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

Alves, Miriam C. Bergue; Drusinsky, Doron; Michael, James Bret; Shing, Man-Tak

End-to-End Formal Specification, Validation, and Verification Process: A Case Study of Space Flight Software

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 requirements correctly, while avoiding the well-known garbage-in garbage-out principle—in this context, poor specification leads to poor software implementation [2]–[4].

Early automatic analysis of software system requirements has the advantage of helping analysts evaluate and reason about complex behavior, such as time-constrained and time-series sequencing behaviors [5]. To that end, natural language requirements (NL) must be expressed formally in machine-processable languages to enable eventual computer-aided V&V. 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 (REM). This paper builds upon that research and describes an end-to-end formal specification, validation & verification (SV&V) process, including a quantitative assessment—driven by the goal-question-metric (GQM) approach [7] of the SV&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 distributed verification [8]. We describe how computer-aided formal V&V, using the tool StateRover [9], addresses the challenge of correctly capturing NL requirements, converting the NL statements into formal requirement specifications, and then checking the formal specifications to ensure that they match the original intent of the stakeholders. Our approach enforces a one-to-one mapping between NL requirements and unified modeling language (UML) statechart assertions.

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 of those requirements. Section VI discusses the V&V results and Section VII presents some findings from applying the

Manuscript received August 9, 2011; revised March 6, 2012; accepted April 1, 2012. Date of publication November 12, 2012; date of current version September 25, 2013. This work was supported in part by the State of São Paulo Research Foundation under Grant 2010/08173-3.

M. C. B. Alves is with the Software Engineering Laboratory, Institute of Aeronautics and Space, São José dos Campos, SP 12228-904, Brazil (e-mail: miriammcba@iae.cta.br).

D. Drusinsky, J. B. Michael, and M.-T. Shing are with the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943 USA (e-mail: ddrusin@nps.edu; bmichael@nps.edu; shing@nps.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSYST.2012.2220591

V&V process, based on the GQM approach. Section VIII contains a summary of the benefits of the proposed process and lessons learned from the flight software case study.

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 to monitor the system runtime execution and check the observed behavior against the formal specification of the system. In [27] and [28], REM was used for formal runtime verification of two safety-critical systems. The Eclipse-based StateRover tool [9] uses 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 *bSuccess*, 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 *bSuccess* to 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

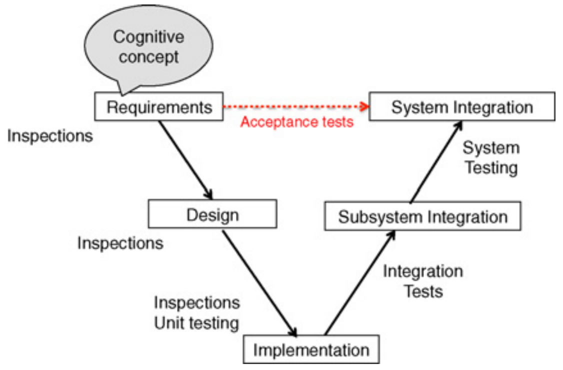


Fig. 1. Software development V model.

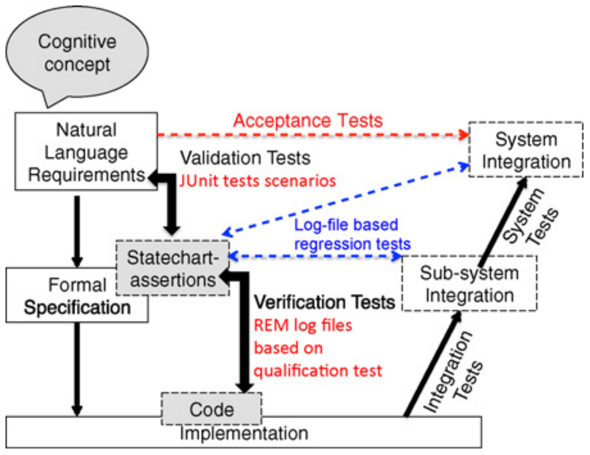


Fig. 2. Model-based development.

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].

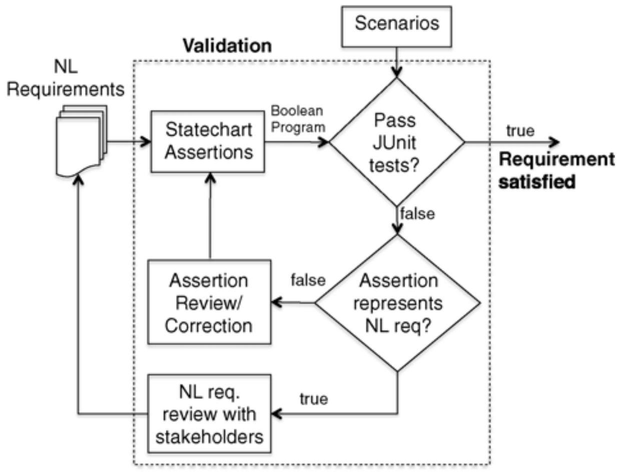


Fig. 3. Validation activities.

The proposed SV&V process is explained in the following sections. In the specific case of the flight software, we used distributed verification, where specification and validation activities were performed in Monterey, CA, generation of log files took place in São José dos Campos, Brazil, and requirements verification was conducted in Monterey, CA [8].

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-

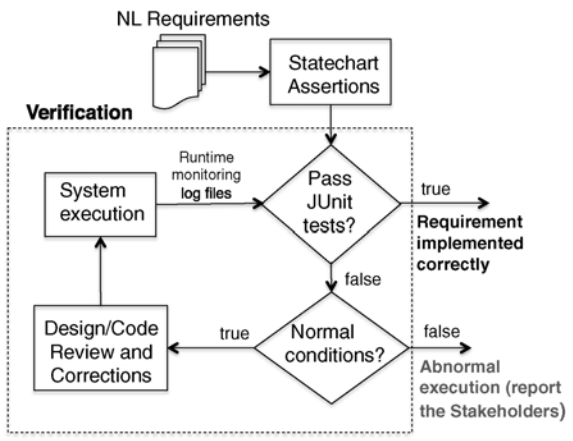

NL Requirements
Statechart Assertions
Verification
Runtime monitoring log files
System execution
Pass JUnit tests?
true
Requirement implemented correctly
false
Design/Code Review and Corrections
true
Normal conditions?
false
Abnormal execution (report the Stakeholders)

Fig. 4. Verification activities.

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

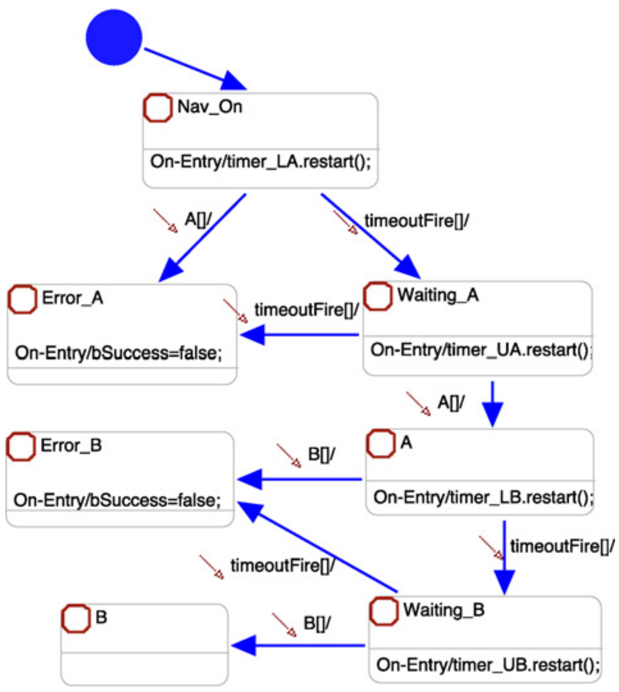


Fig. 5. Statechart assertion for the *Req_Ref_B*.

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 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 [*lA, uA*].
2) *Req_Ref_B: B* must be detected (only) within the time interval [*lB, uB*] 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

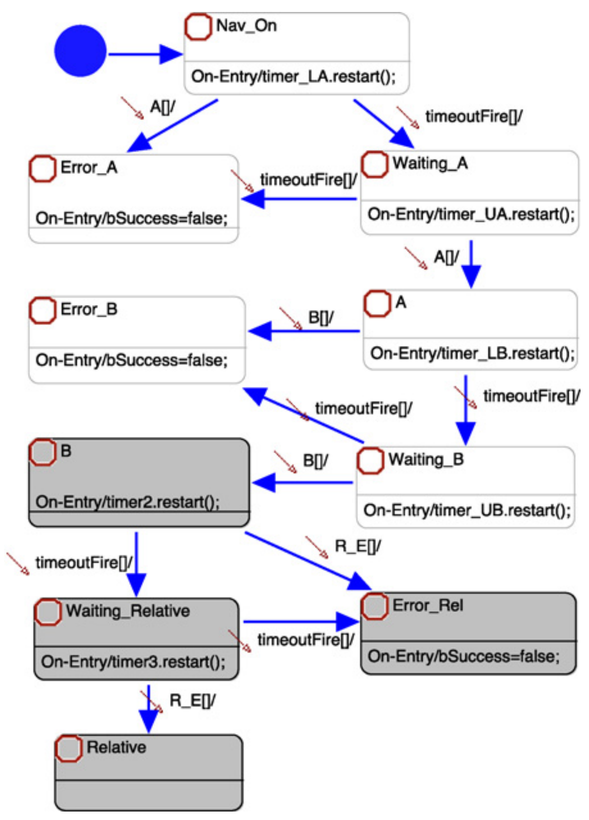


Fig. 6. Statechart assertion for the *Req_Rel_B_i*.

observes the event A while in state *Nav_On*, before event A's interval of occurrence lower limit (lA) has been reached (lA 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_UA*) 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 (lB) has been reached (lB 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 δ_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.

Req_Rel_B_i: whenever B is detected, then B_i must occur between δ_i and $\delta_i + m$ 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 δ_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 δ_i timeout has expired (δ_i is manifested in Fig. 6 by the *timer2*, which is the timeout value for the ith 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) *Req_Ref_C*: C must be detected within the interval [lC, uC] after the detection of A.
2) *Req_Ref_D*: D must be detected within the interval [lD, 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_Ref_A*, for example, could be reused as part of the validation tests for *Req_Ref_B*, *Req_Ref_C*, and *Req_Ref_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_1 (denoted as R_E in Fig. 6) requirement as an example. B_1 is supposed to occur 1000 ms ($\delta_1 = 1000$) after the B occurrence. Fig. 7 illustrates the timeline associated with three JUnit tests, for the following scenarios: 1) a success scenario, where the event R_E occurs 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.

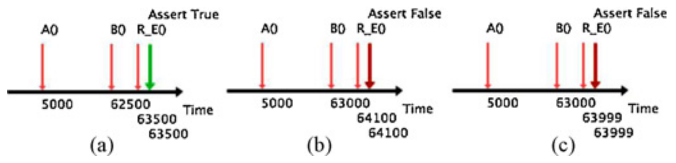


Fig. 7. Three validation tests. (a) Full scenario success test. (b) Full scenario failure where the event R_E occurred too late. (c) Scenario where R_E occurred early.

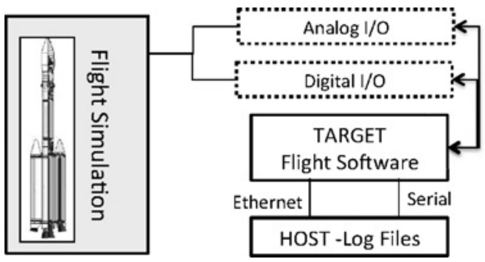


Fig. 8. Lab setup scheme for runtime execution monitoring.

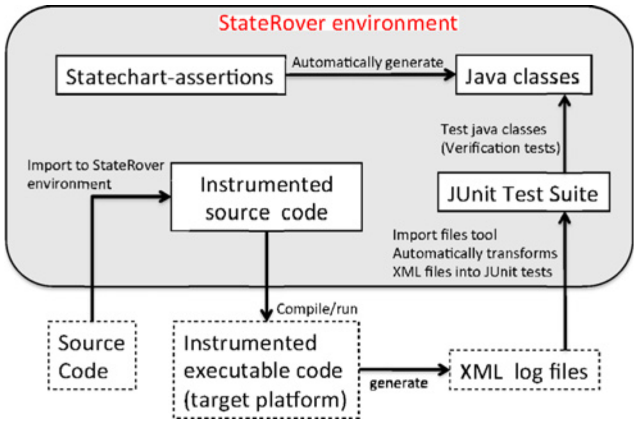


Fig. 9. Code instrumentation and verification testing.

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.

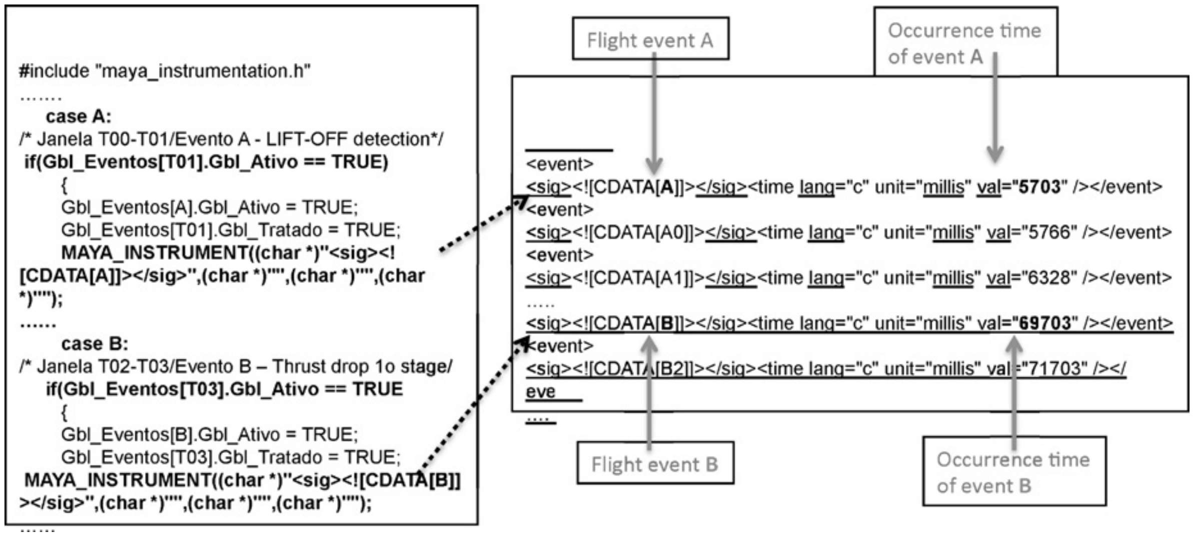
In the verification phase, we observed that approximately 30% of the assertions were violated due to late reference-events detection. The time in the log files was compared to the telemetry time data, which was collected under the same testing conditions, but with no instrumentation in the code. In both 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-



```
#include "maya_instrumentation.h"
.......
    case A:
/* Janela T00-T01/Evento A - LIFT-OFF detection*/
 if(Gbl_Eventos[T01].Gbl_Ativo == TRUE)
      {
      Gbl_Eventos[A].Gbl_Ativo = TRUE;
      Gbl_Eventos[T01].Gbl_Tratado = TRUE;
      MAYA_INSTRUMENT((char *)"<sig><!
[CDATA[A]]></sig>",(char *)"",(char *)"",(char
*)"");
......
     case B:
/* Janela T02-T03/Evento B – Thrust drop 1o stage/
   if(Gbl_Eventos[T03].Gbl_Ativo == TRUE
      {
      Gbl_Eventos[B].Gbl_Ativo = TRUE;
      Gbl_Eventos[T03].Gbl_Tratado = TRUE;
 MAYA_INSTRUMENT((char *)"<sig><![CDATA[B]]
></sig>",(char *)"",(char *)"",(char *)"");
......
```

```
<event>
<sig><![CDATA[A]]></sig><time lang="c" unit="millis" val="5703" /></event>
<event>
<sig><![CDATA[A0]]></sig><time lang="c" unit="millis" val="5766" /></event>
<event>
<sig><![CDATA[A1]]></sig><time lang="c" unit="millis" val="6328" /></event>
.....
<sig><![CDATA[B]]></sig><time lang="c" unit="millis" val="69703" /></event>
<event>
<sig><![CDATA[B2]]></sig><time lang="c" unit="millis" val="71703" /></
eve
....
```

Fig. 10. Instrumented code and the resulting log file.

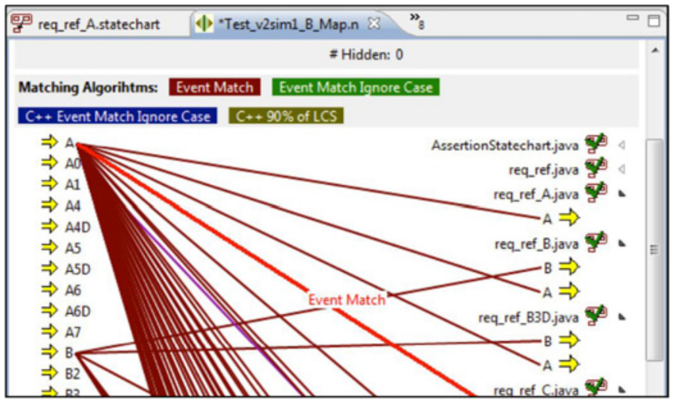


Fig. 11. Namespace mapping for verification testing.

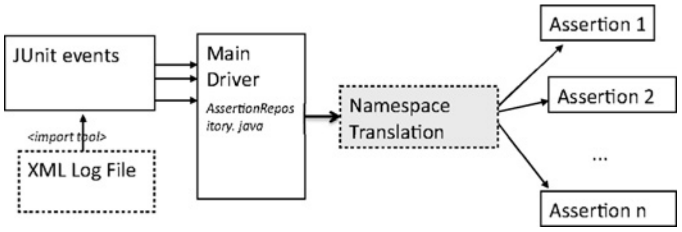


Fig. 12. Verification tests execution from log files.

TABLE I

SUMMARY OF THE VALIDATION AND VERIFICATION TESTS

Validation Tests	Verification Tests
220 (around five tests per assertion)	Four log files (four tests per assertion)
220 JUnit classes—one JUnit class per test	4 JUnit class—one JUnit class per log file
132 success scenarios (around 60% of the scenarios)	31 assertions passed in all tests (around 70% of the assertions)
88 premeditated failure scenarios (around 40% of the scenarios)	13 assertions failed at least in one test (around 30% of the assertions)

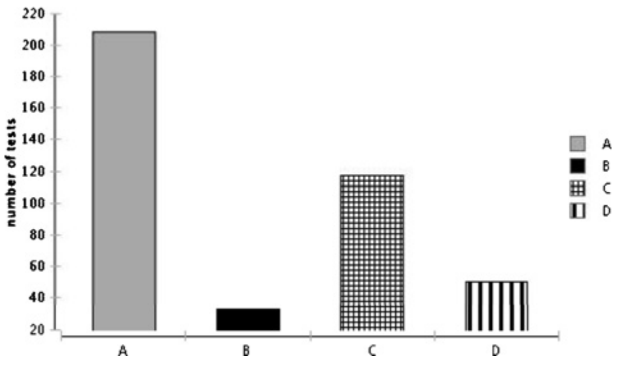


Fig. 13. Number of validation tests per *reference* events (metric S1).

ysis. Unfortunately, such frameworks frequently become prohibitively costly, resulting in large amounts of data that is never analyzed or used.

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.

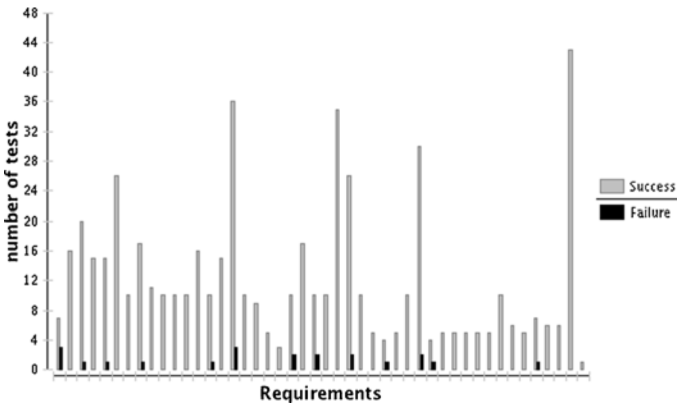


Fig. 14. Validation tests results X requirements (metric S2).

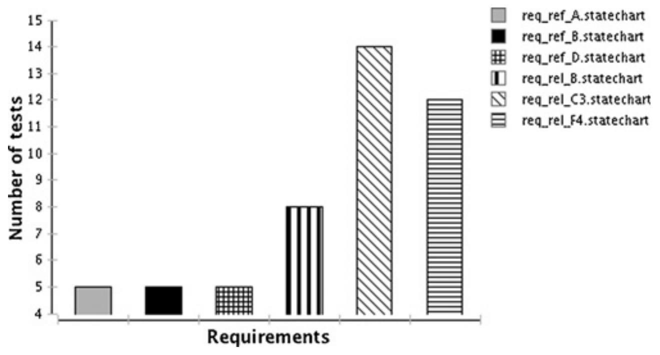


Fig. 15. Number of validation tests per requirement (metric S3).

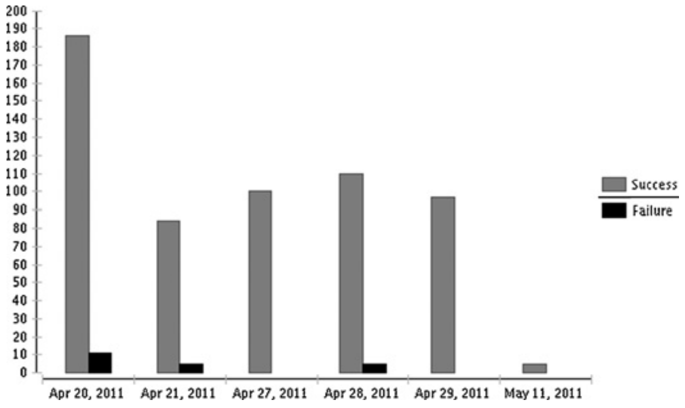


Fig. 16. Validation test results per day (metric P1).

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 SV&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 and reviewed 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

[1] A. Dekhtyar, J. H. Hayes, S. Sundaram, A. Holbrook, and O. Dekhtyar, "Technique integration for requirements assessment," in *Proc. 15th IEEE Int. Requirements Eng. Conf.*, Oct. 2007, pp. 141–150.

[2] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling," *IEEE Trans. Softw. Eng.*, vol. 24, no. 1, pp. 4–14, Jan. 1998.

[3] J. M. Thompson, M. P. E. Heimdahl, and S. Miller, "Specification-based prototyping for embedded systems," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, LNCS 1687. 1999, pp. 163–179.

[4] M. P. E. Heimdahl, "A case for specification validation," in *Verified Software: Theories, Tools, Experiments* (LNCS, vol. 4171), B. Meyer and J. Woodcock, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 392–402.

[5] D. Drusinsky and M. Shing, "Verification of timing properties in rapid system prototyping," in *Proc. 14th IEEE Int. Workshop Rapid Syst. Prototyping*, Jun. 2003, pp. 47–53.

[6] D. Drusinsky, J. B. Michael, and M. Shing, "A framework for computer-aided validation," *Innovations Syst. Softw. Eng.*, vol. 4, no. 2, pp. 161–168, 2008.

[7] P. Berander and P. Jönsson, "A goal question metric based approach for efficient measurement framework definition," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, Sep. 2006, pp. 316–325.

[8] M. C. B. Alves, D. Drusinsky, J. B. Michael, and M. Shing, "Formal validation and verification of space flight software using statechart-assertions and runtime execution monitoring," in *Proc. 6th IEEE Int. Syst. Syst. Conf.*, Jun. 2011, pp. 155–160.

[9] D. Drusinsky, *Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-Based Model Checking*. Burlington, MA: Elsevier, 2006.

[10] D. Drusinsky, J. B. Michael, and M. Shing, "A visual trade-off space for formal verification and validation techniques," *IEEE Syst. J.*, vol. 2, no. 4, pp. 513–519, Dec. 2008.

[11] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Trans. Softw. Eng.*, vol. 20, no. 9, pp. 684–706, Sep. 1994.

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[13] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "Statemate: A working environment for the development of complex reactive systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, pp. 403–414, Apr. 1990.

[14] M. P. E. Heimdahl and N. G. Leveson, "Experiences from specifying the TCAS II requirements using RSML," in *Proc. 17th Digital Avionics Syst. Conf.*, vol. 1. Oct. 1998, pp. 1–8.

[15] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. Mckenzie, *Systems and Software Verification: Model-Checking Techniques and Tools*. Berlin, Germany: Springer-Verlag, 2001.

[16] F. Schneider, S. M. Easterbrook, J R Callahan, and G. J. Holzmann, "Validating requirements for fault tolerant systems using model checking," in *Proc. 3rd Int. Conf. Requirements Eng.*, Apr. 1998, pp. 4–13.

[17] P. R. Glück and G. J. Holzmann, "Using SPIN model checking for flight software verification," in *Proc. IEEE Aerospace Conf.*, Mar. 2002, pp. 105–113.

[18] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White, "Formal analysis of the remote agent before and after flight," presented at the 5th NASA Langley Formal Methods Workshop, Williamsburg, VA, Jun. 2000.

[19] K. Havelund, M. Lowry, and J. Penix, "Formal analysis of a space craft controller using SPIN," *IEEE Trans. Softw. Eng.*, vol. 27, no. 8, pp. 749–765, Aug. 2001.

[20] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann, "Validating requirements for fault tolerant systems using model checking," in *Proc. 3rd Int. Conf. Requirements Eng.*, 1998, pp. 4–13.

[21] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng*, vol. 23, no. 5, pp. 1–17, May 1997.

[22] U. Nitsche, "Propositional linear temporal logic and language homomorphisms," in *Proc. 3rd Int. Symp. Logical Found. Comput. Sci.*, LNCS 813. 1994, pp. 265–277.

[23] W. Thomas, "Automata on infinite objects," in *Handbook of Theoretical Computer Science: Formal Models and Semantics*, vol. B, J. van Leeuwen, Ed. Cambridge, MA: MIT Press, 1990, pp. 133–191.

[24] R. Jeffords and C. L. Heitmeyer, "Automatic generation of state invariants from requirements specifications," in *Proc. 6th Int. Symp. Found. Softw. Eng.*, Nov. 1998, pp. 56–69.

[25] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, pp. 231–261, 1996.

[26] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Trans. Softw. Eng.*, vol. 6, no. 1, pp. 2–13, Jan. 1980.

[27] D. Drusinsky, K. M. Shing, and A. Demir, "Test-time, run-time, and simulation-time temporal assertions in RSP," in *Proc. 16th IEEE Int. Workshop Rapid Syst. Prototyping*, Jun. 2005, pp. 105–110.

[28] D. Drusinsky and G. Watney, "Applying run-time monitoring to the deep-impact fault protection engine," in *Proc. 28th IEEE/NASA Softw. Eng. Workshop*, Dec. 2003, pp. 127–133.

[29] K. Havelund and T. Pressburger, "Model checking Java programs using Java Pathfinder," *Int. J. Softw. Tools Technol. Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[30] K. Havelund and G. Rosu, "An overview of the runtime verification tool Java PathExplorer," *Formal Methods Syst. Design*, vol. 24, no. 2, pp. 189–215, 2004.

[31] D. Drusinsky and M. Shing, "Verifying distributed protocols using MSC-assertions, run-time monitoring, and automatic test generation," in *Proc. 18th IEEE/IFIP Int. Workshop Rapid Syst. Prototyping*, Jun. 2007, pp. 82–88.

[32] A. Madhukar, S. Fischmeister, Y. Hur, J. Kim, and I. Lee, "Generating reliable code from hybrid-systems models," *IEEE Trans. Comput.*, vol. 59, no. 9, pp. 1281–1294, Sep. 2010.

[33] M. C. B. Alves, D. Drusinsky, and M. Shing, "A practical formal approach for requirements validation and verification of dependable systems," in *Proc. 5th Latin-American Symp. Dependable Comput.*, Apr. 2011, pp. 47–51.

[34] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Rep.*, vol. 3, no. 7, pp. 37–50, 1998.

[35] D. Drusinsky, B. Michael, T. Otani, and M. Shing, "Validating UML statechart-based assertions libraries for improved reliability and assurance," in *Proc. 2nd Int. Conf. Secure Syst. Integr. Reliability Improvement*, Jul. 2008, pp. 47–51.

[36] D. Drusinsky, J. B. Michael, and M. Shing, "Rapid runtime system verification using automatic source code instrumentation," in *Proc. 6th IEEE Int. Syst. Syst. Eng. Conf.*, Jun. 2011, pp. 1–6.

[37] T. Fields, *Distributed Game Development: Harnessing Global Talent to Create Winning Games*. Burlington, MA: Focal Press, 2010.

[38] F. Brooks, *The Mythical Man-Month*, anniversary ed. Reading, MA: Addison-Wesley, 1995.

Miriam C. Bergue Alves received the Doctoral degree in applied computer science from the National Institute for Space Research, São José dos Campos, Brazil, in 1999.

She is currently a Government Researcher with the Software Engineering Laboratory, Institute of Aeronautics and Space, São José dos Campos. From 2010 to 2011, she was a Post-Doctoral Researcher with the Department of Computer Science, Naval Postgraduate School, Monterey, CA. She has been developing aerospace software systems since 1995 and currently leads the team responsible for software development of the Brazilian Satellite Launcher Program. Her current research interests include modeling, specification, and formal verification and validation of mission-critical systems.

Doron Drusinsky received the Ph.D. degree in computer science from the Weizmann Institute of Sciences, Rehovot, Israel, in 1988.

He is currently an Associate Professor with the Department of Computer Science, Naval Postgraduate School, Monterey, CA, and is the President of Time-Rover, Cupertino, CA. He was with Sony, Atsugi, Japan, from 1988 to 1993. He has authored Better-State, a UML statecharts design tool that was later acquired by ISI/WindRiver Systems, Alameda, CA, and the Temporal Rover and StateRover verification and validation (V&V) tools that are currently in active use by the NASA IV&V Facility. He has published two books on V&V of mission-critical systems. His current research interests include computer-aided specification and V&V of mission-critical systems.

James Bret Michael (S'87–M'92–SM'97) received the Ph.D. degree in information technology from George Mason University, Fairfax, VA, in 1993.

He is currently a Professor with the Department of Computer Science and the Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA. Prior to that, he was an Assistant Research Engineer with the University of California, Berkeley. His current research interests include formal methods, reliability, safety, and security.

Dr. Michael won the 2010 Engineer of the Year Award from the IEEE Reliability Society for his contributions to the field of trustworthy distributed systems. He is an Associate Editor of the IEEE SYSTEMS JOURNAL.

Man-Tak Shing (M'03–SM'07) received the Ph.D. degree in computer science from the University of California, San Diego.

He is currently an Associate Professor with the Department of Computer Science, Naval Postgraduate School, Monterey, CA. His current research interests include modeling, specification, and formal verification and validation of real-time system behaviors.

Dr. Shing is on program committees of several conferences dedicated to software engineering. He has served as the Program Co-Chair of the IEEE International Conference on System of Systems Engineering and as the Program and General Co-Chair of the IEEE International Symposium on Rapid System Prototyping.