Formal Verification for a Next-Generation Space Shuttle

Stacy D. Nelson[1], Charles Pecheur[2]

[1] Nelson Consulting, NASA Ames Research Center, M/S 269-2, Moffett Field, CA 94035, USA
nelsonconsult@aol.com
[2] RIACS, NASA Ames Research Center, M/S 269-2, Moffett Field, CA 94035, USA
pecheur@ptolemy.arc.nasa.gov

Abstract. This paper discusses the verification and validation (V&V) of advanced software used for integrated vehicle health monitoring (IVHM), in the context of NASA's next-generation space shuttle. We survey the current V&V practice and standards used in selected NASA projects, review applicable formal verification techniques, and discuss their integration into existing development practice and standards. We also describe two verification tools, JMPL2SMV and Livingstone PathFinder, that can be used to thoroughly verify diagnosis applications that use model-based reasoning, such as the Livingstone system.

1 Introduction

NASA is investing in the future of space transportation by investigating automated and integrated technologies for monitoring the health of future space shuttles and their ground support equipment. This application field, known as Integrated Vehicle Health Management (IVHM), is being developed in by the aerospace industry under the auspices of NASA's Space Launch Initiative (SLI) program.

The proposed IVHM system includes advanced software technologies such as model-based diagnosis using NASA's Livingstone system. This holds the promise of automating the diagnosis across a number of subsystem components and possible scenarios that is not tractable for more conventional diagnosis techniques. On the flip side, however, it also carries the burden of verifying that this very complex system will perform as expected in all those situations.

This paper is based on a survey of V&V techniques for IVHM, carried at NASA Ames in support of Northrop Grumman's IVHM project [16,17,18]. Section 2 gives an overview of IVHM for space vehicles, Section 3 surveys software verification and validation practices at NASA, Section 4 discusses applicable formal methods and ways to incorporate them into the software process in accordance with standards, Section 5 presents our tools for V&V of model-based diagnosis, and Section 6 discusses ongoing work on maturing these tools and infusing them in the IVHM design process.

2 Advanced Health Management for Space Vehicles

Advanced health management for space vehicles makes it possible to detect, diagnose, and in some cases, remediate faults and failures without human intervention. This is critical to future space exploration because longer missions into deep space cannot be effectively managed from earth due to the length of time for a telemetry stream to reach earth from the space vehicle. It is also important to NASA's Space Launch Initiative focusing on affordable low earth orbit space vehicle, like the U.S. Space Shuttle, in order to improve crew safety and reduce costs.

NASA's Space Launch Initiative 2nd Generation Reusable Launch Vehicle (2nd Gen RLV) program is investing into future space transportation technologies, towards a flexible, commercially-produced fleet of reusable launch vehicles. The objective of the current Risk Reduction Phase is to enable a mid-decade competition such that critical technology demonstrations for each proposed architecture are adequately integrated, funded, and scheduled.

Integrated Vehicle Health Management, or IVHM, is one of the technology areas supported as part of 2nd Gen RLV. Simply stated, IVHM exists to diagnose/prognose, evaluate and remediate failure modes. The system is composed of a generic (in-flight & maintenance) architecture suitable for building an IVHM system from health management subsystems developed by different vendors [19].IVHM consists of both flight vehicle (FV-IVHM) and ground (GIVHM) components. FV-IVHM is primarily concerned with diagnosing and prognosing failures that have or might occur during the current flight. Any response to or remediation of these failures would occur during the current flight. GIVHM is primarily concerned with diagnosing/prognosing failures that may occur on the ground prior to take off or on a subsequent flight. This includes any pre-existing failure states. Both FV-IVHM and GIVHM contain model-based reasoning software.

Model-Based diagnosis is one of the key technologies currently adopted for next-generation shuttle IVHM. Model-Based Reasoning consists of applying a general-purpose *reasoning engine* to a declarative *model* of the application's artifacts. Specifically, model-based diagnosis uses a description the different components in the system and their interactions, including the failure modes of each component. These models capture all the application-relevant information in an abstract, concise, declarative representation. The diagnosis program itself is re-usable across different diagnosis applications.

Livingstone is a model-based diagnosis system developed at NASA Ames [26]. Livingstone models describe the normal and abnormal functional modes of each component in the system. Livingstone observes the commands issued to the plant and uses the model to predict the plant state. It then compares the predicted state against observations received from the actual sensors. If a discrepancy is found, Livingstone performs a diagnosis by searching for the most likely configuration of component modes that are consistent with the observations.

3 Software V&V at NASA

Software V&V is defined as the process of ensuring that software being developed or changed will satisfy functional and other requirements (verification) and each step in the process of building the software yields the right products (validation). A survey of current practice in Verification & Validation (V&V) of safety-critical software across NASA was conducted to support initial planning and analysis for V&V of the 2nd Generation Re-usable Launch Vehicle IVHM.

Three missions were selected as being representative of current software V&V practices: Checkout & Launch Control System (CLCS); X-37 IVHM Experiment; and Deep Space One (DS1) - Remote Agent (RA) including review of the Formal Verification conducted on RA. The following sections summarize survey results for CLCS, DS1 and the Formal V&V conducted on RA. X-37 is not included because, while the V&V effort was excellent, the experiment was at an early stage and less useful information could be collected.

3.1 Check-out and Launch Control (CLCS) System

The objective of the CLCS Project was to provide a real-time computerized Space Shuttle checkout system used to control and monitor test operations and launch. The CLCS project had comprehensive V&V plans based on NASA standards and contained in an online repository (http://clcs.ksc.nasa.gov/docs/test-specs.html). CLCS was canceled in August 2002; however, review of this project revealed two important lessons:

1. using the spiral or evolutionary strategy described in IEEE 12207.2 Annex I is more cost effective than the waterfall strategy; and
2. it is important to evaluate IV&V budget requirements early in the project. For example, a manned mission or program costing more than $100M requires review by the NASA Independent Verification and Validation (IV&V) team.

3.2 Remote Agent

The objective of the Deep Space One (DS1) mission was to test 12 advanced technologies in deep space so these technologies could be used to reduce the cost and risk of future missions. One of the 12 technologies on DS1 was Remote Agent, a software product designed to operate a spacecraft with minimal human assistance. The successful demonstration of Remote Agent on DS1 lead its team to become co-winners of the NASA 1999 Software of the Year Award.

The V&V of DS1 used a number of testbeds, as detailed in Table 1. V&V was conducted via carefully planned operations scenarios and tests were distributed among low, medium and high-fidelity testbeds, which improved project team agility and reduced testing costs. Operations scenarios were used effectively to test nominal and off-nominal events. Operational Readiness Tests identified procedural problems during "dress rehearsal" so they could be corrected before the actual mission.

Table 1. Deep Space One – Remote Agent Testbeds

Testbed	Fidelity	CPU	Hardware	Availability	Speed	Readiness Dates
Spacecraft	Highest	Rad6000	Flight	1 for DS1	1:1	05/99
DS1 Testbed	High	Rad6000	Flight spares + DS1 sims	1 for DS1	1:1	04/99
Hotbench	High	Rad6000	Flight spares + DS1 sims	1 for DS1	1:1	03/99
Papabed	Medium	Rad6000	Flight spares + DS1 sims	1 for DS1	1:1	11/98
Radbed	Low	Rad6000	RAX Simulators	1 for RAX	1:1	04/98
Babybed	Lowest	PowerPC	RAX Simulators	2 for RAX	7:1	02/98
Unix	Lowest	SPARC UNIX	RAX Simulators	Unlimited	35:1	08/97

Throughout initial stages of the project, the goal of testing was to discover bugs so they could be repaired. As it grew closer to take off; however, the discovery of a bug did not automatically imply it would be fixed. Instead, a Change Control Board (CCB) composed of senior RA project members reviewed the details of each bug and proposed fix to assess the risk of repair. The CCB became increasingly conservative near mission launch date preferring to work around bugs rather than risk inadvertently breaking more code during while repairing a bug.

3.3 Formal V&V of Remote Agent's Executive

Incidentally, the executive part of the Remote Agent has been the target of a very illustrative formal verification experiment [6]. The results came in two phases.

In 1997, a team from the ASE group at Ames used the Spin model checker [9] to verify the core services of Remote Agent's Executive (EXEC) and found five concurrency bugs. Four of these bugs were deemed important by the executive software development team, which considered that these errors would not have been found through traditional testing. Once a tractable Spin model was obtained, it took less than a week to carry out the verification activities. However, it took about 1.5 work-months to manually construct a model that could be run by Spin in a reasonable length of time, starting from the Lisp code of the executive.

In May 1999, as the Remote Agent was run in space Deep Space One, an anomaly was discovered in the EXEC. Shortly after, the ASE team took the challenge of performing a "clean room" experiment to determine whether the bug could have been found using verification and within a short turnaround time. Over the following weekend, they successfully revealed and demonstrated the bug using the group's Java PathFinder tool [7]. As it turns out, the bug was a deadlock due to improper use of synchronization events, and was isomorphic to one of the five bugs detected in another part of EXEC with Spin two years before. This verification effort clearly demonstrated that advanced V&V techniques can catch the kind of concurrency bugs that typically pass through heavy test screens and compromise a mission.

3.4 Software Process Standards

In order for advanced software to fly on spacecraft or aircraft, it must be approved by relevant authorities—NASA for space, the Federal Aviation Authority (FAA) for civil

aviation. This approval generally involves conformance with some established software process and V&V standards. The following NASA Standards are relied upon for guidance:

NASA NPG 2820 *"Software Guidelines and Requirements"* [14]. This document references IEEE/EIA Standards 12207.0, 12207.1 and 12207.2 [10,11,12], which are themselves based on ISO/IEC 12207. This series is in widespread use in the industry.

NASA NPG 8730 *"Software Independent Verification and Validation (IV&V) Management"* [15] discusses the requirements for independent verification and validation. In a nutshell, a manned mission and any mission or program costing more than $100M will require IV&V.

In addition to the NASA standards, RTCA DO-178B, "Software Considerations in Airborne Systems and Equipment Certification" [24] contains guidance for determining that software aspects of airborne systems and equipment comply with airworthiness certification requirements.

In summary, to comply with the above described standards, each project must have a well-defined process with discrete phases and thoroughly documented work products for each phase.

4 Formal Methods for IVHM V&V

IVHM considerably increases the complexity of the software V&V task, in two different ways:

The software components are more complex and may involve non-conventional programming paradigms. For example, model-based reasoning uses a logic reasoning component issued from artificial intelligence research; furthermore, a lot of the application-specific complexity lies in the model rather than in the diagnosis program itself.

The state space to be covered is incomparably larger, as the IVHM system aims at properly handling, at least partly autonomously, a large set of possible faults under a broad range of unforeseen circumstances. This contrasts with the current largely human-based approach to diagnosis, with a number of mission controllers monitoring the vehicle's telemetry.

These two aspects pose a serious challenge to conventional V&V approaches based on testing. To achieve the required level of confidence in IVHM software, more advanced methods need to be applied. This section surveys the main formal analysis techniques issued from the research community and discusses their applicability to IVHM systems. It summarizes the results from the second report of the survey [17].

4.1 The Verification Spectrum

The term "Formal Methods" refers to various rigorous analysis and verification techniques based on strong mathematical and logic foundations. In principle, formal veri-

fication will guarantee that a system meets the specifications being verified, whereas informal techniques can only detect errors or increase confidence. In practice though, the limit is blurred by the abstractions, restrictions and simplifications needed to express the system into a formal representation amenable to formal analysis. One should rather think of a spectrum of techniques, with various degrees of formality.

Below is an overview the broad categories of verification methods. The methods are ordered in increasing level of formality. Generally, more formal methods provide greater assurance, at the cost of greater required expertise—from testing methods used throughout the software industry, up to theorem proving techniques mastered only by a few experts. Nevertheless, there can be wide variations between tools and applications within a category, and many approaches blend aspects of different categories.

Testing consists in executing the system through a pre-established collection of sequences of inputs (*test cases*), while checking that the outputs and the system state meet the specification. This is the most common, and often the unique, mechanized verification technique used in most projects. It defines the baseline against which other techniques should be compared. The test cases are commonly developed and tuned manually by application experts, a cumbersome, error-prone and very time-consuming task. A *test harness* also has to be developed, to simulate the operational environment of the tested system. As a result, testing is often the most costly part of the whole development phase, especially in safety-critical applications such as space transportation.

Runtime Monitoring consists in monitoring a system while it is executing, or scrutinizing the artifacts (event logs, etc) obtained from that execution. It can be used to control complex specifications that involve several successive events. In some cases, it can even flag suspicious code even if no error actually occurs. Runtime monitoring typically requires little computing resources and therefore scales up well to very large systems. On the other hand, it will only observe a limited number of executions and thus gives only uncertain results. In the case of error predictions, it can also give false negatives, i.e. flag potential errors that cannot actually occur. Applications of runtime monitoring at NASA include the analysis of generated plans using database queries at Jet Propulsion Labs [5] and NASA Ames' JPaX tool for monitoring Java programs [8].

Static Analysis consists in exploring the structure of the source code of a program to extract information or verify properties, such as absence of array bound violations or non-initialized pointer accesses [20]. In principle, static analysis can be applied to source code early in the development and is totally automatic. There is, however, a trade-off between the cost and the precision of the analysis, as the most precise algorithms have a prohibitive complexity. More efficient algorithms make approximations that can result in a large number of *false positives*, i.e. spurious error or warning messages. NASA has performed several experiments using PolySpace, a static analyzer for C programs [23].

Model Checking consists in verifying that a system satisfies a property by exhaustively exploring all its reachable states [3,1]. This requires that this state space be finite—and tractable: model checking is limited by the *state space explosion* problem, where the number of states can grow exponentially with the size of the

system. Tractability is generally achieved by abstracting away from irrelevant details of the system. When the state space is still to big or even infinite, model checking can still be applied: it will not be able to prove that a property is satisfied, but can still be a very powerful error-finding tool. Model checking in itself is automatic, but the modeling phase can be a very arduous and error-prone effort. Model checkers often impose their own modeling language, though more and more tools now apply directly to common design and programming languages (UML, Java), either natively or through translation. **Symbolic model checking** is an advanced form of model checking that considers whole sets of states at each step, implicitly and efficiently encoded into data structures called Binary Decision Diagrams (BDDs). Symbolic model checking can address much larger systems than explicit state model checkers, though the complexity of the BDDs can outweigh the benefits of symbolic computations. One of the major symbolic model checkers is SMV from Carnegie-Mellon University (CMU) [2], which has been used in this project, as detailed in Section 5.1.

Theorem Proving consists in building a computer-assisted logic proof that the system satisfies the specifications, where both are suitably represented in the mathematical framework of the proof system being used. When applicable, theorem proving provides the "Holy Grail" of V&V, as it has the potential to provide a mathematically correct, computer-verified proof of compliance. However, the proof systems needed for this kind of endeavor demand a lot of user guidance; proving the final specification will typically require providing and proving a number of intermediate properties (e.g. loop invariants). Although modern proof systems provide ever more elaborate *tactics* that automatically chain more elementary proof steps, this kind of verification still requires a lot of expertise and work, and is most often performed on small-scale designs by specialized researchers.

No matter how effective more formal methods can be, *testing* remains an essential element of the V&V process. In addition, our findings recommended that *model checking*, *static analysis* and *runtime verification* be added to the set of methods applicable to V&V of IVHM. *Theorem proving* was not recommended, due to the excessive effort and expertise it requires, and because it was not considered appropriate for the large, complex IVHM systems under consideration. The next section discusses where these different techniques fit into the software development process.

4.2 Formal Methods in the Software Process

In order to ensure that formal verification techniques meet the V&V standards, the following guidance is provided for integrating formal methods into the Software Life Cycle.

Effective software development requires a well-defined process with discrete Software Life Cycle phases including documented work products (called deliverables) for each phase; analysis procedures established to ensure correctness of deliverables; and scheduled reviews of major product releases. These items have to be defined in order for formal methods to be integrated in the software V&V process.

Formal methods can be applied to any or all phases of the Software Life Cycle. They may be used to enhance rather than replace traditional testing; although tradi-

tional testing efforts may be significantly reduced when formal methods are used effectively. Different types of formal methods can be used at different stages in the life cycle: model checking may be applied at the different levels of the design phase, down to program code; static analysis is typically geared towards executable code; runtime monitoring is applicable at different stages in the integration and testing phase, every time real code is being executed.

Table 2. Recommendations for Formal Methods in the V&V Process (SW = Software, SRA = System Requirements Analysis, SWRA = Software Requirements Analysis)

Formal Methods	Applicable SW Life Cycle Phase	Formal Verification Activities
Any Formal Methods Technique	SRA/SWRA	Perform a new development activity called "formalization" to create a new work product called a "formal specification". Enhance traceability tools and techniques to track new products such as formal specifications and proofs, and their relationships to existing products
Model Checking (Theorem Proving)	SRA/SWRA	Perform a new analysis activity called "proving assertions" to enhance the correctness of the formal specification and to understand the implications of the design captured in the requirements and specification. Perform an Official Review of the formal specification to check the coverage, correctness, and comprehensibility of the formal specification.
Model Checking	SWRA and SW & Model Detailed Design	Perform a new analysis activity called "modeling", producing a new work product called a "formal model". Perform a new activity called "formal analysis" where model checking is applied to the formal models. Model checking enables all execution traces to be verified. This improves the accuracy and reliability of the product and allows early error detection and correction. It may also reduce the amount of traditional testing required.
Model Checking	SWRA and SW & Model Detailed Design	Perform an Official Review of the model to check for correctness
Static Analysis	SW & Model Detailed Design and SW Coding	Use Static Analysis tools in addition to a compiler during code development. This can reduce the amount of traditional unit testing required while increasing the accuracy of the program.
Model Checking	SW Coding and SW & Model Unit Testing	If available for the programming language and platform used, use model checkers in addition to standard debugging and test control tools. This can greatly improve the odds of detecting some errors, such as race conditions in concurrent programs.
Runtime Monitoring	SW Coding, SW & Model Unit Testing, SW Qualification Testing, System Qualification Testing	Use Runtime Monitoring during simulation testing at each phase where program code gets executed. This can provide more information about potential errors.

Planning for formal methods includes activities at each level in the life cycle. At the beginning of the program, staffing requirements for formal methods and enhanced project guidelines must be considered: The software development team must include at least one team member knowledgeable in formal methods. Formal V&V guidelines, standards, and conventions should be developed early and followed carefully.

The V&V team must plan how and when to use formal methods; therefore, these new planning steps are recommended: 1) determine which software or software components will benefit from use of formal methods, 2) select the appropriate type of formal methods, 3) choose the formal methods toolkit and 4) enhance life cycle activities for activities associated with formal methods. Table **2** summarizes recommendations for enhancing the life cycle.

Metrics are important to track effectiveness of formal verification activities. Potentially useful metrics for formal verification include:

Number of issues found in the original requirements (i.e., the requirements in their English description form, before being formalized), along with a subjective ranking of importance (e.g., major, minor)
Amount of time spent in reviewing and in inspection meetings, along with a number and type of issues found during this activity
Number of issues found after requirements analysis, along with a description of why the issue was not found (e.g., inadequate analysis, outside the scope of the analysis, etc.)

Metrics specific to model checking include: amount of time spent in model development (both human and CPU time) and amount of coverage.

5 V&V of Model-Based Diagnosis

A model-based diagnosis system such as Livingstone involves the interaction between various components: the reasoning engine that performs the diagnosis, the model that provides application-specific knowledge to it, the physical system being diagnosed, the executive that drives it and acts upon diagnosis results. There are multiple facets of such a system that need to be verified, and different verification techniques that can be applied to that objective. In this section, based on our third report [18], we present two tools developed at NASA Ames for verifying Livingstone-based applications, and discuss their applicability as part of the larger V&V process.

5.1 Symbolic Model Checking of Livingstone Models

By their abstract, declarative nature, the models used for diagnosis lend themselves well to formal analysis. In particular, Livingstone models are semantically very close to those used by symbolic model checkers such as SMV. The languages are different, though, so a translation is necessary.

In many previous experiences in model checking of software, this translation had to be done by hand, and was by far the most complex and time-consuming part, that has hindered adoption of formal verification by the software industry. Instead, the goal is for Livingstone application developers to use model checking to assist them in designing and correcting their models, as part of their usual development environment. To achieve that, we have developed, in collaboration with Reid Simmons at CMU, a translator to automate the conversion between Livingstone and SMV [22]. The translator supports three kinds of translation, as shown in Figure **1**:

The Livingstone *model* is translated into an SMV model amenable to model checking.
The *specifications* to be verified against this model are expressed in terms of the Livingstone model and similarly translated.
Finally, the *diagnostic traces* produced by SMV are converted back in terms of the Livingstone model.[1]

The translation of Livingstone models to SMV is facilitated by the strong similarities between the underlying semantic frameworks of Livingstone and SMV: both boil down to a synchronous transition system, defined through propositional logic constraints on states and transitions. Based on this, the translation is mostly a straightforward mapping from JMPL to SMV language elements. The specifications to be verified with SMV are provided in a separate file, expressed in a syntax that extends the existing JMPL syntax for logic expressions. They are translated into the CTL temporal logic used by SMV and appended to the SMV model file. CTL is very expressive but requires a lot of caution and expertise to be used correctly. To alleviate this problem, the translator also supports a number of pre-defined templates and auxiliary operators corresponding to Livingstone-relevant properties and features, such as consistency of the model or the number of failed components. Finally, any error traces reported by SMV are translated back to their Livingstone counterpart—this recent addition is further discussed in section 6.

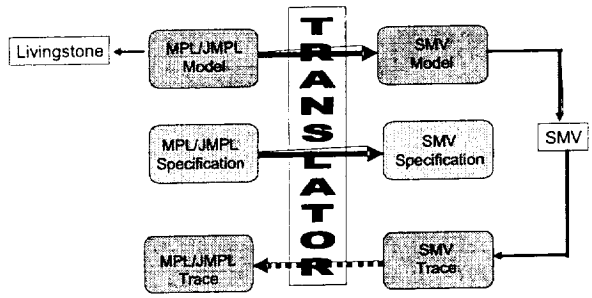

Fig. 1. Translation from Livingstone to SMV.[2]

The translator has been successfully applied to several Livingstone models, such as the Deep-Space One spacecraft, the Xavier mobile robot from CMU and the In-Situ Propellant Production system (ISPP) developed at NASA Kennedy Space Center for Mars missions. The ISPP experience was the most extensive; it did produce useful feedback to the Livingstone model developers, but also experimented with the potentials and challenges of putting such a tool in the hands of application practitioners.

[1] The reverse translation of traces, shown as a dotted arrow, was not available when the survey was made but has since then been implemented.

[2] The original translator applied to an earlier, Lisp-style syntax for Livingstone models (Model Programming Language, or MPL). The translator has later been upgraded to the current Java-like syntax (called JMPL).

Models of up to 10^{55} states could still be processed in a matter of minutes with an enhanced version of SMV. Experience shows that Livingstone models tend to feature a huge state space but little depth, for which the symbolic processing of SMV is very appropriate.

5.2 Extended Simulation with Livingstone PathFinder

Although model-based verification using SMV allows a thorough analysis of Livingstone models, it does not check whether the actual diagnosis engine performs that diagnosis as required. In a complementary approach, we have developed a tool called **Livingstone PathFinder (LPF)** that automates the execution of Livingstone, coupled to a simulated environment, across a large range of scenarios. The system under analysis consists of three parts, as illustrated on figure 2:

The Livingstone *engine* performing the diagnosis, using the user-provided Livingstone model.
A *simulator* for the device on which diagnosis is performed.
A *driver* that generates the commands and faults according to a script provided by the user.

Currently, a second Livingstone engine is used for the simulator. The tool architecture is modular, however, and the different components are accessed through generic programming interfaces (APIs) so that their content can be easily changed.

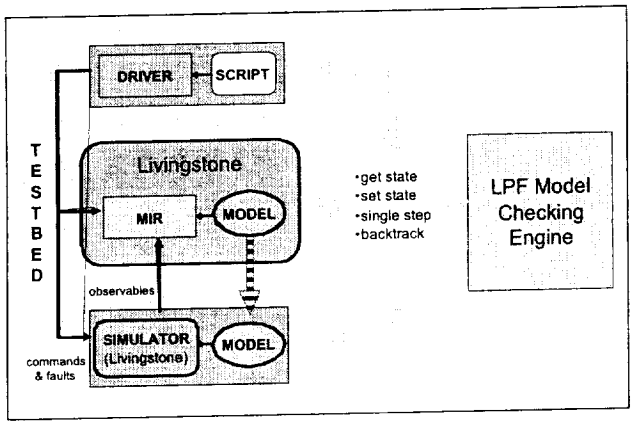

Fig. 2. Architecture of the Livingstone PathFinder tool, showing the diagnosis engine (MIR), the simulator and the driver.

Both Livingstone and its environment are instrumented to allow a closer and more efficient control of the execution, using backtracking to explore alternate paths and observing states to prune redundant executions. This amounts to applying the same state space search as used for model checking. Each forward step consists of a whole diagnosis cycle, where the next event is produced by the driver, applied to the simu-

lator and observed by the diagnosis system. When the current execution terminates (at the end of the scenario or because no consistent next state could be found), LPF backtracks step by step, looking for further alternate executions in previous states. Typically, these alternate paths will explore different failure scenarios. The exploration proceeds along the next alternate path, in a depth-first manner, until the entire tree of possible executions has been covered. [3]

5.3 Correctness and Reliability Criteria

The two tools presented in the previous section only partially address the verification needs of a Livingstone-based application. To put this in perspective, we introduce the following classification of correctness and reliability criteria for model-based diagnosis:

Model Correctness: Is the model a valid abstraction of the actual physical plant? In particular, the model should also be internally well-formed, that is, fulfill generic sanity criteria, such as consistency and completeness.

Diagnosis Correctness: Does the diagnosis engine perform correctly? Note that this verification needs to be addressed once, typically by the engine developers. Once the engine has been verified, it can be viewed as a stable, trusted part, much in the same way as programmers view their programming language compiler.

Diagnosability: Is it possible to perform the required diagnosis? More precisely, is it always possible to correctly detect and diagnose faults or other conditions as specified in the requirements, assuming a perfect model and a perfect engine? Deficiencies against this criterion tend to be design issues rather than problems in the diagnosis system. For example, the system may require additional sensors if a fault can not be adequately detected and isolated.

Assuming model correctness, diagnosis correctness checks that all that can be diagnosed is correctly diagnosed, whereas diagnosability checks that all that needs to be diagnosed can be diagnosed. In principle, if we can fulfill all three conditions, then we can guarantee that the desired diagnosis will be achieved. In practice however, these criteria are often deliberately weakened for technical reasons and efficiency purposes.

To these three architectural criteria, we should add a fourth:

Integration Correctness: Does the diagnosis system correctly interface with its environment? That is, do the different pieces (engine, model, physical system, executive, etc.) properly interact to achieve the required functionality?

Model checking of diagnosis models can definitely address model correctness issues. The translator directly provides specification templates for generic sanity criteria. It can also be used to verify that the model satisfies documented properties of the system, provided those properties can be expressed at the level of abstraction of the model. For example, flow conservation properties have been verified this way. More

[3] The architecture supports other automated or interactive simulation algorithms. Indeed, an implementation of guided search has just been completed.

thorough verification may require case-specific technology, e.g. for comparing models of different nature. Livingstone PathFinder could be modified to perform a comparative simulation for that purpose. We have recently successfully experimented with using symbolic model checking to verify diagnosability, using a duplicated version of the Livingstone model [21]. Livingstone PathFinder can also detect diagnosability problems. Since it performs a focused verification based on running the real engine on real test cases, it provides better fidelity but less coverage. It also provides assurance into diagnosis correctness, though within the same focused scenario limits. Finally, LPF gives a limited assurance in integration correctness, by running the real engine into a simulated environment. This part could be strengthened by plugging higher-fidelity simulators in LPF, and encompassing more components within the LPF-controlled simulation.

6 Maturing V&V Technology

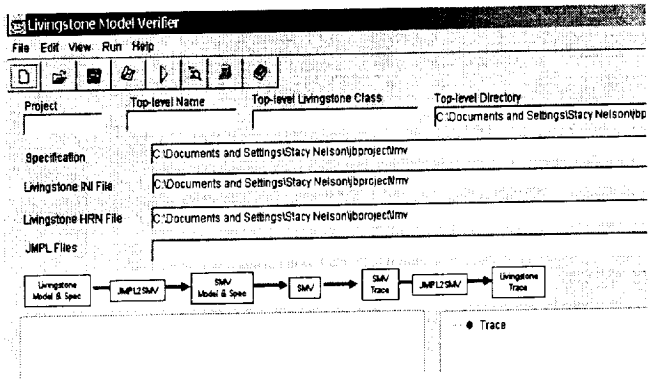

Fig. 3. The Livingstone Model Verifier graphical interface (detail).

As a follow-up on the survey and recommendations presented in the previous sections, work is currently underway to extend formal verification of Livingstone models by enhancing the functionality of JMPL2SMV and Livingstone PathFinder (LPF) and making them easy for a typical engineer to use, without the high cost of hiring formal methods specialists to run them. Enhancements include the following list:

The SMV Trace Translation tool completes the automated translation provided by JMPL2SMV between Livingstone models and SMV models by translating traces produced by SMV back in terms of Livingstone model elements.

A number of new specification patterns have been added to the JMPL2SMV translator, to further simplify the task of specifying the properties to be verified by the model checker.

A graphical front-end interface has been developed to simplify the use of JMPL2SMV, automate the interactions between SMV and the different translators, and provide better visualization capabilities (Figure 3). The resulting tool, called

Livingstone Model Verifier (LMV), appears as an integrated model checker for Livingstone models. A similar interface has also been developed for the Livingstone PathFinder tool.

7 Conclusions and Perspectives

What we have presented here is the result of a six-month, one-person investigation and was therefore necessarily focused on selected applications and tools. Nevertheless, our findings have confirmed that the advanced diagnosis techniques that are being considered for future space transportation vehicles also require advances in verification and validation techniques to guarantee safe and reliable operation. To be applicable, those techniques also need to be easy enough to be used by practitioners and be integrated into existing development frameworks, practices and standards.

We discussed how rigorous verification techniques, coming from research in formal methods, not only improve safety by improving confidence in the system, but can also be implemented and documented in accordance with strict software development and certification standards. We presented our own contributions to the field, JMPL2SMV and Livingstone PathFinder, two verification tools for the Livingstone model-based diagnosis system. These tools make it possible to verify model accuracy earlier in the software development process; therefore, reducing costs and improving system reliability.

These results were received very favorably by our industrial partners in the Space Launch Initiative program. We are now working towards maturing our tools and infusing them into a real IVHM development environment, to demonstrate and evaluate the impact formal methods can bring to the V&V of advanced, safety-critical software architectures.

Bibliography

[1] Beatrice Bérard, Michel Bidoit, Alain Finkel, Francois Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen with Pierre McKenzie. Systems and Software Verification Model-Checking Techniques and Tools. Springer, 1998

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142-170, June 1992.

[3] Edmund M. Clarke, Jr., Orna Grumberg, Doron A. Peled. Model Checking. The MIT Press, 2000.

[4] Ken Costello. Private communication. NASA IV&V Facility, October 13, 2001.

[5] Martin S. Feather. Rapid Application of Lightweight Formal Methods for Consistency Analyses. IEEE Transactions on Software Engineering, Vol. 24, No. 11, November 1998, pp. 949-959.

[6] Klaus Havelund, Mike Lowry, SeungJoon Park, Charles Pecheur, John Penix, Willem Visser, Jon L. White. Formal Analysis of the Remote Agent Before and After Flight. Proceedings of 5th NASA Langley Formal Methods Workshop, Williamsburg, Virginia, 13-15 June 2000.

[7] K. Havelund, T. Pressburger. Model Checking Java Programs Using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT) 2(4), April 2000.

[8] Klaus Havelund, Grigore Rosu. Monitoring Java Programs with Java PathFinder. First Workshop on Runtime Verification (RV'01), Paris, France, 23 July 2001. Electronic Notes in Theoretical Computer Science, Volume 55, Number 2, 2001

[9] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[10] IEEE/EIA. Industry Implementation of International Standard ISO/IEC: ISO/IEC12207 Standard for Information Technology—Software life cycle processes. IEEE/EIA 12207.0-1996.

[11] IEEE/EIA. Industry Implementation of International Standard ISO/IEC: ISO/IEC12207 Standard for Information Technology—Software life cycle processes—Life cycle data. IEEE/EIA 12207.1-1997.

[12] IEEE/EIA. Industry Implementation of International Standard ISO/IEC: ISO/IEC12207 Standard for Information Technology—Software life cycle processes—Implementation Considerations. IEEE/EIA 12207.2-1997.

[13] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. Remote Agent: To boldly go where no AI system has gone before. Artificial Intelligence, 103(1-2):5--48, August 1998.

[14] NASA. NASA Software Guidelines and Requirements. NASA NPG 2820.DRAFT, 3/19/01.

[15] NASA. Software Independent Verification and Validation (IV&V) Management. NASA NPG 8730.DRAFT 2, 30 Nov 2001.

[16] Stacy Nelson, Charles Pecheur. NASA processes/methods applicable to IVHM V&V. Project report, NASA/CR-2002-211401, April 2002.

[17] Stacy Nelson, Charles Pecheur. Methods for V&V of IVHM intelligent systems. Project report, NASA/CR-2002-211402, April 2002.

[18] Stacy Nelson, Charles Pecheur. Diagnostic Model V&V Plan/Methods for DME. Project report, NASA/CR-2002-211403, April 2002.

[19] Northrop Grumman, NASA, DSI. *2nd Generation RLV Risk Reduction Program: TA-5 (IVHM) Project Notebook*. Edited by: Stephen A. Brown. Northrop Grumman, El Segundo, CA, 07/20/01.

[20] F. Nielson, H. R. Nielson, C. Hankin. Principles of Program Analysis. Springer, 1999.

[21] Charles Pecheur, Alessandro Cimatti. Formal Verification of Diagnosability via Symbolic Model Checking. Workshop on Model Checking and Artificial Intelligence (MoChArt-2002), Lyon, France, July 22/23, 2002.

[22] Charles Pecheur and Reid Simmons. From Livingstone to SMV: Formal verification for autonomous spacecrafts. In Proceedings of First Goddard Workshop on Formal Approaches to Agent-Based Systems, April 2000. Lecture Notes in Computer Science 1871, Springer Verlag.

[23] PolySpace Technologies. C Verifier. http://www.polyspace.com

[24] RTCA. Software Considerations in Airborne Systems and Equipment Certification. RTCA (Requirements and Technical Concepts for Aviation) /DO-178B, December 1, 1992.

[25] John Rushby. Assurance for Dependable Systems (Disappearing Formal Methods). Presentation at Safecomp, Budapest, September 2001, TU Vienna, March 2001 and NSA March 2001.

[26] B. C. Williams and P. P. Nayak. A model-based approach to reactive self-configuring systems. In Proceedings of AAAI-96, 1996.