End-to-end formal specification, validation and verification process: a case study of space flight software

Alves, Miriam C. Bergue

IEEE


Downloaded from NPS Archive: Calhoun

Miriam C. Bergue Alves, Doron Drusinsky, James Bret Michael, Senior Member, IEEE, and Man-Tak Shing, Senior Member, IEEE

Abstract—The quality of requirements and the effectiveness of verification and validation (V&V) techniques in guaranteeing that a final system reflects its established requirements have a direct influence on the quality and dependability of the delivered system. The V&V process can be efficient from a managerial point of view, but ineffective from a technical perspective, and vice versa. This paper presents an end-to-end formal computer-aided specification, validation, and verification (SV&V) process, whose feasibility and effectiveness were evaluated against the flight software for the Brazilian Satellite Launcher. Unified modeling language (UML) statechart assertions, scenario-based validation, and runtime verification are used to formally specify and verify the system, and metrics of the ongoing process and its V&V results are collected during the application of the process. The results of the case study indicate that the process and its computer-aided environment were both technically feasible to apply and managerially effective, will likely scale well to cater to SV&V of mission-critical systems that have a larger number of behavioral requirements, and can be used for V&V in a distributed development environment.

Index Terms—Astronautics, behavior, formal methods, metrics, process, requirements engineering, runtime execution monitoring, software, statechart assertions, verification and validation (V&V).

I. INTRODUCTION

MISSION-CRITICAL systems are unique in that they must be highly dependable. Complex real-world requirements are hard to assess for accuracy and are often difficult to specify and understand correctly [1]. As software is an important and increasing part of these systems, several techniques exist to conduct formal verification and validation (V&V), aiming to establish that the final system implements the flight requirements as statechart assertions, and the V&V process and its results. We present an SV&V process case study of a set of critical time-constrained requirements for the flight software of the Brazilian Satellite Launcher, using a computer-aided verification tool. However, because contemporary V&V tools are typically not well integrated into the development process, formal specifications that have been validated in the requirements phase are often not used “as is” in the verification phase, thus making it difficult to guarantee that the delivered system conforms to the system requirements.

In [6], Drusinsky et al. presented a computer-aided V&V framework where NL requirements are formally specified as human-readable and machine-processable statechart assertions, validation is based on scenario testing, and subsequent verification is accomplished using runtime execution monitoring.

The remainder of this paper is organized as follows. Section II provides a summary of the related work and Section III presents a brief overview of statechart assertions. Section IV describes our formal SV&V process. Section V presents the flight software under study, the formalization of the flight requirements as statechart assertions, and the V&V metrics of the ongoing process and its results. We present an SV&V process case study of a set of critical time-constrained requirements for the flight software of the Brazilian Satellite Launcher, using a computer-aided verification tool. However, because contemporary V&V tools are typically not well integrated into the development process, formal specifications that have been validated in the requirements phase are often not used “as is” in the verification phase, thus making it difficult to guarantee that the delivered system conforms to the system requirements.

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 Brazilian Institute of Aeronautics and Space, the U.S. Department of Defense, or the U.S. Government.
II. RELATED WORK

Formal V&V languages and techniques vary in cost, ease of use, required expertise, system coverage, and effectiveness. See [10] for a tradeoff space that treats the cost and coverage dimensions of formal V&V techniques.

In [11], Leveson et al. discussed the need for a specification language to create requirements models that minimize the semantic distance (i.e., the amount of effort to translate from one model to another) between the system’s requirements specification and the mental model of the system in the stakeholder’s mind as well as the semantic distance between the system’s requirements specification and its implementation. They identified state-based models as the specification language that comes closest in meeting these needs. Statecharts [9], [12], [13] and the requirements state machine language (RSML) [11], [14] are some examples of such languages.

Classical model checking [15] is a formal verification technique in which properties such as reachability, safety, liveness, and fairness are formally captured using a formal language. Some references in the literature [16]–[20] describe applications of model checking to verification of safety-critical systems. The SPIN model checker [21], a formal verification technique, uses propositional linear-time temporal logic (PLTL) [22], [23] for requirements specification. The main issue associated with PLTL is its inability to describe real-time requirements or time-series constraints. While model checking can perform complete verification of relatively small components, it provides for limited scalability due to the state explosion problem and its restricted support for the verification of a rather weak set of properties (e.g., properties expressible in PLTL or Buchi automata).

Jeffords and Heitmeyer [24] developed an algorithm for automatic generation of invariants (i.e., properties that hold in every reachable state of a state machine model) from the requirements specification to be presented to the system users for validation. The invariants are derived from specifications expressed in the software cost reduction tabular notation [25], [26]. Their approach can be used to supplement other techniques such as model checking.

Execution-based model checking techniques use REM, where system designers can formally specify mission-critical requirements for subsequent runtime verification and runtime recovery from specification violations. REM has also been used as a component of execution-based model checkers such as the Java Pathfinder [29], [30] and StateRover white box test generator [9].

In [31], message sequence charts assertions were used for specifying behaviors of a distributed network protocol, where simulation of the formal assertions, automatic scenario generation, and runtime monitoring were applied for V&V of the behavioral models.

The work presented in this paper combines statechart formal specification assertions and off-line runtime verification. Validation is based on JUnit tests while verification is based on JUnit tests created from the REM log files. In this way, our approach differs from the aforementioned ones in that V&V of requirements proceeds from specification to implementation using a consistent formal specification notation. In addition, the system under test (SUT) implementation does not need to be abstracted prior to verification.

Our approach also includes data collection integrated within the computer-aided formal V&V environment, data that is later used to assess the technical feasibility and efficiency of applying the process, and its results.

III. STATECHART ASSERTIONS

While statecharts have been part of the UML for many years, they are typically used either for documentation or for modeling and subsequent code generation. In contrast, a statechart assertion differs from a UML statechart in that it has a built-in Boolean flag $\text{success}$, indicating success or failure, thereby enabling Boolean specification of reactive system properties.

A statechart assertion is a formalization of a requirement, that is, a representation that a computer can process, written from an external observer’s viewpoint. The purpose served by the statechart assertion is to declare the detection of a requirement violation, also known as a requirement failure, by assigning $\text{success} \rightarrow \text{false}$. The StateRover Eclipse plug-in tool used in this research extends the statechart diagrammatic notation using Java as an action language, resulting in a Turing-equivalent notation, where code is generated from statechart assertions for subsequent V&V using scenarios. As such, statechart assertions monitor the inputs and outputs of the SUT for both validation and verification tests.

To validate a statechart assertion as a correct representation of its respective NL or cognitive requirement, validation JUnit tests are performed. They help in visualizing the true meaning of the assertions via scenario-based simulations, in which the tester constructs a plurality of scenarios, including nominal and failure scenarios, in order to validate the assertion’s behavior under normal and adverse conditions. Once approved, the statechart assertions become the reference model for the REM-based verification phase.

IV. A CHANGE IN THE V&V PARADIGM

The desire to raise productivity and reduce costs associated with software assurance has spurred on the adoption within the industry of computer-aided V&V tools and methods of automation. The V&V automation paradigm centers on the use of formal specification models and on the automated generation of source code, where the correctness of the models (or specifications) determines the correctness of the derived system [4]. A driving force on the cost side is to assure early on in the development lifecycle that the formal specifications
represent the correct understanding of what the system should and should not do, and the verifiability of the correct source-code generation based on these specification models. The new V&V paradigm, unlike that of the traditional V model shown in Fig. 1, emphasizes the analysis and testing of the specification followed by the verification of the implementation. Fig. 2 shows our adaptation of the V model in order to align our V&V process with the new paradigm, where the traditional acceptance testing is augmented with REM log file-based qualification testing to determine if the requirements of a specification are met at the coding, subsystem-integration, and system-integration phases of the system development.

A. V&V Process

The focus of V&V of reactive software systems is to guarantee that the system correctly implements its functionality and expected behavior, as prescribed by systems requirements. The semantic differences between the requirements model and its implementation in embedded control systems impose even more challenges for the V&V process [32]. While some control requirements are mathematically set in a continuous time, assuming concurrent execution and instantaneous switching in the problem domain, the final target code is running on a digital computer platform, subjected to sampling intervals, round-off errors, and communication delays, possibly producing different system behavior. Thus, it is necessary to dynamically verify the requirements specified in NL against the final code running on the target environment [33].

B. Specification and Validation Activities

The requirements specification and validation follow an abstract-validate-refine process, as shown in Fig. 3, with the following steps:

1) Requirements modeling: statechart assertions are created for a set of critical NL requirements; they are a computer executable version of the NL requirements.

2) Validation: a set of test scenarios is created for every assertion with the purpose of assuring that the assertion conforms to the intent of its corresponding NL requirements. Some scenarios are designed to satisfy the requirement, while others capture event sequences that deliberately fail the requirement, expecting the assertion to flag the requirement violation. In other words, validation testing assures the assertion being validated is a good representative of the NL requirement.

3) Failure of a validation JUnit test: an unexpected validation-test outcome occurs when either the assertion fails when it is expected to succeed, or vice versa. When this occurs, the scenario in question is further examined to determine which of the following issues caused the unexpected assertion behavior: a) a “bug,” that is, the statechart assertion exhibits a behavior different from what the developer expects; b) the developer misunderstood the requirement (e.g., when the NL requirement is ambiguous) thereby creating an erroneous statechart assertion; and c) the validation test contains a error (i.e., a “driver error”). In the first case, the statechart assertion is reviewed for correction and the second step is repeated. In the second case, the NL requirement is reviewed with the stakeholder to uncover missing or ill-
specified behaviors and the first step is repeated. In the third case, the test itself has to be corrected.

C. Verification Activities

Fig. 4 describes the verification framework based on an abstract-verify-refine process. It involves the following steps.

1) **Runtime execution monitoring**: data is gathered in a log file by observing the system behavior in its real-time or simulated execution environment.

2) **Log-file-based verification**: a set of JUnit verification tests is autogenerated from the log files. The tests are executed against the assertions.

3) **Failure tests**: At this time, all assertions are expected to succeed. Verification tests that fail an assertion are examined to determine whether they are due to an implementation error or due to abnormal system execution. In the latter case, the problem is reported to the stakeholders. Otherwise, the process proceeds.

4) **Design review**: feedback is given to the system-design and implementation team. Given the updated design and code, the loop proceeds to the first step.

V. Flight Software Formal SV&V

A. Specification

For our case study, the SUT is the flight control software of the Brazilian satellite launcher. The software is responsible for the control of the launcher, except for the launcher destruction, from a few minutes preceding liftoff until the satellite has been deployed into Earth’s orbit. The software is also in charge of the main launcher’s subsystems verification (e.g., inertial platforms, autopilot chain, and sequencing chain) during flight preparation.

The external interfaces of the flight software include a variety of analog and digital components and, as a consequence, the interface has to be able to detect and recover from error conditions resulting from interaction with the embedded environment. The algorithms and control loops, which operate during the powered and ballistic phases of flight, compare the measured state of the vehicle (i.e., position, speed, acceleration, and attitude) with the desired state, and generate pitch and roll guidance commands in order to minimize the difference between the measured and desired state. The flight events sequence characterizes the different stages of the flight and determines when and which algorithms and control loops have to be executed during the flight. As a critical part of the software, with very tight time-constraint requirements, we chose to formally specify this sequence. The sequence is specified by 44 requirements, representing approximately 80% of the behavioral requirements of the flight phase.

There are two distinct types of events in the flight sequence: reference events and relative events. The reference events are the main events in each flight stage; the time of their occurrence dynamically determines the occurrence time of the relative events. The relative events determine a certain set of actions to be performed at each stage of flight.

Reference events must occur in a predetermined timeframe, and depend on the time occurrence of the previous reference events during the flight. Consider the following the four reference events:

1) reference event A: LiftOff;
2) reference event B: ThrustDrop_1Stage;
3) reference event C: ThrustDrop_2Stage;
4) reference event D: ThrustDrop_3Stage.

We created a statechart-assertion specification for all reference event and relative event requirements based on the NL specifications; the NL specification for reference events A and B requirements is as follows.

1) Req_Ref_A: Once the navigation starts (time = 0), A must occur (only) within the time interval $[l_A, u_A]$.
2) Req_Ref_B: B must be detected (only) within the time interval $[l_B, u_B]$ of the detection of A.

Fig. 5 shows the statechart assertion for Req_Ref_B (superimposed on Req_Ref_A). It enters Error_A state when it
Fig. 6. Statechart assertion for the Req. Rel., B_i.

observes the event A while in stateNAV, On, before event A’s interval of occurrence lower limit (uA) has been reached (uA is manifested in Fig. 5 by the timeout value timer_LA) or when in state Waiting_A, it observes that event A’s interval of occurrence upper limit (uA) has been reached (uA is manifested in Fig. 5 by the timeout value timer_LA) and A was not detected. The statechart assertion enters the Error_B state when it observes the event B while in state A, before event B’s interval of occurrence lower limit (uB) has been reached (uB is manifested in Fig. 5 by the timeout value timer_LB) or when in state Waiting_B, it observes that event B’s interval of occurrence upper limit (uB) has been reached (uB is manifested in Fig. 5 by the timeout value timer_UB) and B was not detected. StateRover’s code generator creates a Java class for every statechart-assertion file that is integrated with the JUnit Java testing framework [34].

There exist five relative events that must be detected within a pre-established time interval after the detection of event B, with a tolerance of m milliseconds. Let \( \delta_i \) be the earliest occurrence time of the relative events \( B_i, i = 0, 1, \ldots, 4 \). The relative event requirement \( B_i \) is as follows:

\[
\text{Req. Rel.} - B_i: \text{whenever} B_i \text{is detected, then} B_i \text{must occur between} \delta_i \text{and} \delta_i + m \text{milliseconds afterward.}
\]

Fig. 6 illustrates the statechart-assertion formalization pattern for these requirements. The associated Java class generated by StateRover was then refactored for each of the relative requirements and a special Java function was created to set the corresponding \( \delta_i \) for each \( B_i \).

The first part of the statechart assertion in Fig. 6 formalizes the detection of the events A and B, as explained previously. The remaining part (states colored gray) formalizes the generic relative event requirement \( B_i \) (in Fig. 6 event R.E stands for relative event \( B_i \)). The statechart assertion will enter the Error_Rel state if it observes the R.E event when in state B, before its \( \delta_i \) timeout has expired (\( \delta_i \) is manifested in Fig. 6 by the timer2, which is the timeout value for the \( i \text{th} \) instance of the assertion), or when in state Waiting_Relative for more than m milliseconds (m is manifested in Fig. 6 by the timer3).

The requirements for reference events C and D are as follows.

1) \( \text{Req. Rel.} - C \): C must be detected within the interval \([IC, uC]\) after the detection of A.
2) \( \text{Req. Rel.} - D \): D must be detected within the interval \([ID, uD]\) after the detection of C.

Because a statechart assertion represents a corresponding flight-sequence requirement, there is a potential overlap between the statechart assertions—an overlap induced by requirement dependences. As a result, the set of validation tests created to validate Req. Rel.A, for example, could be reused as part of the validation tests for Req. Rel. B, Req. Rel. C, and Req. Rel. D.

The use of the StateRover tool and the creation of the statechart-assertion patterns for the relative-events requirements were instrumental in rapidly creating a large plurality of assertion instances. The initial requirements analysis, the creation of statechart assertions and their validation, took approximately three months of work by one subject matter expert in flight-event sequencing. Two more months were spent in verification testing with the assistance of another expert, which included the simulations and REM.

B. Validation Tests

JUnit tests were hand coded. They consisted of sequences of events and timing information. In order to cover all scenarios of interest in the statechart assertion, the creation of the JUnit validation tests followed the following approach [35].

1) Obvious success: test a trivial scenario that conforms to the NL requirement.
2) Obvious failure: test a trivial scenario that violates the NL requirement.
3) Full scenario success: test a nontrivial scenario that goes through the entire basic scenario while in agreement with the NL requirement.
4) Full scenario failure: test a nontrivial scenario that goes through the entire basic scenario while violating the NL requirement.

Let us take the relative event \( B_i \) (denoted as R.E in Fig. 6) requirement as an example. \( B_i \) is supposed to occur 1000 ms after B; 2) a failure scenario, where the event R.E occurred too late, after 1000 + m ms (m = 50); and 3) a failure scenario where R.E occurred 1 ms before its interval-of-occurrence lower limit had been reached. Scenarios 2 and 3 violate the requirement.
C. Verification Tests

The flight control software executes on a target computer governed by the VxWorks real-time operating system (RTOS); the RTOS communicates with the embedded environment and the host computer (Fig. 8), where the log files are created during execution. The StateRover tool provides automatic code instrumentation [36] that is optimized to collect the sequences of flight events and their corresponding time of occurrence. The relationship between the statechart assertions, the statechart-assertions Java code, the instrumented source code, and the log files is shown in Fig. 9. Fig. 10 presents an excerpt of the instrumented code and the resulting log file. Four log files were created and analyzed to date. Using StateRover’s log-file-to-JUnit converter, the log files were imported into the offline verification environment, and converted into an equivalent JUnit Java class. This class contained the log-file-based verification tests for the statechart assertions.

StateRover’s namespace mapping tool was used to create a namespace mapping linked the SUT’s name space (events and variables as defined in the source code) to the assertion repository’s namespace. The StateRover’s namespace mapping in Fig. 11 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 were created algorithmically—using a built-in matching algorithm—with a few handpicked connections created manually via the user interface.

The verification tests were executed according to the scheme shown in Fig. 12. The main driver (assertionrepository.java) is in charge of running JUnit tests using the namespace mapping, thus verifying the statechart-assertion’s behavior against sequences of events (acceptance-test scenarios) collected during runtime system execution.

VI. PRODUCT EVALUATION

Table I summarizes the tests and their results for the V&V process. Two hundred and twenty JUnit validation tests were created to validate 44 requirements represented as statechart assertions. In the validation phase, approximately 40% of the tests were created to fail the assertion.

In the verification phase, we observed that approximately 30% of the assertions were violated due to late reference-events detection. The time in the log files was compared to the telemetry time data, which was collected under the same testing conditions, but with no instrumentation in the code. In both data sets, the events’ occurrence times were very similar, hence eliminating instrumentation overhead as a possible cause of delays. The late detection of some events is likely attributable to interface-communication delays and also to unrealistic timing requirements specified by the stakeholders. For instance, in one of the test scenarios, the event B (thrust drop of the first stage) occurred approximately 4 s after the upper limit time specified for its detection. In this specific case, the flight software would not detect the event B and the rest of the flight events that depend on this event-detection would be compromised, as they are serialized in time. Should the specified time interval be redefined? The answer for this question certainly relies on the judgment of subject matter experts. We believe that acceptable tradeoffs between the flight’s functional and environmental requirements can be found without sacrificing the desired performance standards for the mission.

The V&V results also indicate that there is room for improving the system requirements in the area of failure and fault management. For instance, the failure caused by out-of-time errors in the events sequence was not properly addressed in the NL requirements.

In addition to detecting implementation errors using automated V&V, the computer-aided process assisted in highlighting requirements ambiguities and implementation deficiencies (e.g., the absence of few events detection and fault management in the implementation code) that could only be discovered using runtime verification.

VII. PROCESS EVALUATION

Typical measurement frameworks are ambitious undertakings that often require substantial data collection and anal-
Berander and Jönsson presented an extended GQM approach that focuses on the most important measurements for an organization regarding a specific project [7]. Goals generate questions, which in turn generate metrics. Metrics in the context of this paper are understood to be the application of measurement-based techniques to the software V&V process and its products with the goal of providing significant and timely actionable information to product, project, or program managers. Nevertheless, the main focus of this section is to present a sample of some product and process information that was obtained from the SV&V data collection.

One of the key factors for a successful measurement framework is to start with a small set of goals, metrics, and measurements, and subsequently grow them incrementally as the organization matures. There is always a risk of choosing unsuitable metrics and measurements, resulting in an overwhelming amount of data. The StateRover’s embedded database collects data for predefined datasets during the ongoing V&V process. This productivity database was used to define the datasets for our metrics and measurements.

### Table I

<table>
<thead>
<tr>
<th>Validation Tests</th>
<th>Verification Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>220 (around five tests per assertion)</td>
<td>Four log files (four tests per assertion)</td>
</tr>
<tr>
<td>220 JUnit classes—one JUnit class per test</td>
<td>4 JUnit class—one JUnit class per log file</td>
</tr>
<tr>
<td>132 success scenarios (around 60% of the scenarios)</td>
<td>31 assertions passed in all tests (around 70% of the assertions)</td>
</tr>
<tr>
<td>88 premeditated failure scenarios (around 40% of the scenarios)</td>
<td>13 assertions failed at least in one test (around 30% of the assertions)</td>
</tr>
</tbody>
</table>
A. The Environment and Data Gathering

StateRoverDB plug-in (database) combines the Derby database engine and the BIRT Eclipse reporting plug-in. The StateRoverDB provided the data collection and report generation framework for our metric-generation process. The reporting tool was used to present selected data, as database views, collected automatically during the active execution of the V&V process. The tool provided us with an continuously updated overview of the progress of the V&V effort—an overview that helped us prioritize our V&V tasks, focusing on critical areas of assurance while avoiding unnecessary tasks.

B. Goals and Metrics

Metrics are normally responses to questions linked to the goals, with different questions yielding different metrics. We defined two categories of goals: 1) flight software product, and 2) V&V process. Examples of metrics are given as follows.

1) Product Goal: This validates the flight software requirements according to the expected system behavior, with specific emphasis on what the system should do, what the system should not do, and what it should do under adverse conditions.

Three possible associated questions and their corresponding metrics are as follows.

Q1: Did the validation tests cover all the flight events as specified by the users? Metric S1: Number of validation tests involving an event (system-events testing coverage).

Q2: Are the NL requirements easily understandable? Metric S2: Number of validation test runs per requirement and their results. This is an indirect measure of statechart-assertion rework since the corresponding validation tests are rerun after each statechart-assertion update.

Q3: Did the validation tests cover all the requirements? Metric S3: Number of validation tests per requirement. Note that metric S3 only provides a simple measure of test coverage. It does not measure the effectiveness or sufficiency of the tests. Structured query language queries retrieved information associated with the aforementioned metrics. Fig. 13 presents the resulting chart for metric S1 (with visualization restricted to flight events A, B, C, and D). We can observe that the number of tests that involves event A is greater than the ones involving events B, C, and D. This is due to the presence of event A in the tests involving events B, C, and D, caused by the dependence among the corresponding requirements.

A snapshot of metric S2 is illustrated in the report depicted in Fig. 14. At the time this metric was collected, the number of validation test runs per requirement was unbalanced, highlighting those requirements that require additional allocation of assurance resources to ensure that their counterpart statechart assertions are correct. Metric S3 is illustrated in Fig. 15; it depicts S3 for six selected requirements. In fact, the number of validation tests for each statechart assertion represents the number of scenarios covering at least all of its possible states and events.

2) Process Goal: Assessment of the effort spent on V&V activities.

Two possible associated questions and their corresponding metrics are as follows.

Q1: To what degree was the V&V process effective in finding and correcting problems in the requirements (i.e., manifested as errors in the assertions, developer misunderstandings of the requirements, and validation test-driver errors) during a certain time period (e.g., month)? Metric P1: Results of the validation tests per day during one month. The idea behind this metric is to observe if there is a reduction in the number of tests that fail the assertion when they were supposed to succeed, and if new failures were not introduced. Fig. 16 illustrates this metric.
Q2: How many tests were executed per day during a certain month?

Metric P2: Total number of validation and verification tests runs per day. This number tends to get smaller if the tests results are satisfactory and rework is not necessary anymore.

The collected metrics for the Brazilian Satellite Launcher Project can be compared with those of other software development projects that adopt the approach to V&V described in this paper.

VIII. CONCLUSION

We presented in this paper an end-to-end process for formal V&V of reactive systems. Having an ample specification of a set of real-world requirements, simple, formal, and visual requirements representation and their correct understanding are among the most prominent advantages that statechart assertions and validation tests can bring to early stages of software development projects.

Another equally important benefit includes checking requirements correctness with respect to the hardware and software platforms on which the system will run, by monitoring at runtime the system implementation’s behavior against the formal requirements model, without any additional language translation. Managerial and technical data collected by monitoring the V&V process and its results make it possible to document dependability cases for the product, system, or service, in addition to providing useful data for improving the quality-assurance process.

We presented a defined repeatable process with the requisite level of tool integration and automation to make distributed V&V efficient and effective, and demonstrated the application of the process on the flight software of the Brazilian Satellite Launcher. In addition, this paper, to our knowledge, is the first to document in the open literature the details of how to go about conducting distributed computer-aided formal V&V, in which software execution in its real embedded environment occurred in one location and the collected log files were analyzed in a different location. Distributed V&V is important for software development projects, in which multiple geographically dispersed project teams collaborate with one another. For instance, in the development of video games it is now uncommon for the developers of the game engine to be physically collocated with the developers of the other parts of the game or even the visual artists, storyline writers, music composers, or quality-assurance team [37].

Our research yielded several lessons learned about applying statechart assertions. Because part of the flight software requirements specification was close to the pseudocode level, it was challenging to abstract away from the design and flowchart-like logic (i.e., the accidents of a software entity according to Brooks [38]) and to specify the problem (i.e., the essence of a software entity). With practice, it is possible to write simple and readable specifications that can be understood by subject matter experts with a minimal knowledge of REM and other aspects of computer-aided formal V&V. Readability and simplicity are very important when the resulting statechart assertion is complex, chances are that it captures the design logic instead of the software requirements, which will have the opposite effect from what we aim for in formal V&V.

Our follow-on research includes further improving test coverage and requirements comprehensiveness, wider use of the StateRover database plug-in for process and test-quality assessment, and completing a case study of an industrial space ground system.

ACKNOWLEDGMENT

The authors would like to thank M. A. Romani, Institute of Aeronautics and Space, Brazil, for running the flight simulations and collecting the log files.

REFERENCES


