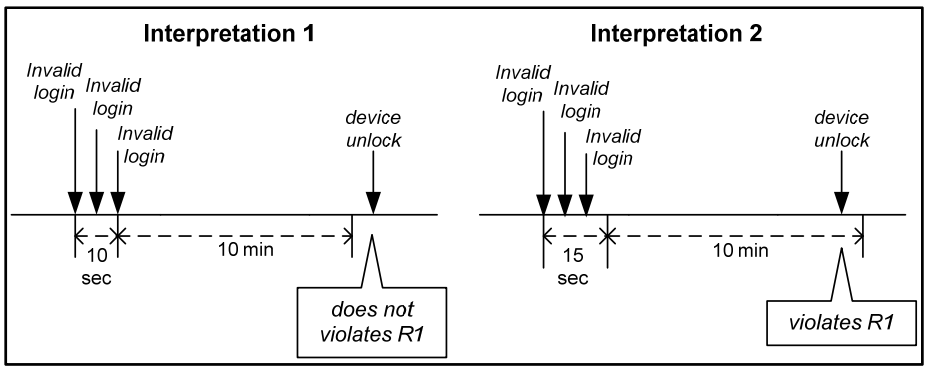

Figure 1. Example of Requirements Ambiguity

The best way to validate and disambiguate complex behavioral requirements is to walk through the different scenarios with the stakeholders and ask them to confirm or clarify the analyst's cognitive understanding of the natural language requirements. Drusinsky, Shing, and Demir (2007) proposed 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.

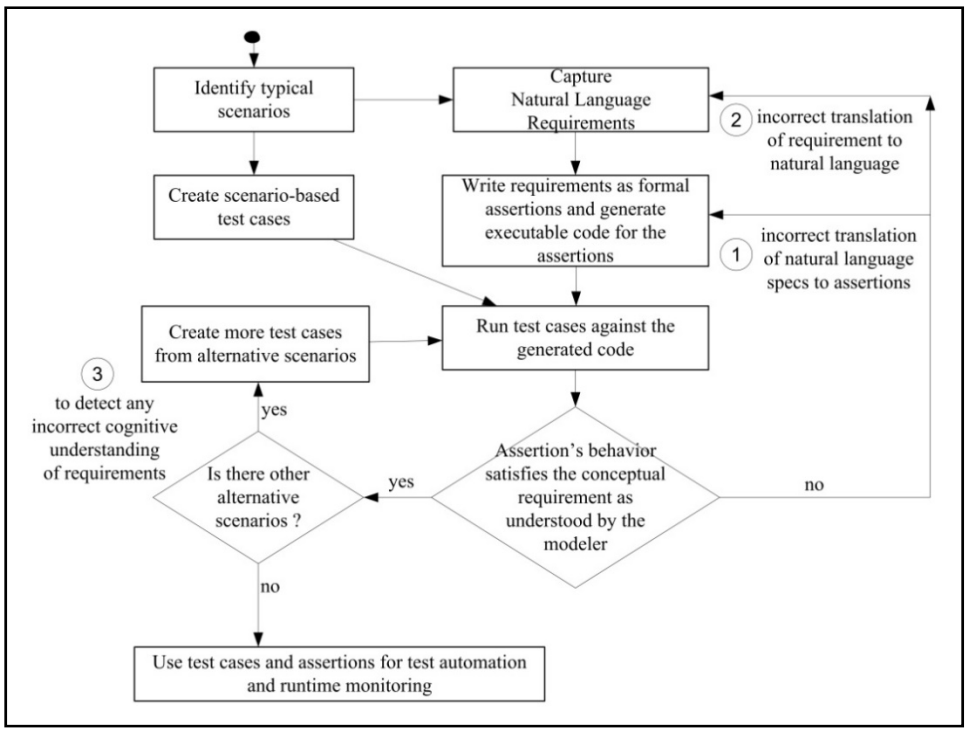

Figure 2. Iterative Process for Assertion Validation
(Based on Drusinsky et al., 2007)

Statechart Assertions

A statechart assertion is a UML statechart-based formal specification for use in prototyping, runtime monitoring, and execution-based model checking (Drusinsky, 2011). It extends the Harel (1987) statechart formalism and is supported by StateRover, a plug-in for the Eclipse integrated development environment (IDE). StateRover provides support for design entry, code generation, and visual debug animation for UML statecharts combined with flowcharts.

The statechart assertion extends Harel statecharts by adding a *bSuccess* Boolean flag and by enabling non-determinism. Statechart assertions are formulated from an external observer's perspective. Though the *bSuccess* Boolean is a simple mechanism, it is instrumental in determining whether an assertion ever fails. The Boolean indicates whether the assertion was violated by the system being analyzed. A statechart assertion assumes the requirement that it is based on is met (*bSuccess* = true), and it will retain that assumption unless a sequence of events leading to the violation of the requirement specified by the statechart assertion is observed. Once an assertion fails (i.e., reaches an error state), *bSuccess* becomes false and will stay false for the remainder of the execution. Because the statecharts are simple, it is easy to identify the assertion that failed and the cause.

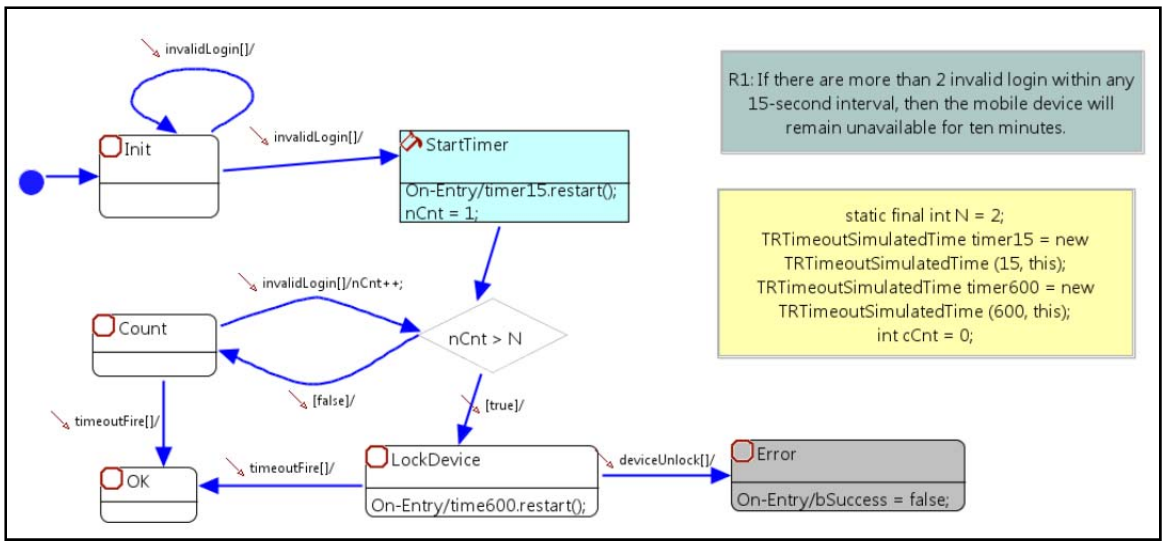

Figure 3. A Statechart Assertion for Requirement R1

Figure 3 shows a statechart assertion for the requirement R1, where the 10-minute interval starts immediately at the detection of the third *invalidLogin* event within a 15-second interval, according to the analyst's interpretation of the natural language requirement. The statechart is written from the standpoint of an observer, who is interested in the proper sequencing of two system events: *invalidLogin* and *deviceUnlock*. It uses two timers to keep track of the timing constraints in R1. Starting out in the *Init* state, the statechart transitions to the flowchart-action box *StartTimer* when it observes an *invalidLogin* event. It increments the counter *nCnt* and starts the 15-second timer, and then it checks to see whether the counter *nCnt* exceeds 2. If $nCnt \leq 2$, it enters the *Count* state. Whenever the statechart observes an *invalidLogin* event in the *Count* state, it increments the counter *nCnt* and then checks to see whether the counter *nCnt* exceeds 2. The statechart remains in the *Count* state until either the 15-second timer expires or until $nCnt > 2$. If $nCnt > 2$, the statechart enters the *LockDevice* state and starts the 10-minute timer. The statechart remains in the *LockDevice* state until either the 10-minute timer expires or until it observes a *deviceUnlock* event. If the statechart observes a *deviceUnlock* event in the *LockDevice* state, then it enters the *Error* state. The entry action for the *Error* state sets *bSuccess* to false, meaning that the requirement R1 has been violated.

The StateRover supports the specification of complex requirements using non-deterministic statecharts. Although deterministic statechart assertions suffice for the specification of many requirements, theoretical results show that non-deterministic statecharts are exponentially more succinct than deterministic Harel statecharts (Drusinsky & Harel, 1994). Non-deterministic statechart assertions provide a very intuitive way for designers to specify behaviors involving a sliding time

window. In the statechart assertion shown in Figure 3, there is an apparent next-state conflict when an event *invalidLogin* is observed in the *Init* state. StateRover uses a special code generator to create a plurality of state-configuration objects for non-deterministic statechart assertions, one per possible computation in the assertion statechart. Non-deterministic statechart assertions use an existential definition of the *isSuccess* method, where if there exists at least one state-configuration that detects an error (assigns *bSuccess*=false), then the *isSuccess* method for the entire non-deterministic assertion returns false. Likewise, terminal state behavior is existential: If at least one state configuration is in a terminal state, then the non-deterministic statechart assertion wrapper considers itself to be in a terminal state.

For example, the statechart assertion in Figure 3 generates four state-configuration objects for the test scenario shown in Figure 4 at runtime, one for each *invalidLogin* event. The state-configuration object that starts with the second *invalidLogin* event ends up in the *Error* state, causing the *isSuccess* method to return false to the test driver.

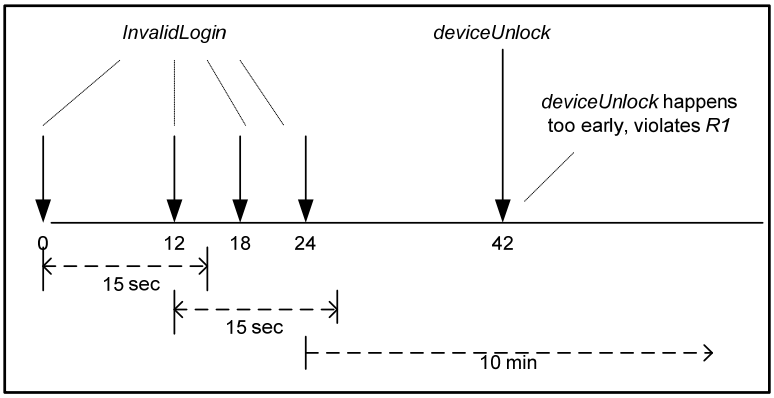

Figure 4. An Exception Test Scenario for Statechart Assertion R1

Validation of Statechart Assertions

StateRover's code generator generates a Java class R1 for the statechart assertion file. The generated code is designed to work with the JUnit Java testing framework (Beck & Gamma, 1998), as illustrated in Figure 5.

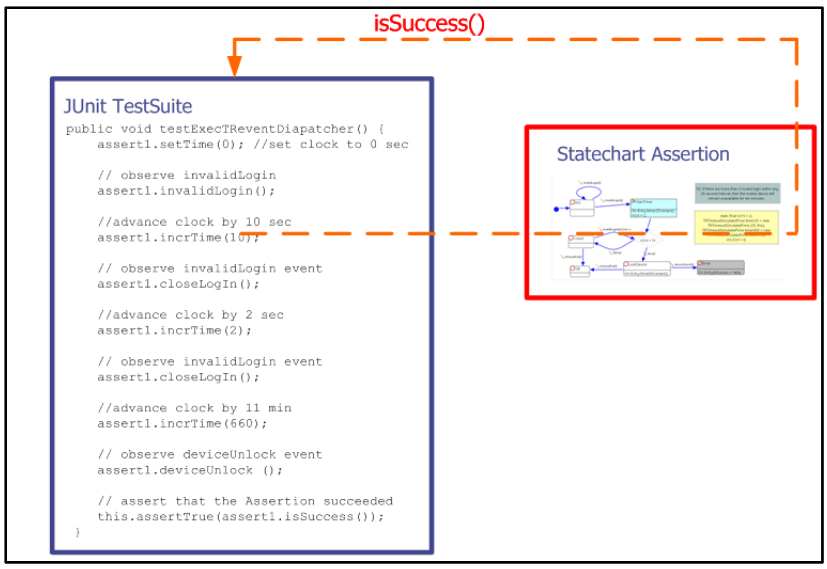

Figure 5. Validating Statechart Assertion via Scenario-Based Testing

To ensure that the statechart assertion works as specified in R1, we test its behavior using the JUnit test cases corresponding to the different scenarios shown in Figure 6 and the one shown in Figure 4.

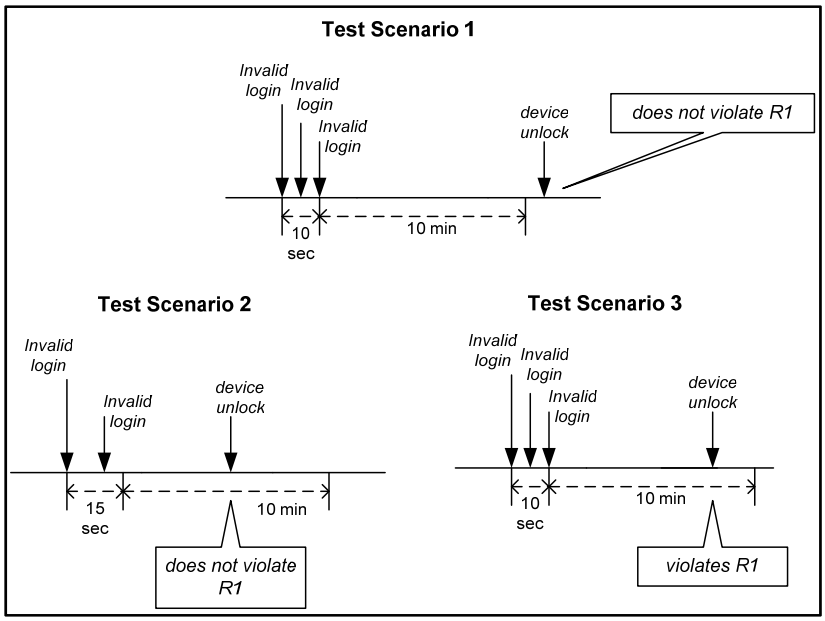

Figure 6. Test Scenarios for Statechart Assertion R1

Test scenarios 1 and 2 in Figure 6 represent two typical “happy” scenarios. Test scenario 1 expects the system to detect the three *invalidLogin* events within a 15-second interval and then lock the device for 10 minutes. Test scenario 2 expects

the system to keep the device open because it observes only two *invalidLogin* events within a 15-second interval. Test scenario 3 in Figure 6 and the test scenario in Figure 4 represent two exception scenarios, where the system allows the device to be unlocked too early, causing the statechart assertion to enter the *Error* state, thereby signaling that the assertion detected a requirement violation.

Logfile-Based Runtime Verification of Mobile Apps

Alves. Drusinksy, Michael, and Shing (2011) presented an end-to-end process that begins with a system requirement as a natural language specification, followed by the creation and computer-aided validation of UML statechart-formal specification assertions, and ends with the logfile-based runtime verification of target system (see Figure 7). The log files collected from the system execution were converted into JUnit tests and were run against the assertions in the assertion repository. If the results of the JUnit tests are positive, then the software application satisfies the requirements specified by the statechart assertions. If not, the software application has violated one or more requirements specified by the state assertions. There can be one or more possible causes of the problem. A detailed analysis of the execution trace using the StateRover's animation debug tool is needed to determine whether the cause of the violation was software errors due to incorrect implementation of the software or incorrect statechart assertions due to unexpected operating conditions and scenarios. Actions are then taken to correct the problems, and the verification cycle is run again.

Alves et al. (2011) applied the process to the specification, validation, and verification of the critical time-constrained requirements of the Brazilian Satellite Launcher flight software, and uncovered several inaccuracies in the requirements understanding and implementation.

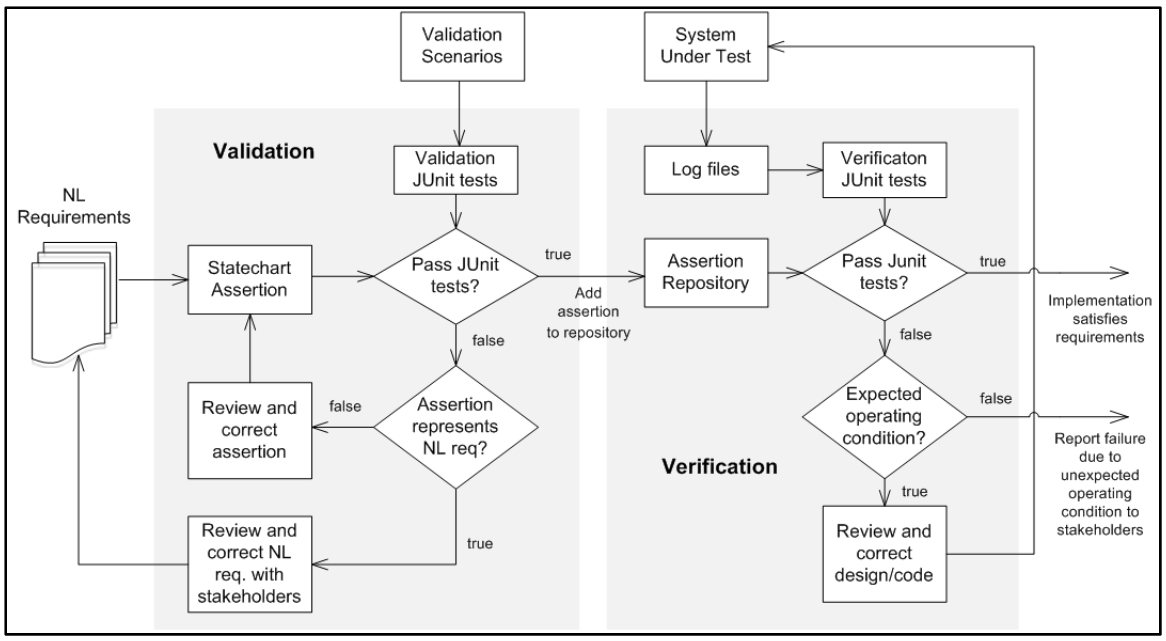

Figure 7. An End-to-End V&V Process

Computer-Aided Process for the V&V of Mobile Apps

We shall apply a similar process to the V&V of mobile apps, which consists of the following steps:

1. Subject-matter experts determine the properties of interest and the metrics to verify/measure those properties in the lab.
2. The properties are then expressed precisely as statechart assertions, whose correctness is validated via runtime verification.
3. The mobile devices and applications are then instrumented, if needed, for data collection and log file generation.
4. The instrumented codes are deployed to the field via mobile apps downloads. Metric data are collected in log files while the mobile devices are being used in the tactical environment, and the log files are uploaded back to the lab while the mobile devices are being recharged.
5. The log files are then converted into JUnit tests, and the tests are run against the statechart assertions in the lab. The test results are analyzed and reported.

Using log files produced by mobile apps brings two benefits: (1) it captures the behavior of the application on an actual, physical device; and (2) the data generated from the execution of the application on a device that is fully mobile is representative of expected normal operation of the application. Therefore, we can

analyze the log files to determine whether the behavior was correct based on the requirements. As demonstrated in the next section, we do not need to instrument the mobile device or its software if the events of interest are derivable from the output data of the mobile apps.

Case Study

The case study involves a smartphone application that uses a global positioning system (GPS) to track the location and speed of a person in motion. A log of the collected GPS data must be kept in the smartphone until it can be uploaded to a server via Wi-Fi connection. GPS applications can consume a lot of power and storage space and since mobile devices have limited amounts of both, minimizing the consumption of both is important.

Due to the limited available storage space on the mobile device we must minimize the amount of GPS data stored. The method chosen to accomplish this is to adjust the rate at which the GPS updates occur to be based on the speed at which the user is traveling. An additional requirement is that the log file must be able to be transmitted from the device to a server by a Wi-Fi connection only, since many of the users will not have wired connectors for the devices. If at any point Wi-Fi connectivity is lost and there is an active transmission, it must be terminated. The application has a limit of 30 seconds to transmit the log file, after which, if not successful, the user must be notified of the failed transmission within five seconds. Additionally, a log file must not be transmitted within one hour of a previous log transmission. Both the use of a time-limited transmission window for the log file as well as an infrequent upload of the log file will aid in reducing the amount of power and bandwidth the application consumes.

The GPSTracker Application

The GPSTracker application is implemented as an iOS app. As such there are several iOS framework classes included in this app. These framework classes are generic and particular to iOS apps. Because they are not relevant to the understanding of the domain-specific logic, no framework classes are included in Figure 8 (except the UIWindow class, displayed in brown in the diagram).

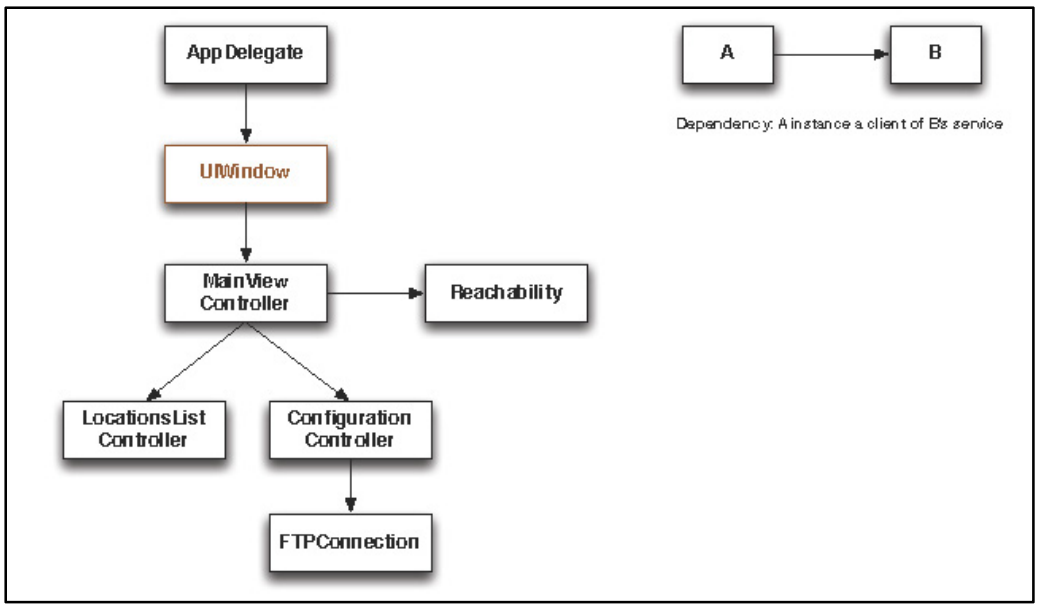

Figure 8. Simplified Class Dependency Diagram for the GPSTracker Application

The GPSTracker software consists of the following seven classes:

- AppDelegate—This is the top-level "main" class of the application.
- UIWindow—This is a framework class for managing the window on the screen. Each application has exactly one window. The content of the window is called a view. An application is built from one or more views. Each view is managed by a view controller. In this application, we have three views: main, location list, and configuration (see Figure 9).
- MainViewController—This is a controller for managing the main view. The main view consists of a single toggle button for starting and stopping the tracking. It also contains two small icons at the bottom of the view to access the location list and the configuration views.
- LocationListController—This is a controller for displaying the GPS locations tracked so far. The user has the option of clearing the list.
- ConfigurationController—This is the third controller in the application. The user enters the necessary information of the server to which the GPS tracking data are uploaded. The current implementation requires the host server IP address, the user account, and the password.
- FTPConnection—This a service class for managing the file transfer protocol (FTP) connection for uploading the data to the FTP server.

- Reachability—This service class is for keeping track of the network connectivity. Information such as the availability of Wi-Fi connectivity is managed.

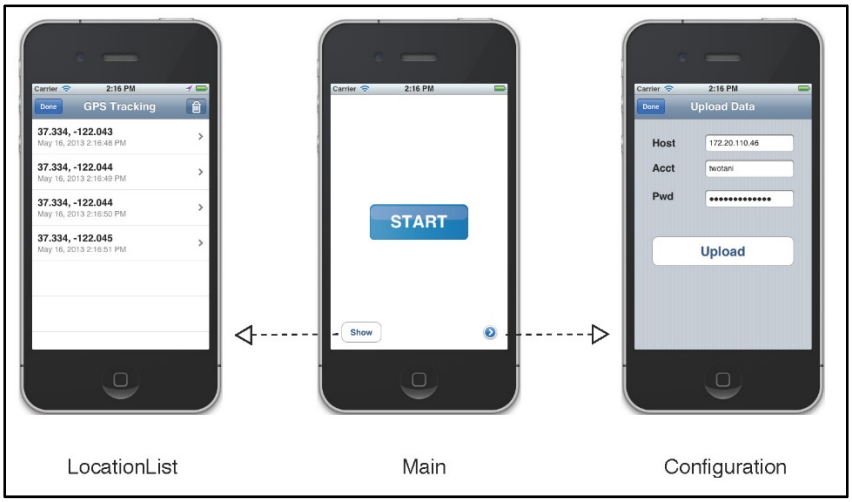

Figure 9. Screenshots of the Three Views of the GPS Tracker Application

The StateRover Specification, Validation, and Verification Environment

The case study uses the StateRover tool that runs on the Microsoft Windows 7 Professional operating system and the Eclipse (version Indigo) software development environment to create and evaluate the statechart assertions for the GPSTracker application.

After downloading and starting Eclipse, the standard setup for Eclipse will be shown. To install the StateRover tool, select “Install New Software” under the Help menu in Eclipse. The location of the menu is shown in Figure 10.

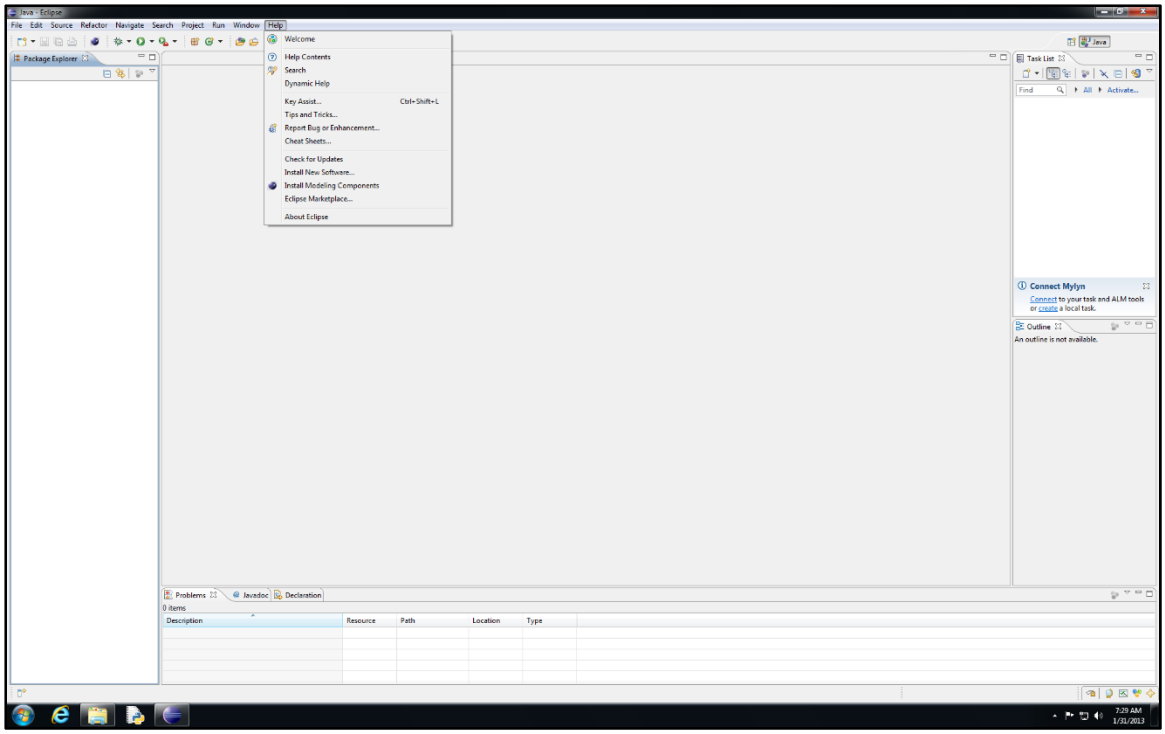

Figure 10. Eclipse Version Indigo on Windows 7

Select the "Add" button, and enter the address http://www.time-rover.com/updates/staterover_Team_3_6 along with a name for the site. A user name and log-in is necessary to download the files. Make sure to deselect the "Group Items by Category" check box. Select all six files that appear and select "Next." Figure 11 shows the files that need to be downloaded. Ensure that there are no errors on the next screen, and select "Finish." Once installed, the user can refer to the "StateRover User Guide" in the Help Contents/Time-Rover folder under the Help menu in Eclipse for more details of StateRover set-up and usage.

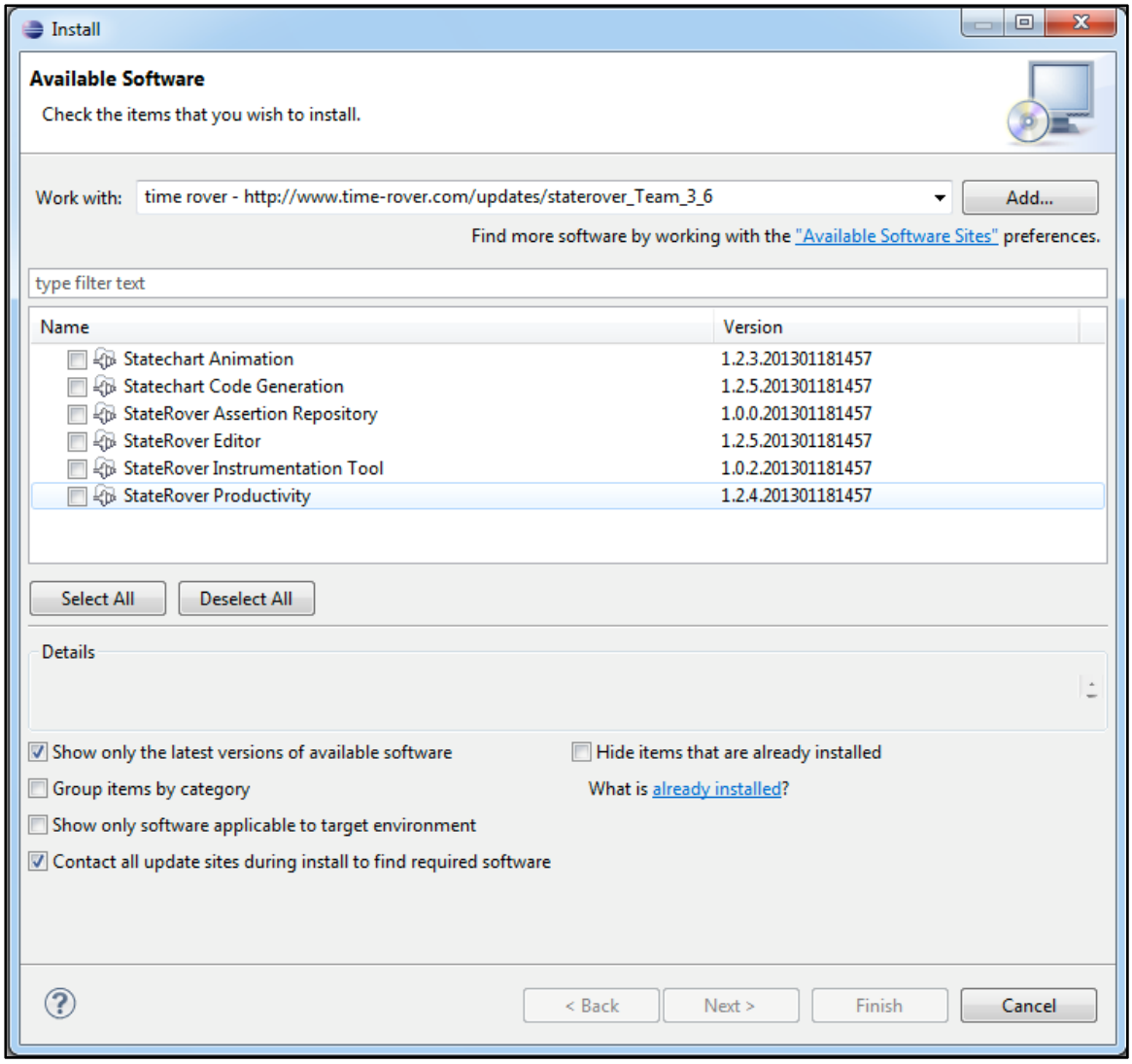

Figure 11. StateRover Installation Into Eclipse

Next, create a new project for the case study. Select “New” under the File menu in Eclipse, and then select “Project.” Select the folder “Java” and then “Java Project.” Eclipse will ask for a project name. Enter a unique name for the project and click the “Finish” button.

Add the two libraries to the project build path. Select “Properties” under the File menu in Eclipse. Select “Java Build Path” in the left panel of the pop-up menu and the “Libraries” tab in the right panel, and then select the “Add External Jars” button. For our setup, the files are located in the C:/eclipse/plugins/ com.timerover.assertionrepositoryjars_1.0.2.201205162118/ folder. Add both the TReclipseAnimation.jar and stateroverifacesrc.jar files. After adding them, select the “Add Library” button. This will allow the addition of JUNIT4 test suite. Select “JUNIT”

and then the “Next” button. In the dropdown menu, select “JUNIT 4” and click “Finish.”

A small amount of configuration needs to be performed, and an assertion repository must be created before building statecharts. The assertion repository allows several statechart assertions to be evaluated concurrently for each log file. Right click on the new project name in the Package Explorer and select “Toggle Assertion Repository.” To make the Java code create automatically for your project, right click on the project name again and select “Enable Statechart Code Generation.” A quick tip is that when creating a new project, the “Enable Statechart Code Generation” may appear to be already checked, but it is not. We suggest the that user uncheck and then re-check it again to ensure that it is actually checked.

Generally, Eclipse will ask if the user would like to view the “Statechart Diagram Perspective.” A perspective configures the user’s view for the particular type of coding he/she is doing. This will allow the user to see the properties of statechart elements. Select “Yes.” If the user was not asked, the perspective can be selected manually. In the upper right, the user can see the perspective selector buttons. “Statechart Diagram Perspective” should be available and should be selected.

Next, add a package for each assertion diagram to the project folder. A package is Java’s way of organizing the files needed to create a software program under development. Right click on the project name in the Package Explorer and select “New,” and then select “Package.” A window will open to ask for a name for the package. Enter a unique name for the package and click the “Finish” button. To create a statechart assertion in the package, right click on the package name in the Package Explorer and select “New,” and then select “Statechart Assertion Diagram” under the Other/TimeRover folder. A window will open to ask for a name for the statechart assertion. Enter a unique name for the statechart assertion. Ensure that the name ends with “.statechart_diagram.” Click “Next” and then “Finish.” Edit the details of the statechart assertion in the drawing pane. The user may add as many packages in the same manner as needed.

Specification and Validation of the Statechart Assertions

When a user is traveling at a slow speed like walking, frequent updates are unnecessary because significant distance changes do not happen quickly. If the user is traveling at a faster pace, then more updates allow for more consistent tracking. When the user is traveling at less than or equal to two meters per second, the application should average five seconds or more per update. This is approximately the walking speed of a human (Cary, 2005). If the user is traveling at greater than two meters per second but less than or equal to five meters per second,

then there must be an average of between two and five seconds between updates. This will be considered the running speed. If traveling greater than five meters per second, then there must be an average of less than two seconds between updates. This will be the driving speed.

We decided to use an average time interval (in seconds) between updates due to the typically less-than-accurate GPS data provided by mobile devices. A requirement for an average over a minimum of five GPS update events will be included to reduce the effects of any lack of precision in the GPS data from the mobile device. Table 1 lists the requirements.

Table 1. Speed-Based Requirements

Speed Category	Average Speed (x) in meters per second	Expected Time between Updates (y) in Seconds per Update
Walking	x<=2	5<=y
Running	2<x<=5	2<=y<5
Driving	5<x	y<2

Drusinsky, Michael, and Shing (2007) stated that a model-based specification that uses a single, intertwined representation of the software requirements (e.g., as a single statechart) can become complex and difficult to understand due to the interaction of each requirement with others. They advocate the use of assertion-based specification, which allows the requirements to be decomposed into their simplest forms, and then create a formal representation (e.g., a statechart assertion) for each requirement. This decomposition allows a one-to-one connection between a statechart assertion and a customer requirement. A significant benefit of this connection is that it simplifies the development, analysis, and testing of the statechart assertions. Other benefits include the following:

1. There is a reduction of the statechart assertion complexity. Because the complexity of the statechart assertions is minimized, the statechart assertions are much easier to test for correctness.
2. The one-to-one connection between a statechart assertion and a customer requirement simplify the changes that need to be made to the assertions when the requirements change.
3. Statechart assertions can be made to represent a test for both negative and positive behaviors where as a model-based specification usually only captures positive behaviors.
4. Tracing unexpected behaviors to the one or more requirements that they violate is simpler because there is a one-to-one mapping.

Hence, we refined the speed-based GPS Update requirement into three requirements. Using the StateRover tool, we created the three statechart assertions shown in Figures 12, 13, and 14, one for each of the three speed categories of the speed-based GPS Update requirement.

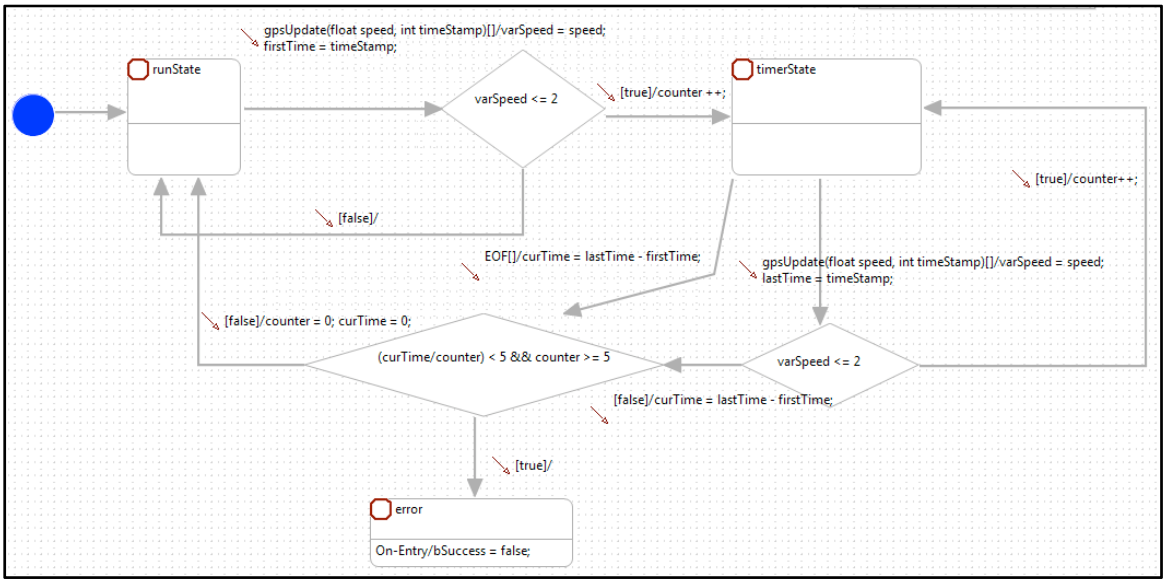

Figure 12. Statechart Assertion for Speed Less Than or Equal to Two Meters per Second

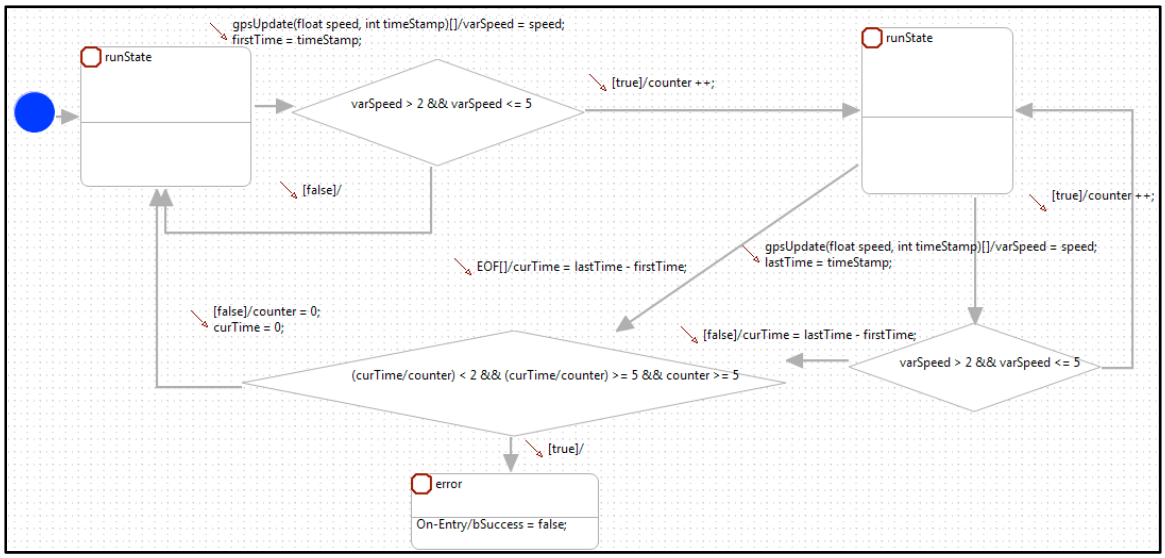

Figure 13. Statechart Assertion for Speeds Between Two and Five Meters per Second

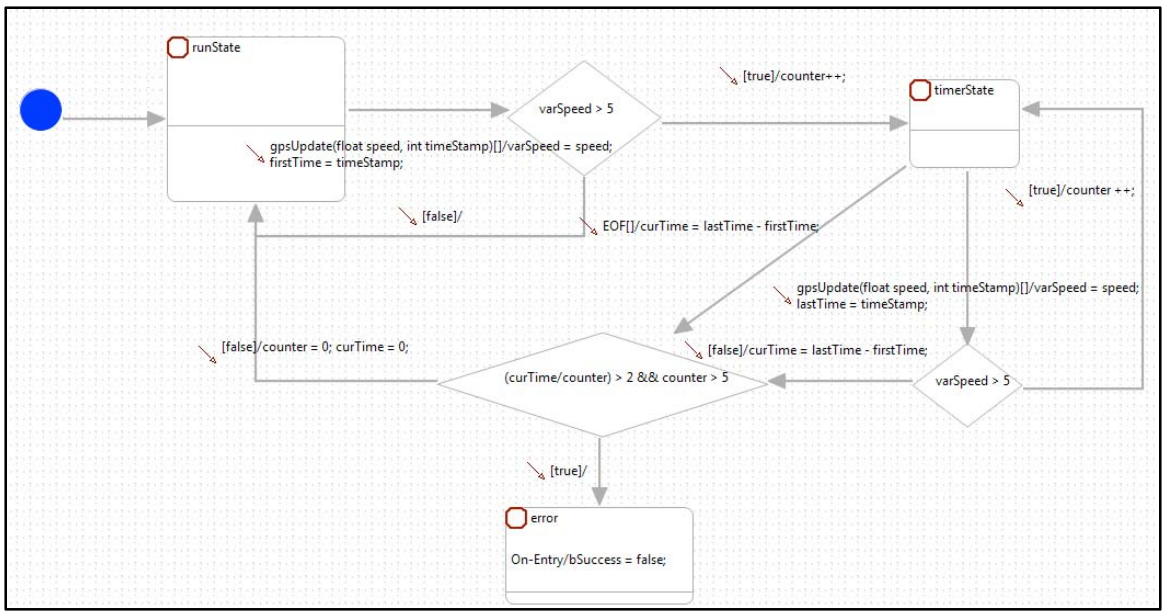

Figure 14. Statechart Assertion for Speeds Greater Than Five Meters per Second

Figure 15 shows the statechart assertion for the requirement that a log file can be transmitted only when the device is connected to a Wi-Fi access point. Note that this statechart assertion covers only the requirement that a transmission cannot start when not connected to Wi-Fi, but it does not capture the requirements that log files cannot be transmitted within an hour of each other, nor does it cover what needs to be done when the Wi-Fi connection is lost during a transmission. We chose to capture the latter with three other statechart assertions (Figures 16, 17, and 18), thus simplifying the complexity of each statechart assertion.

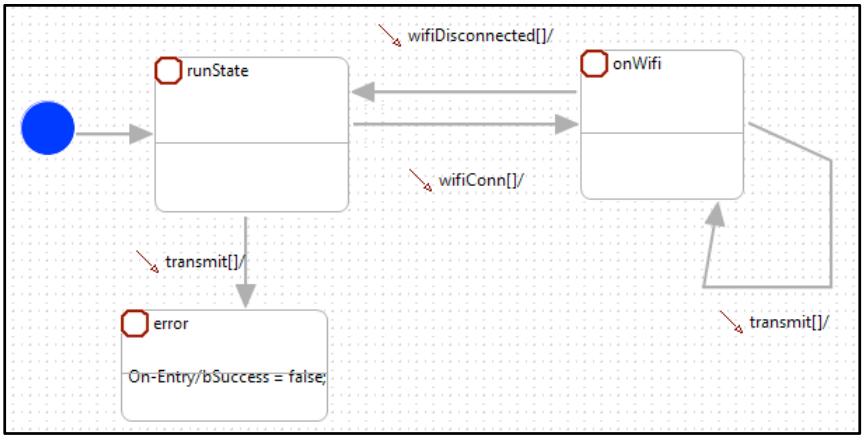

Figure 15. Statechart Assertion for WiFi-Only Transmission

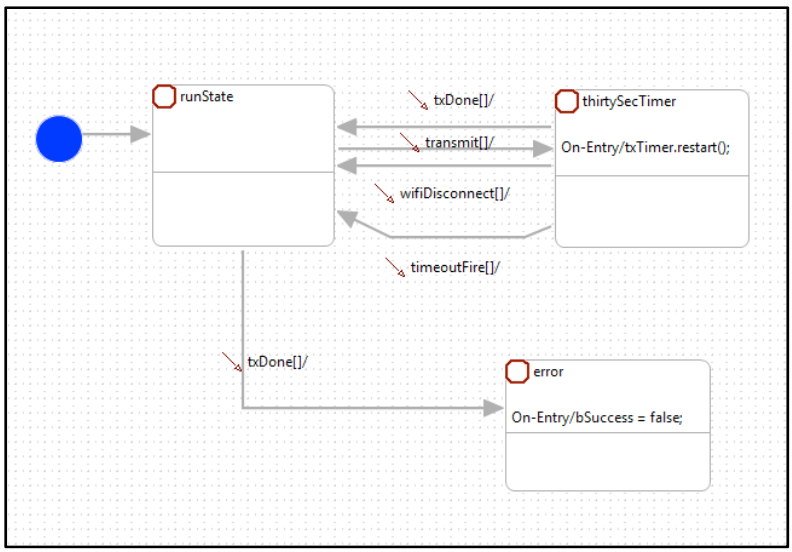

Figure 16. Statechart Assertion Limiting Log File Transmission Time to 30 Seconds

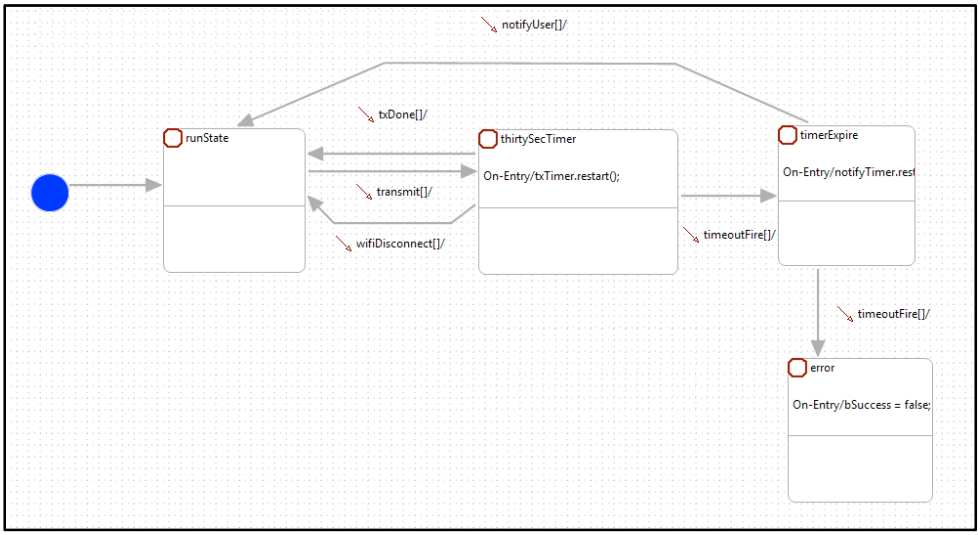

Figure 17. Five Seconds to Notify User of Transmission Failure

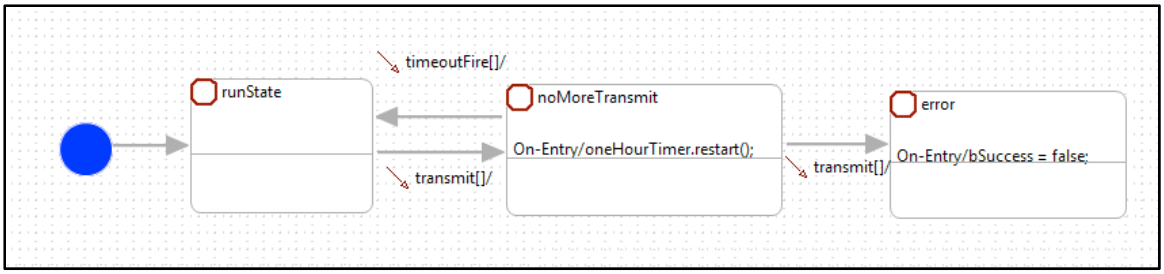

Figure 18. One Hour Time Out Between Successive Log File Transmission

We tested each of the above statecharts with different scenarios to ensure that they correctly capture the intent of the natural language requirements.

Log File Preprocessing and Runtime Verification

The GPS application generates log file with the data format shown in Figure 19, which is different from those required by StateRover, like those shown in Figure 20.

```
WIFI_CONN @ 11/30/2012 01:07:44 PM

WIFI_DISCONNECT @ 11/30/2012 01:07:45 PM
```

Figure 19. GPS Application Generated Data Format

```
<event>

<sig><![CDATA[wifiConn]]></sig>

<time lang="c" unit="sec" val="1354309664" />

</event>

<event>

<sig><![CDATA[wifiDisconn]]></sig>

<time lang="c" unit="sec" val="1354309665" />

</event>
```

Figure 20. StateRover Required Log File Format

In order to test the log file produced by the GPS application against the statechart assertions, we need to convert the original log into a log that can be read by the StateRover tool. We developed a Python script to convert the application log file into what we call a StateRover log file.

Because the timestamps in our log files are in seconds, we need to make sure that StateRover interprets the time increment between two consecutive events as seconds prior to importing the newly created StateRover log. Double click on the file named “AssertionTime.properties” located in the package named *assertionrepository* in the Package Explorer. There is only a single statement that defaults the AssertionTime unit to “milli” in the AssertionTime.properties file. Replace “milli” with “sec” and save the file.

Once that has been done, import the StateRover log. Right click on the project name in the Package Explorer and select “Import.” A window will open to ask for a type of file to be imported. Select “Import Log File and Convert into JUnit” under the TimeRover folder, and then click “Next.” Find the log file to be imported and click “Finish.” Once done, two files will be added to the project: the log file and an XML file. The XML file is added to the LegalXmlLogs folder and will be the file used by the StateRover to generate code. The StateRover’s logfile-to-JUnit converter converts the StateRover log file into an equivalent JUnit Java class. This class contained the logfile-based verification test for the statechart assertions.

The final step is to add the *namespace* map file to the project. This must be done after importing the log file because it needs to reference the log file. To add the file, right click on the project name in the Package Explorer and select “New.” Select “Other” in the pop-up menu, scroll down to the TimeRover folder, and then select “Namespace Map.” The map can be named to anything, but the default works fine. The field titled “New Source Log-File” needs to be set to the XML file created in the LegalXmlLogs folder during the import step. The field titled “New Target Assertion Repository” needs to be set to the location of the project in which you are working. Once entered, press the “Finish” button.

Using StateRover’s namespace mapping tool, we created a namespace mapping that linked the JUnit Java class’s name space (events as defined in the log files) to the assertion repository’s namespace (events of the Statechart assertions). Double click on the namespace map file (with suffix “.namespace_map”) in the Package Explorer to open the namespace map tool shown in Figure 21. The StateRover’s namespace map in Figure 21 depicts on the left side tree (denoted the source tree) events taken from a log file and, on the right side tree (denoted the target tree) events from all assertions in the assertion repository. Connections between the source and the target trees can be done manually using the user interface, or automatically using a built-in matching algorithm.

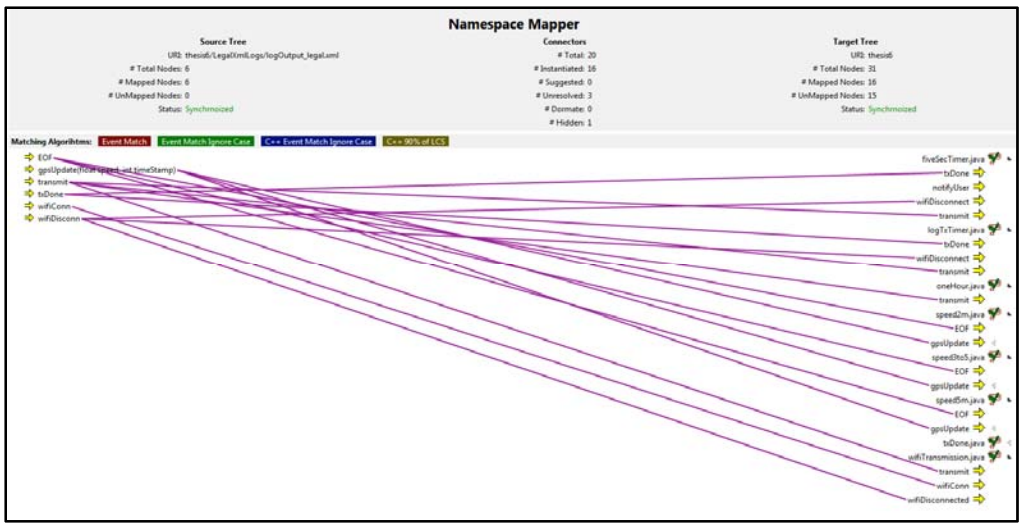

Figure 21. Namespace Mapping for Runtime Verification

Once this is complete, the test can be run by clicking the "Run" button in the tool bar. Figure 22 shows the desired result after testing one or more statechart assertions. If an assertion failure exists (i.e., a *bSuccess* variable in one of the assertions was set to false), the statechart assertion where it occurs will be listed on the left side under the header Statechart assertion failures.

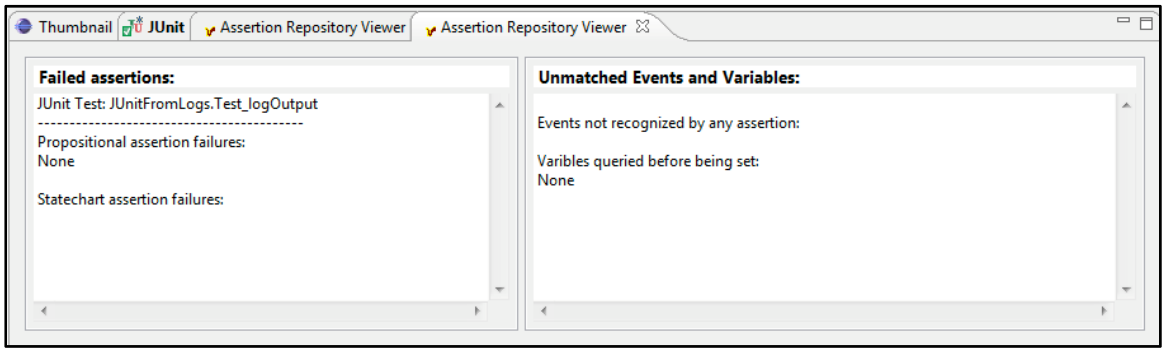

Figure 22. Test Result With Zero Failure

To validate the correct operation of the statechart assertions, we manually generated some log files containing errors. The log file in Figure 23 is an example snippet of such log file.

```
1.0959 mps @ 11/30/2012 12:11:20 AM
1.3764 mps @ 11/30/2012 12:11:21 AM
0.9190 mps @ 11/30/2012 12:11:22 AM
0.7197 mps @ 11/30/2012 12:11:23 AM
WIFI_CONN @ 11/30/2012 12:11:23 AM
TX_START @ 11/30/2012 12:11:23 AM
1.9180 mps @ 11/30/2012 12:11:24 AM
0.5781 mps @ 11/30/2012 12:11:25 AM
0.3186 mps @ 11/30/2012 12:11:26 AM
0.7450 mps @ 11/30/2012 12:11:27 AM
0.1642 mps @ 11/30/2012 12:11:28 AM
0.7080 mps @ 11/30/2012 12:11:29 AM
1.7338 mps @ 11/30/2012 12:11:30 AM
WIFI_DISCONNECT @ 11/30/2012 12:11:30 AM
1.3015 mps @ 11/30/2012 12:11:31 AM
1.5235 mps @ 11/30/2012 12:11:32 AM
WIFI_CONN @ 11/30/2012 12:11:32 AM
0.8866 mps @ 11/30/2012 12:11:33 AM
1.4841 mps @ 11/30/2012 12:11:34 AM
TX_DONE @ 11/30/2012 12:11:34 AM
3.0644 mps @ 11/30/2012 12:11:35 AM
4.0769 mps @ 11/30/2012 12:11:35 AM
3.2224 mps @ 11/30/2012 12:11:36 AM
3.5195 mps @ 11/30/2012 12:11:36 AM
2.0872 mps @ 11/30/2012 12:11:37 AM
```

Figure 23. Sample Log File Containing Erroneous Events

The events in the snippet show that the application that would have produced this would not have met the requirements. The device completed its log file transmission even after the device lost the Wi-Fi signal, and there were too many GPS updates while traveling at the walking speed. This is easy to see in a short log file, but if this were a log file that was several kilobytes or megabytes in size, then manually analyzing the file would be tedious and error-prone. Figure 24 shows the results of running the log file in StateRover. The log file failed the assertion statechart depicted in Figure 12 and Figure 16.

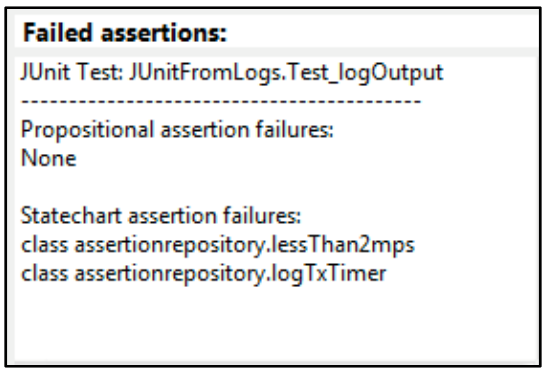

Figure 24. Failures After Using the Log File

Conclusion

This paper presented a method for performing V&V on a mobile application using statechart assertion and logfile-based runtime verification. The environment in which the DoD frequently operates is abnormal to say the least, and it is tough to emulate when attempting to perform V&V in a lab environment. It is important that an application is evaluated in the environment in which it is expected to operate, especially because the programers are probably unfamiliar with that environment, making it almost impossible to duplicate the environment in the laboratory. Log files provide direct insight into the operation of the application and, when used in the expected environment, can ensure a thorough and valid set of V&V tests. Combining the use of application log files and statechart assertions allows testers to evaluate the behavior of an application as it pertains to its adherence to the stated requirements. Statechart assertions provide a mechanism to represent application requirements into easy-to-follow diagrams that will be used by StateRover to automatically produce executable evaluators to evaluate the application log files. The modeling of the requirements independent of the implementation allows for multiple applications to be evaluated against the same set of requirements.

We demonstrated the method with a case study involving the V&V of a GPS mobile app. There are two different services one can use to get the user's current location: the standard location service, and the significant-change location service. The standard location service is a configurable, general-purpose solution and is supported in all versions of iOS. The significant-change location service offers a low-power location service that is available only in iOS 4.0 and later and can also wake up an app that is suspended. Initially in our case study, we attempted to use the significant-change location service to generate the log file, but this resulted in failure of the statechart assertions for the speed-based GPS update requirements. After switching to the standard location service with highest accuracy to generate the GPS updates, we were able to produce a new log file that satisfies the statechart assertions. It would be very labor intensive and difficult to manually determine

whether the new log file meets the requirements any better than the previous version. The StateRover's logfile-to-JUnit converter and the namespace mapping tool significantly ease the task of checking test results; we can quickly see that the new log file (and hence the new implementation) does indeed meet the requirements once we have imported the log file into StateRover. The methods for testing mobile apps, as discussed in The V&V of Mobile Apps section of this paper, all require manual evaluation of test results. The method put forth in this paper not only automates the checking of test results, it also allows testing of the application in the expected environment of operation. The case study provides a non-trivial example of how the use of log files and statechart assertions provides a significant improvement in the V&V process of applications.

References

Alves, M. C. B., Drusinsky, D., Michael, J. B., & Shing, M. (2011, June 27–30). Formal validation and verification of space flight software using statechart-assertions and runtime execution monitoring. In *Proceedings: Sixth International Conference on System of Systems Engineering* (pp. 155–160).

Beck, K., & Gamma, E. (1998). Test infected: Programmers love writing tests. *Java Report, 3*(7), 37–50.

Bo, J., Xiang, L., & Xiaopeng, G. (2007, March 20–26). Mobile test: A tool supporting automatic black box test for software on smart mobile devices. In *Proceedings: Second International Workshop on Automation of Software Test* (p. 8).

Carey, N. (2005). *Establishing pedestrian walking speeds* (Draft). Retrieved from http://www.westernite.org/datacollectionfund/2005/psu_ped_summary.pdf

Delamaro, M. E., Vincenzi, A. M. R., & Maldonado, J. C. (2006, May 23). A strategy to perform coverage testing of mobile applications. In *Proceedings: 2006 International Workshop on Automation of Software Test* (pp. 118–124).

Drusinsky, D. (2011). *Practical UML-based specification, validation, and verification of mission-critical software*. Indianapolis, IN: Dog Ear Publishing.

Drusinsky, D., & Harel, D. (1994). On the power of bounded concurrency I: Finite automata. *Journal of the ACM, 41*(3), 517–539.

Drusinsky, D., Michael, J. B., & Shing, M. (2007). *The three dimensions of formal validation and verification of reactive system behaviors* (NPS-CS-07–008). Monterey, CA: Naval Postgraduate School.

Drusinsky, D., Shing, M., & Demir, K. (2007). Creating and validating embedded assertion statecharts. *IEEE Distributed Systems Online, 8*(5), 3.

Easterbrook, S., Lutz, R. L., Covington, J. K., Ampo, Y., & Hamilton, D. (1998). Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering, 24*(1), 4–11.

Freierman, S. (2011, December 12). One million mobile apps, and counting at a fast pace. *New York Times*. Retrieved from http://www.nytimes.com/2011/12/12/technology/one-million-apps-and-counting.html

Gillett, F. (2012, April 23). Why tablets will become our primary computing device. Retrieved June 12, 2012, from http://blogs.forrester.com/frank_gillett/12-04-23-

why_tablets_will_become_our_primary_computing_device?cm_mmc=RSS-_-IT-_-71-_-blog_154

Harel, D. (1987). Statecharts: A visual approach to complex systems. *Science of Computer Programming, 8*(3), 231–274.

Kenyon, H. (2012a, January 11). Army sets tone for government's mobile enterprise with Android. Retrieved from http://defensesystems.com/articles/2012/01/16/army-mobile-secure-android-authentication.aspx

Kenyon, H. (2012b, January 27). DISA office to manage mobile devices, online app store. *Government Computer News*. Retrieved from http://gcn.com/articles/2012/01/27/disa-launches-program-office-to-manage-mobile-devices.aspx

Lesspainful device lab. (n.d.). Retrieved from https://www.lesspainful.com/documentation

Michael, J. B., Drusinsky, D., Otani, D. W., & Shing, M. (2011, November–December). Verification and validation for trustworthy software systems. *IEEE Software, 28*(6), 86–92.

Monkeyrunner tool. (n.d.). Retrieved from http://developer.android.com/tools/help/monkeyrunner_concepts.html

Muccini, H., Francesco, A. D., & Esposito, P. (2012, June 2–3). Software testing of mobile applications: Challenges and future research directions. In *Proceedings: Seventh International Workshop on Automation of Software Test* (pp. 29–35).

TestQuest automated testing. (2003). Retrieved from http://www.bsquare.com/products/testquest-automated-testing-platform

User scenario testing for Android. (n.d.). Retrieved from http://code.google.com/p/robotium/

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CA 93943

www.acquisitionresearch.net