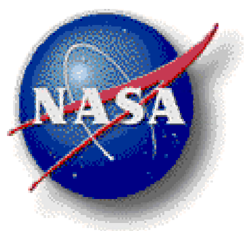
Reduced Public Version

NEW V&V TOOLS FOR DIAGNOSTIC MODELING ENVIRONMENT (DME)

FOR

Northrop Grumman Corp.



TASK NO: 10 TA-5.3.3 (WBS 1.4.4.5.3)

PREPARED FOR:
Northrop Grumman Corp

PREPARED BY:
NASA ARC

Dated: January 25, 2001

Contributors: Stacy Nelson, CSC
Charles Pecheur, RIACS

Changes to this Reduced Public Version:
The purpose of section 5 (Advanced V&V Applicable to DME) is to explain how Advanced V&V tools like MPL2SMV are beneficial to the DME system planned for the 2nd Generation RLV. Because this section contains information described in documents falling under the purview of the U.S. Munitions List (USML) as defined in the International Traffic in Arms Regulation (ITAR), 22 CFR 120-130, it is export controlled and not available on this website.

APPROVALS

Approvals:		
Task Lead:	Charles Pecheur	1/25/02 [signature]
Project Manager:	Marshal Merriam	1/28/02 Marshal L. Merriam
Area Manager:	Michael Lowry	[signature] 1/28/02

Original held by Michael Lowry, Ames Research Center

RECORD OF REVISIONS

REVISION	DATE	SECTIONS INVOLVED	COMMENTS
Initial Delivery	10/26/01	Sections 1, 2, 3, 6, 7, 8	This is a draft only and not intended to be the final deliverable.
Initial Delivery	11/09/01	Section 4	This is a draft only and not intended to be the final deliverable.
Initial Delivery	11/30/01	Section 5, 7 and 8	This is a draft only and not intended to be the final deliverable.
Revisions	12/14/01	All Sections	This is a draft only and not intended to be the final deliverable
Final Deliverable	1/21/02	Sections 3, 4, 5, 8, 9	Revisions to incorporate comments from review. This is the final deliverable.
Reduced Public Version	2/1/02	Section 5	Section 5 was removed because it contains information described in documents falling under the purview of the U.S. Munitions List (USML) as defined in the International Traffic in Arms Regulation (ITAR), 22 CFR 120-130, it is export controlled and not available to the public.

TABLE OF CONTENTS

1. DOCUMENT CONVENTIONS 5
2. EXECUTIVE SUMMARY 6
2.1. CORRECTNESS AND RELIABILITY CRITERIA 7
2.2. TOOLS FOR V&V OF MODEL BASED REASONING 7
2.3. ADVANCED V&V APPLICABLE TO DME 8
3. CORRECTNESS AND RELIABILITY CRITERIA 9
4. TOOLS FOR V&V OF MODEL BASED REASONING 12
4.1. MPL2SMV 12
4.1.1. Introduction to MPL2SMV 13
4.1.2. Overview of Livingstone[1] 14
4.1.3. Description of SMV[1] 15
4.1.4. Verification of Livingstone Models 17
4.1.5. Model Translation[1 & 2] 18
4.1.6. Specifications Translation[1] 19
4.1.7. Traces Translation[1] 19
4.1.8. Applications 19
4.1.9. Lessons Learned Using MPL2SMV[1] 20
4.2. JMPL2SMV 21
4.3. Livingstone PathFinder (LPF) 21
5. ADVANCED V&V APPLICABLE TO DME 24
6. ACRONYMS 25
7. GLOSSARY 27
8. Appendix A: Model-based Verification of Diagnostic Systems 30
8.1. Verification to Design Requirements 30
8.2. Problem Statement in Diagnostic Space {*D*} 30
8.3. Verification of the Diagnostic System Requirements 31
8.4. Fundamental (Black Box) Requirements 31
8.5. Derived (Implementation Specific) Requirements 33
8.6. Deficient Algorithm/Model Forms 34
8.7. Model Verification and Validation: Functional (System Hardware) Models 34
8.8. Conclusions 35
9. REFERENCES 36

1. DOCUMENT CONVENTIONS

The following conventions are used throughout this document:

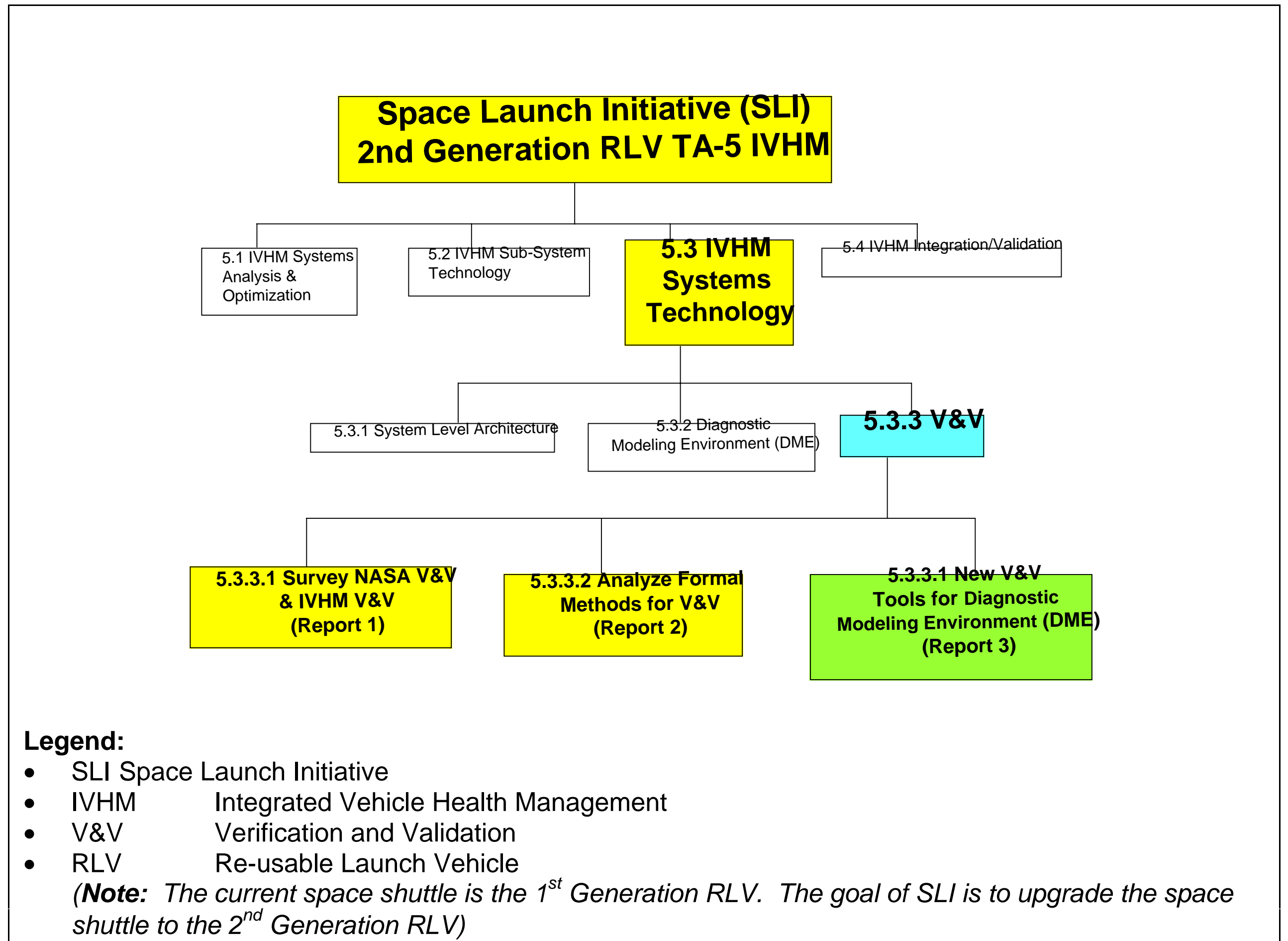
- The term "formal testing" has two meanings. Traditionally, "formal testing" has been used to describe an official test occurring at the end of each life cycle phase and demonstrating that software is ready for intended use. It includes the following:
  - Approved Test Plan and Procedure
  - Quality Assurance (QA) witnesses
  - Record of discrepancies (Problem Reports)
  - Test Report

  With the invention of more advanced software, the term "formal testing" also refers to a type of mathematical testing using Formal Methods. Formal Methods include the following types of tests:
  - Model Checking
  - Theorem Proving
  - Static Analysis
  - Runtime Monitoring

  Therefore, to avoid on confusion in this document, the traditional use of "formal testing" has been replaced with the term "official testing". The term "formal testing" used in this document means formal mathematical testing (i.e. Formal Methods).
- The term "Program" is used as a generic term to describe a mission or project conducted at NASA. For example, this document contains a survey of the Deep Space One Program, rather than the Deep Space One Mission.
- The term "Advanced Software" is used to describe rule-based expert systems, model-based reasoning software and/or artificial intelligence (AI) software.
- The term "meta-model" refers to a model in a generic or high-level format. It does not refer to the traditional use of the term "meta" meaning information about information.

2. EXECUTIVE SUMMARY

The NASA Ames Research Center Automated Software Engineering (ASE) group prepared this report as the deliverable for Task 5.3.3.1 “New V&V Tools for Diagnostic Modeling Environment (DME)” highlighted in green on Figure 1. It is the third report for Task 5.3.3 “V&V”, highlighted in blue on Figure 1.



Legend:

- SLI Space Launch Initiative
- IVHM Integrated Vehicle Health Management
- V&V Verification and Validation
- RLV Re-usable Launch Vehicle

*(**Note:** The current space shuttle is the 1st Generation RLV. The goal of SLI is to upgrade the space shuttle to the 2nd Generation RLV)*

Figure 1: SLI 2nd Generation RLV TA-5 IVHM Project Structure

The purpose of this report is to provide Correctness and Reliability Criteria for V&V of 2nd Generation RLV Diagnostic Modeling Environment, describe current NASA Ames Research Center (ARC) tools for V&V of Model Based Reasoning systems and discuss the applicability of Advanced V&V to DME.

This report is divided into the following three sections:

- Correctness and Reliability Criteria
- Tools for V&V of Model Based Reasoning
- Advanced V&V Applicable to DME

The Executive Summary includes an overview of the main points from each section. Supporting detail, diagrams, figures and other information are included in subsequent sections. A glossary, acronym list, appendices and references are included at the end of this report.

2.1. CORRECTNESS AND RELIABILITY CRITERIA

This section provides a list of criteria for model-based diagnosis systems like those considered for 2nd Generation RLV IVHM. The criteria relate to the following components:

- Diagnosis engine
- Diagnosis model
- Physical system

Correctness and Reliability criteria are listed in alphabetical order and described in Section 3.

- Accuracy
- Engine Correctness
- Integration Correctness
- Internal Sanity
- Physical System Correctness
- Suitability for Diagnosis

2.2. TOOLS FOR V&V OF MODEL BASED REASONING

This section describes new tools at NASA Ames Research Center (ARC) to automate verification of Livingstone Models, a key technology proposed for 2nd Generation RLV IVHM.

Tools include MPL2SMV, JMPL2SMV and Livingstone PathFinder. MPL2SMV and JMPL2SMV automatically translate Livingstone models to SMV for verification then translate back for verification results logs called counterexamples.

Note: *MPL2SMV is used for version one of Livingstone and JMPL2SMV is used for version two.*

Livingstone PathFinder provides an automatic way to analyze Livingstone-based diagnosis applications including a Livingstone simulator.

The benefits of MPL2SMV and JMPL2SMV are listed below:

- Provides robust verification of advanced IVHM software making it possible to meet stringent certification standards so the 2nd Generation RLV can take advantage of advanced IVHM software in flight and on the ground
- Shields Livingstone application designers from the technicalities of SMV (Symbolic Model Verifier) while providing them access to powerful model checking capabilities
- Improves reliability of Livingstone models making missions using Livingstone safer
- By investigating all model states during a simulation, MPL2SMV quickly finds anomalies during early phases of the Software Life Cycle, thereby reducing program costs

The benefits of Livingstone PathFinder include:

- Provides automatic way to analyze Livingstone-based diagnosis applications across a wide range of scenarios. Scenarios include tests for commands to be issued and faults to be injected making verification of Livingstone fast and robust.
- Contains a simulator for the device upon which diagnosis is performed making it possible for test engineers to create a very effective test bed

- Provides three search strategies: all errors, one error and shortest error trace. It also has optional reporting of all transitions and error traces.
- Automatically generates of draft script from the model to jump start the verification process

2.3. ADVANCED V&V APPLICABLE TO DME

First, this section includes an overview of DME to provide a foundation for discussing applicability of V&V Tools. Next, it provides an explanation of how new V&V tools (developed at NASA ARC by Charles Pecheur and Carnegie Mellon by Reid Simmons) meet correctness and reliability criteria. Finally, this section contains detailed recommendations for incorporating MPL2SMV/JMPL2SMV and Livingstone PathFinder into DME.

3. CORRECTNESS AND RELIABILITY CRITERIA

In general, correctness and reliability criteria for model-based diagnosis systems such as those considered as part of 2nd Generation RLV IVHM relate to the following three components:

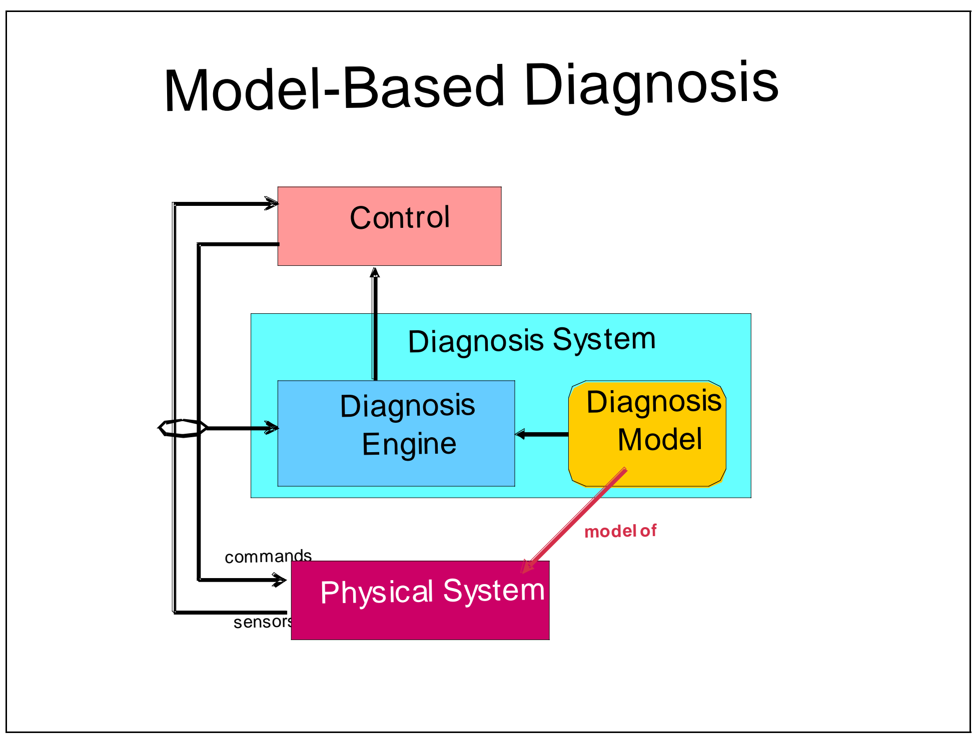


Figure 2: Model Based Diagnosis

- The diagnosis *engine* that applies generic reasoning principles to perform diagnosis
- The diagnosis *model* that provides domain-specific knowledge to the engine
- The *physical system* (with interfacing hardware and software) that is being diagnosed. System control (like the Executive in Deep Space One) receives sensor readings from the physical system and issues commands to the physical system.

Based on this the following correctness and reliability criteria can be defined:

Criteria	**Description**
Internal Sanity	Is the model internally well-formed? These properties are general sanity checks independent of the specific application. Typical examples include: • Completeness: does the model contain all needed constraints? An under-constrained model has missing information causing ambiguity. • Consistency: is the model free of unneeded constraints? An over-constrained model has too many restrictions resulting in contradictions.
Accuracy	Is the model a valid abstraction of the specified physical system? This is hard to do completely, because the physical system is a complex and heterogeneous entity, typically with a largely informal specification. Nevertheless, one can maximize the confidence in the model accuracy by verifying that it satisfies documented properties of the system (provided those properties can be expressed at the level of abstraction of the model). For example, suppose a physical property of the system requires water to flow in and out of a pipe with no water retention in the pipe. The model must indicate the amount of water in equals the amount out. If these two properties are not equal, then the model does not accurately reflect the physical system.
Suitability for Diagnosis	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 (the model is the physical system) and A perfect engine (diagnosis performs according to its specification)? Deficiencies against this criterion often reflect overall 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.
Physical System Correctness	Does the implemented physical system match its specification? This is impossible to formally prove in practice, considering that the implemented physical system is a real-world entity with no complete formal description. Nevertheless, rigorous development processes can minimize the risk and impact of discrepancies. It is important to ensure that interface drivers between the physical components and the diagnosis system correctly implement the desired abstractions. For example, sensor readings must be properly delayed after actuator changes to allow for physical propagation delays.

Criteria	**Description**
Engine Correctness	Does the implemented engine match its specification? This is a hard part, considering the complexity of the algorithms involved. It can be further decomposed in two steps: • Do the algorithms correctly compute the specified results (a mathematical proof problem)? • Does the engine code correctly implement those algorithms (a program verification problem)? However, 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.
Integration Correctness	Does the interaction of the diagnosis software with its operating environment meet system requirements?

Alternative approaches may use an application-specific engine, compiled from the model at development time rather than interpreted at run-time. In this case, the generic part is the compiler rather than the engine, but the arguments above remain essentially the same.

The first three criteria (internal sanity, accuracy and suitability for diagnosis) apply to the diagnosis model, while the remaining three apply to other parts of the diagnosis system and its environment.

Internal sanity is a pre-requisite for other application-specific criteria.

The next four criteria form the basis of the following compositional argument for global correctness:

IF the desired diagnosis can be performed,
assuming the specified model and engine (Suitability for Diagnosis), and
the implemented engine matches its specification (Engine Correctness), and
the specified physical system matches its model (Accuracy), and
the implemented physical system matches its specification (Physical System Correctness),
THEN the implemented system can perform the desired diagnosis.

Integration correctness carries this argument to the next system level.

Appendix A, Model-based Verification of Diagnostic Systems, contains a thorough discussion of issues surrounding verification of diagnostic systems like IVHM.

4. TOOLS FOR V&V OF MODEL BASED REASONING

According to the Survey of Current V&V Processes/Methods in Report 1, Advanced IVHM Software has been used primarily on experimental missions because it is difficult to adequately verify this software in accordance with NASA standards and guidelines for certification of airborne software for the following reasons:

- Autonomous onboard planning technology challenges the comfort level of the team responsible for certification per NASA guidelines. It is difficult to move past the mindset of expecting complete predictability from an autonomous system. However, the Deep Space One Remote Agent Experiment demonstrated that the paradigm shift is indeed possible because it performed a flawless demonstration of onboard planning and the formal verification techniques before and after flight proved successful in identifying deadlock issues and other anomalies common to autonomous systems.
- While there are no new processes, methods and tools for V&V of autonomous systems specifically noted in the NASA guidelines, these guidelines are flexible enough to allow addition of such techniques as long as they meet the overall criteria in the guideline. Report 2 contains a section describing how to incorporate new processes, methods and tools into the Software Life Cycle in accordance with NASA guidelines.
- Testing coverage of autonomous software behaviors becomes more difficult due to the larger numbers of possible combinations of parameters and higher numbers of possible interactions between subsystems. This is a testing coverage problem that can be mitigated by restricting harmful interaction “by design” and by using powerful, new V&V tools to automate testing of autonomous systems. Two new V&V tools are described below.

This section discusses new tools to automate verification of Livingstone Models, a key technology proposed for 2nd Generation RLV IVHM. Tools include MPL2SMV/JMPL2SMV and Livingstone PathFinder.

4.1. MPL2SMV

MPL2SMV is a tool to automate model checking for Livingstone models. It was co-developed by Charles Pecheur at NASA ARC and Reid Simmons at Carnegie Mellon University. Using MPL2SMV, developers can take an MPL model, specify desired properties in a natural extension of the MPL syntax, use SMV to check them and get the results in terms of their MPL model without reading or writing a single line of SMV code.

The benefits of MPL2SMV are listed below:

- Provides robust verification of advanced IVHM software making it possible to meet stringent certification standards so the 2nd Generation RLV can take advantage of advanced IVHM software in flight and on the ground
- Shields Livingstone application designers from the technicalities of SMV (Symbolic Model Verifier) while providing them access to powerful model checking capabilities
- Improves reliability of Livingstone models making missions using Livingstone safer
- By investigating all model states during a simulation, MPL2SMV quickly finds anomalies during early phases of the Software Life Cycle, thereby reducing program costs

This section includes the following:

- Introduction to MPL2SMV
- Overview of Livingstone

- Description of SMV
- Verification of Livingstone Models
  - Model Translation
  - Specifications Translation
  - Traces Translation
- Applications
  - Deep Space One
  - Mars In-Situ Propellant Production (ISPP) Plant
- Lessons Learned Using MPL2SMV

4.1.1. Introduction to MPL2SMV[1]

A Model Checking tool, like SMV, can efficiently check all possible execution traces of a system in a fully automatic way. By simulating every execution trace, anomalies appear that are difficult, if not impossible, to find using traditional testing methods.

In order to use SMV, the system being verified must be translated into special SMV syntax. Previously, this was a tedious and complex manual task. Now, MPL2SMV automates this translation process. It converts a Livingstone Model and the specification from Livingstone to SMV syntax and then converts a diagnostic trace from SMV back to Livingstone.

A full understanding of Livingstone and SMV is not required to use MPL2SMV; however, relevant information about both is included in the following sections as a basis for explaining the inner workings of MPL2SMV.

4.1.2. Overview of Livingstone[1]

Livingstone is a model-based health monitoring system developed at NASA ARC. It uses a symbolic, qualitative model of equipment to infer state and diagnose failures. Livingstone is one of the three parts of the Remote Agent (RA), an autonomous spacecraft controller (described in Report 1).

Livingstone is also used in other applications such as the control of a propellant production plant for Mars missions and the monitoring of a mobile robot.

Figure 3: Livingstone Mode Identification (MI) and Mode Recovery (MR shows how Livingstone functions.

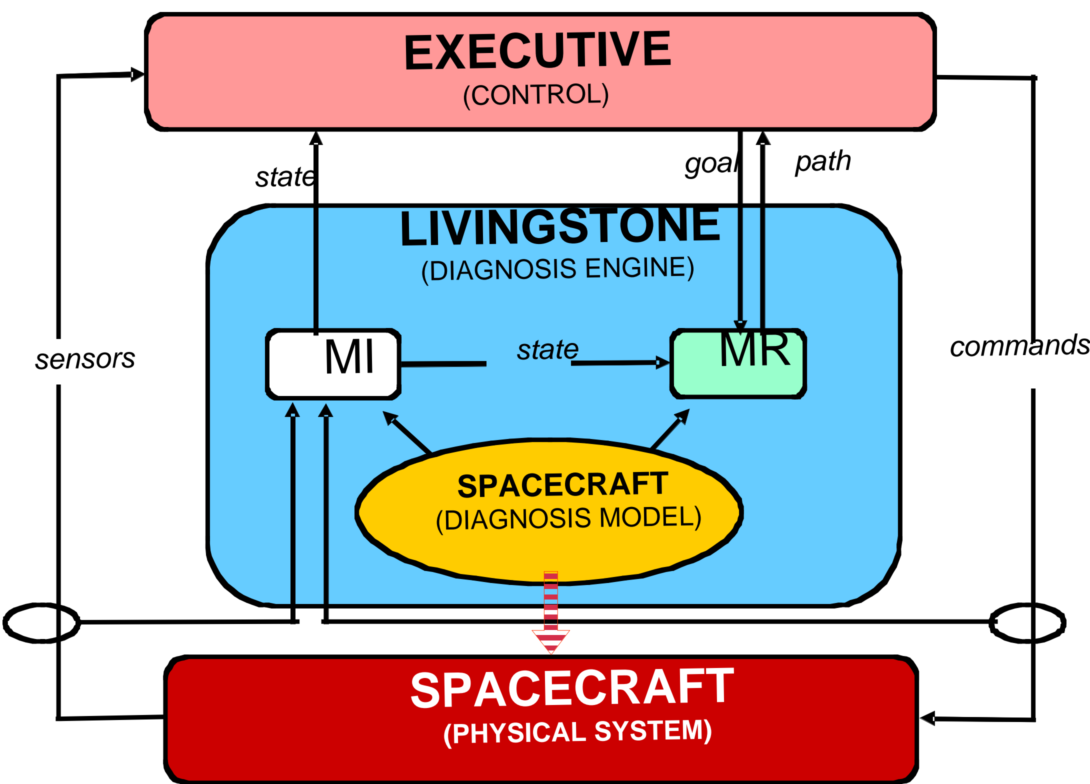


Figure 3: Livingstone Mode Identification (MI) and Mode Recovery (MR) [12]

The Mode Identification (MI) module estimates the current state of the system by tracking the commands issued to the device. Then, it compares the predicted state of the device against observations received from actual sensors. If a discrepancy occurs, Livingstone performs a diagnosis by searching for the most likely set of component mode assignments that are consistent with the observations. Using this diagnosis, the Mode Recovery (MR) can compute a path to recover to a given goal configuration.

The model used by Livingstone describes the normal and abnormal functional modes of each system component using Model Programming Language (MPL).

Components are parameterized and described using variables with discrete values. Each component has a set of modes identifying nominal and failure modes. Each mode specifies constraints on the values

that variables may take when the component is in that mode and how the component can switch to other modes.

```
(defvalues flow (off low nominal high))
(defvalues valve-cmd (open close no-cmd))
(defcomponent valve (?name)
      (:inputs (cmd "type valve-cmd))
      (:attributes ((flow ?name) :type flow) …)
      (closed :type ok-mode :model (off (flow ?name))
        :transitions ((do-open :when (open cmd) :next open) …))
      (open :type ok-mode …)
      (stuck-closed :type fault-mode …)
      (stuck-open :type fault-mode …))
```

Figure 4: Partial MPL Model of a Valve[1]

Figure 4: Partial MPL Model of a Valve[1] presents salient parts of a simple MPL model with a variable flow ranging over `{off, low, nominal, high}`. It has two nominal modes `open` and `closed` and two failure modes `stuck-open` and `stuck-closed`. The closed mode enforces `flow=off` and allows a transition `do-open` to the open mode, triggered when the `cmd` variable has value `open`.

4.1.3. Description of SMV[1]

SMV is a symbolic model-checking tool. Model checking is based on the exhaustive exploration of a system's achievable states. Given a model of a concurrent system and an expected property of that system, a model checker will run through all possible executions of that system, including all possible interleavings of concurrent threads, and report any execution that leads to a property violation.

Classic, explicit-state model checkers, like SPIN, trace system executions by generating and exploring every single state. This can lead to a problem called "state space explosion" because a system can have a huge number of states. Therefore, explicit-state model checkers sometimes run out of memory and cannot complete system verification.

Symbolic model checkers, like SMV, offer a technique for mitigating state space explosion. Instead of generating and exploring every state like explicit model checkers, symbolic model checkers manipulate whole sets of states at a time.

A set of states is evaluated for each transition by implicitly representing the states as the logical conditions the states satisfy. Sets of states are encoded as Binary Decision Diagrams (BDDs). BDDs are a special representation for Boolean formulas that is often more compact than traditional representations. BDDs are constructed by removing redundancy from Binary Decision Trees.[9]

A BDD is a graph structure used to represent a Boolean function, that is, a function f over Boolean variables $b_1,\dots,b_n$ with Boolean result $f(b_1,\dots,b_n)$. Each node of the graph carries a variable b_i and has two outgoing edges. It can be read as an *if-then-else* branching on that variable, with the edges corresponding to the *if* and *else* part. The leaves of the BDD are logical values 0 and 1 (i.e. false and true). For example, a BDD for $f(a,b,c) = a \vee (b \wedge c)$ is shown in the following figure:

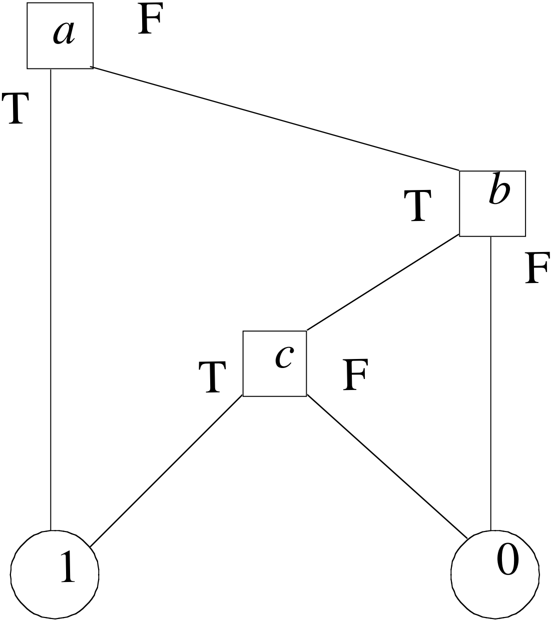


Figure 5: BDD for *a* V (*b* Λ *c*) [9]

Using BDDs, Symbolic Model Checking can verify much larger state spaces than explicit-state model checkers making it possible to verify larger, more complex systems.

4.1.4. Verification of Livingstone Models

MPL2SMV supports three kinds of translations as shown in the following figure.

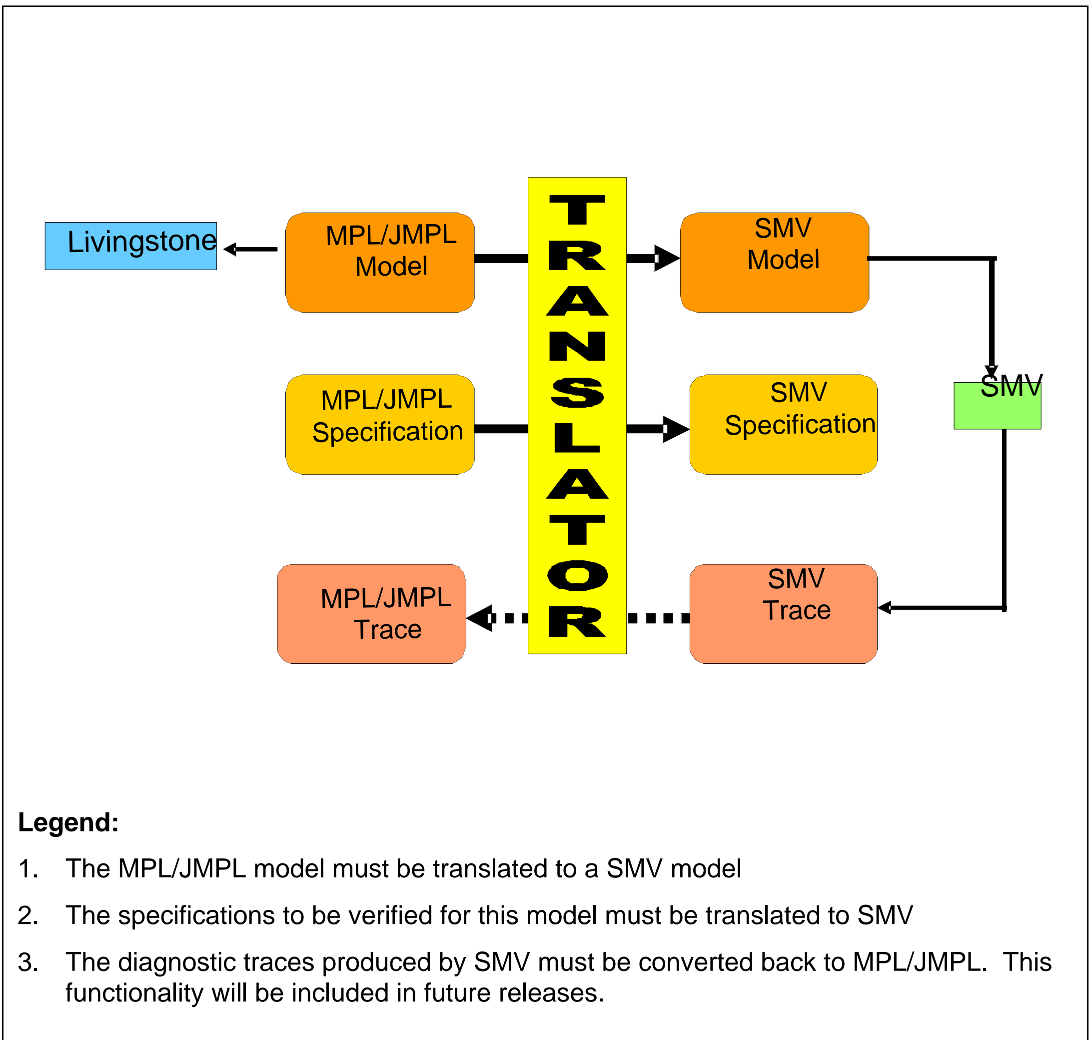


Figure 6: Translation between MPL/JMPL and SMV[1]

The following sections provide details about these three types of translations.

4.1.5. Model Translation[1 & 2]

The translation of MPL models to SMV models is facilitated by strong similarities between the underlying semantic frameworks of Livingstone and SMV. Both boil down to a synchronous transition system defined through propositional logic constraints: *I* on initial states, *C* all states and *T* on transitions. Based on this, there is a straightforward mapping of language elements from MPL to SMV shown in the following table:

MPL Description	MPL Syntax	SMV Syntax
Attribute type	`(defvalues T (V…))`	`{V, …}`
Elementary component	`(defcomponent C …)`	`MODULE C …`
Compound module	`(defmodule M…)`	`MODULE M…`
Component Attribute	`(:attributes (A :type T) …)`	`VAR A : {V,…};…`
Sub-components	`(:structure M …)`	`VAR X : M;…`
Mode Transitions	`:transitions …`	`TRANS …`
Mode Constraints	`:model …`	`INVAR …`
Component Constraints	`:facts …`	`INVAR …`
Initial conditions	`:initial-mode`	`INIT…`

Table 1: Mapping of MPL to SMV Elements[2]

Translation of transition relations must allow for fault transitions. Therefore, for each component a DEFINE declaration is produced that defines a symbol `faults` as the set of its fault modes. Then each MPL transition from a mode M to a mode M' with condition P is translated into the following SMV declaration:

```
TRANS (mode=M & P) -> (next(mode)=M' | next (mode) in faults)
```

Figure 7: SMV Model of a Valve presents the SMV conversion of the MPL model in Figure 4: Partial MPL Model of a Valve[1]

```
MODULE valve
VAR       mode:  {open,closed,stuck-open,stuck-closed};
          cmd: {open,close,no-cmd};
          flow: {off,low,nominal,high};
DEFINE faults:={stuck-open,stuck-closed};
INVAR mode=closed -> flow=off
TRANS (mode=closed  & cmd=open) ->
          (next(mode)=open | next(mode) in faults)
```

Figure 7: SMV Model of a Valve

4.1.6. Specifications Translation[1]

Specifications to be verified with SMV are added to a MPL model using the new `defverify` declaration. This `defverify` declaration also defines the top-level module to be verified.

Properties to be verified are expressed in a Lisp-like style that is consistent with the MPL syntax. Their translation function is an extension of the MPL logic formulae translation. The following three figures show a specification in English, MPL and SMV, respectively.

Specification: from a non-failure state, a high flow in valve 1 can eventually be reached

Figure 8: Specification in English[1]

```
(defverify
      (:structure (ispp))
      (:specification
            (always (globally (implies
                  (not (broken))
                  (exists (eventually (high (flow valve-1)))))))
```

Figure 9: Specification in MPL[1]

```
MODULE main
      VAR ispp : ispp;
      SPEC AG ((!broken) ->
                  EF (ispp.inlet.valve-1.flow=high))
```

Figure 10: Specification in SMV[1]

4.1.7. Traces Translation[1]

When a violated specification is found, SMV reports a diagnostic trace consisting of a sequence of states leading to the violation. This trace is essential for diagnosing the nature of the violation.

The trace lists variables by their SMV names making it challenging for a Livingstone developer to decipher. Future releases of the JMPL2SMV will convert the trace back to the original Livingstone model names.

4.1.8. Applications

MPL2SMV was used successfully in the following applications:

Deep Space One Remote Agent Experiment[1]
The Livingstone model for Deep Space One Remote Agent Experiment (DS1-RAX) consists of several thousand lines of MPL code. Using MPL2SMV, it was automatically converted to SMV models and several important internal sanity properties were verified consisting of:

- Consistency: Is the model free of unneeded constraints?

- Completeness of mode transition relations: Does the model contain all needed constraints?
- Reachability of each mode – Can states for each mode be reached?

Using MPL2SMV, engineers were able to identify several bugs in the DS1 models even after these models had been extensively tested by more traditional means.

In-Situ Propellant Production (ISPP) [1]

MPL2SMV was used at NASA Kennedy Space Center by the developers of a Livingstone model for the In-Situ Propellant Production (ISPP) system that will produce spacecraft propellant using the atmosphere of Mars.

Initial experiments indicated that MPL2SMV could easily convert this model into SMV to verify useful properties like reachability of normal operating conditions or recovery from failures.

The current ISPP Livingstone model has 10^{50} states but little depth (all states can be reached within at most three transitions). It can be verified in less than a minute using SMV optimizations!

The same internal sanity properties checked for DS1-RAX were verified for ISPP plus some model accuracy constraints were also checked. For example, if all active components were turned on and no fault occurred, then ISPP should function properly and produce chemicals.

4.1.9. Lessons Learned Using MPL2SMV[1]

Verification of a Livingstone model is very different from verification of more conventional concurrent applications.

While a typical concurrent system is a collection of active entities each following a well-scoped algorithm, Livingstone describes a passive component such as a tank, valve or sensor and states how this component reacts to external commands. It hardly ever imposes any kind of order of operations on the component itself.

Additionally, failures amount to unrestricted spontaneous transitions in every component that allows them. This results in state spaces that have a peculiar shape:

- A huge branching factor (one state can transition to a huge number of next states) due to all the command variables that can be set and all the failures that can occur at any given step
- Very low depth (reach all states with very few transitions), due to little inherent sequential constraints in the model

This peculiar shape affects the manner in which properties that can be verified. Consistency and completeness properties can be verified. It is important to verify these properties because the declarative nature of MPL makes it easy to develop an over- or under-constrained model.

MPL2SMV is a useful tool but has a few limitations:

- Cannot establish that the Livingstone MI will properly identify a situation, but it can establish that there is insufficient information to properly identify a situation. This line of work is under investigation at NASA Ames.
- Does not address the interaction of Livingstone with other parts of the system including real hardware with hard timing issues

4.2. JMPL2SMV

Reid Simmons at Carnegie Mellon University developed JMPL2SMV. It works like MPL2SMV, but was designed to translate the new model type in the new version of Livingstone.

Livingstone was originally written in LISP. However, due to problems certifying LISP to fly on NASA missions, Livingstone was re-written in C++. To avoid confusion the new Livingstone software was nicknamed L2. It functions like Livingstone, but L2 has a new modeling language is called JMPL.

JMPL2SMV translates the JMPL model to SMV. For purposes of this report, JMPL2SMV functionality and benefits are identical to MPL2SMV described in section 4.1.

4.3. Livingstone PathFinder (LPF)

The Livingstone PathFinder (LPF) is a tool developed by Charles Pecheur, NASA ARC, for automatically analyzing Livingstone-based diagnosis applications.[3]

The benefits of Livingstone PathFinder (LPF) include:

- Provides automatic way to analyze Livingstone-based diagnosis applications across a wide range of scenarios. Scenarios include tests for commands to be issued and faults to be injected making verification of Livingstone fast and robust.
- Contains a simulator for the device upon which diagnosis is performed making it possible for test engineers to create a very effective test bed
- Provides flexible search strategies in different modes
- Automatically generates of draft script from the model to jumpstart the verification process

How LPF Works

LPF receives the following input:

- Livingstone model in a pre-compiled XMPL format with its associated harness and initial state files
- Script describing a flexible scenario of events, a non-deterministic program describing a whole family of possible executions. Scenarios describe both the commands to be issued and the faults to be injected.

After obtaining input, LPF creates a Livingstone engine embedded into a simulation testbed and runs that assembly through all executions described by the scenario. While running the scenario, LPF checks for various selectable error conditions.

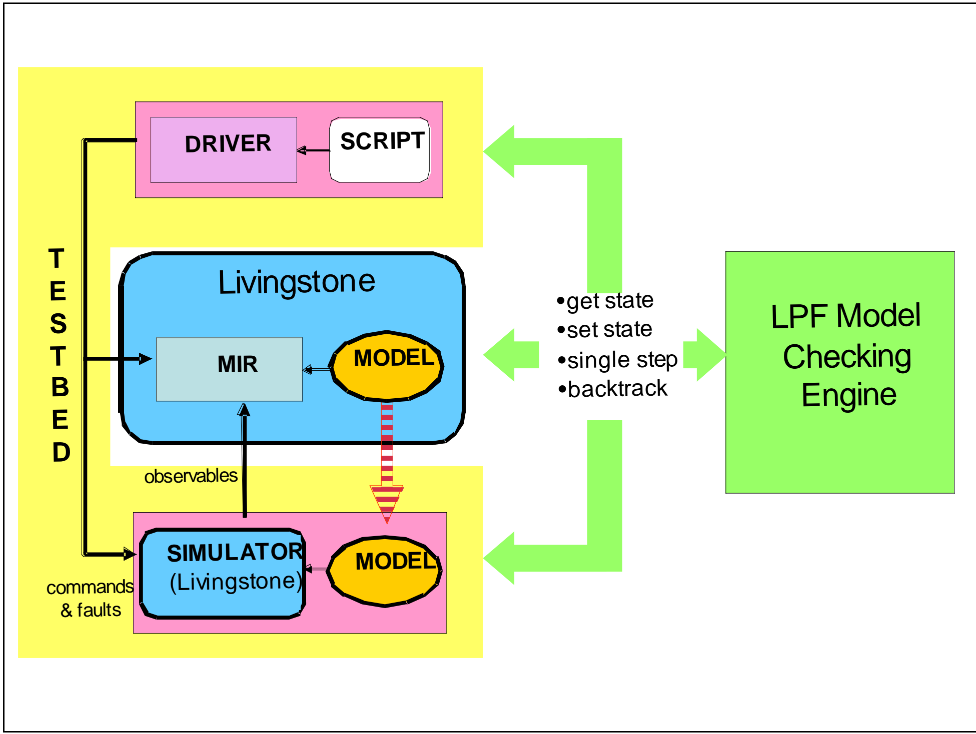


Figure 11: Livingstone PathFinder

LPF Testbed
As shown in the figure above, the generic LPF testbed consists of the following:

- Driver: a component that enumerates the commands and faults according to a script provided either as a file or directly entered on the command line. If no file or command line script is found, LPF looks for a file with the same name as the model and an extension of .lpf.
- MIR: a Livingstone engine performing diagnosis initialized from the user-provided Livingstone model
- Simulator: a simulator for the device on which diagnosis is performed. Currently, this is a second Livingstone engine initialized from the same or a separate model used for simulation
- Model Checking Engine: decides which paths to explore. Currently, it is a simple depth-first search algorithm, but it may be replaced by more elaborate searches (goal-driven, interactive, random, …) in the future.

These three components are instrumented so LPF can single-step in both forward and backward directions through an execution. A single-step corresponds to one Livingstone diagnosis cycle.

Error Conditions Supported
LPF supports checking for the following kinds of error conditions:

- Simulator consistency – reports when the simulator reaches an inconsistent state. This typically occurs after executing a command or injecting a fault that results in conflicts among the model constraints and assignments. LPF also reports when no state completion was found.
- MIR consistency – reports when MIR reaches an inconsistent state. This can occur either following the interpretation of a command or the updating of observables, if diagnosis fails to find any candidate.

- Mode comparison – compares the modes of all components in the simulator to those assumed by MIR and report when there is a discrepancy. This provides a first estimation of states where diagnosis may fail to perform as desired.

When an error has been reported, the search can proceed in either of the following ways:

- Find one error and stop searching
- Find all errors
- Find the shortest error trace: the search proceeds after finding an error but LPF limits further execution to the depth of the last error found. This will report further errors only if they have shorter traces than the last one reported.

Inconsistent states of either the Simulator or MIR are terminal states so execution will not proceed past them even though a scenario may still have further events to process. Instead, the exploration will backtrack to alternate routes.

LPF Modes
LPF is very flexible and provides the following modes of execution:

- Full exploration: complete setup with Driver, Simulator and MIR. This is the default.
- Simulation only: commands and faults are applied only to the simulator and not to MIR
- Driver only: LPF prints a complete execution trace of the driver
- Count: LPF computes and reports the number of state and traces produced by the driver for each execution depth. This is useful to approximate the time required to verify a scenario. It is good practice to count before testing complex scenarios.
- Create default script: No analysis is performed and the Livingstone model is scanned for commands and fault modes to generate a default script. This script can be used to jumpstart the verification process. It can be edited to fit more complex testing needs.

Summary
Livingstone PathFinder provides significant improvement over traditional test case generation. It is currently being tested to verify Livingstone based projects at NASA ARC and initial feedback is very positive.

5. ADVANCED V&V APPLICABLE TO DME

The purpose of this section is to explain how Advanced V&V tools like MPL2SMV are beneficial to the DME system planned for the 2nd Generation RLV. Because this section contains information described in documents falling under the purview of the U.S. Munitions List (USML) as defined in the International Traffic in Arms Regulation (ITAR), 22 CFR 120-130, it is export controlled and not available on this website.

6. ACRONYMS

Term	Definition
ACB	Application Control Board
AI	Artificial Intelligence
ANSI	American National Standards Institute
BDD	Binary Decision Diagram
CCB	Change Control Board
CCP	Command & Control Processors
CCWS	Command & Control Workstations
CLCS	Checkout & Launch Control System
CM	Configuration Management
CMM	Capability Maturity Model
COTS	Commercial Off The Shelf
CVS	Concurrent Version System
DAR	Delivery Acceptance Review
DCR	Design Certification Review
DDP	Data Distribution Processors
DP-2	Design Panel 2 – Requirements Design Panel
DRP	Data Recording Processors
DS1	Deep Space One
EIA	Electronic Industries Association
EXEC	Smart Executive or EXEC
FEMA	Failure Mode Effects Analysis
FEMCA	Failure Mode Effects and Criticality Analysis
GSE	Ground Support Equipment
HIT	Hardware Installation Test
IEC	International Electro-technical Commission
IEEE	Institute of Electrical and Electronic Engineers
IPS	Ion Propulsion System
ISO	International Organization for Standardization
IV&V	(NASA) Independent Verification & Validation
IVHM	Integrated Vehicle Health Management
LPS	Launch Processing System
MICAS	Miniature Integrated Camera and Spectrometer

Term	Definition
MIL STD	Military Standard
MIR	Mode Identification Reconfiguration (also referred to as Livingstone)
NASA	National Aeronautical and Space Administration
NASA ARC	NASA AMES Research Center
NASA/KCS	NASA Kennedy Space Center
NPD	NASA Policy Directive
NPG	NASA Procedures and Guidelines
O&M	Operations and Maintenance
ORR	Operational Readiness Review
ORT	Operational Readiness Tests
PCO	Project Controls Office
PR	Problem Report
PTR	Post-Test Review
RA	Remote Agent
RAX	Remote Agent Experiment
RCS	Reaction Control System
RLV	Re-usable Launch Vehicle
RMA	Reliability, Maintainability, Availability
RTC	Real Time Control
RTCA	Requirements and Technical Concepts for Aviation
STP	Software Test Plan
SVP	Software Validation Procedures
SW	Software
TC	Test Conductor
TR	Test Report
TRR	Test Readiness Review
UML	Unified Modeling Language
UPPAAL	Acronym based on a combination of UPPsala and AALborg universities
USA	United Space Alliance
V&V	Verification & Validation
VMC	Vehicle Management Computer
VMS	Vehicle Management System

Note: *More Acronyms: http://www.ksc.nasa.gov/facts/acronyms.html*

7. GLOSSARY

Term	Definition
Advanced Software	The term "Advanced Software" is used to describe model-based and/or artificial intelligence (AI) software like model based reasoning software.
Algorithm	A rule or procedure for solving a problem[4]
Aliasing	An alias at program point *t* is a pair *(u, v)* of references ((u,v) $\in R^2$ where *R* is the set of references in the program) that point to the same store location. In other words, *u* and *v* give access to the same object.
Automaton	Machine evolving from one state to another under the action of transitions. For example, a digital watch can be represented by an automaton in which each state represents the current hour and minutes (seconds omitted in this example), there are 24x60 = 1440 possible states and one transition links any pair of states representing times one minute apart[5].
Autonomous Systems	Autonomous systems rely on intelligent inference capabilities to be able to take appropriate actions even in unforeseen circumstances. They enable a whole range of new applications, such as sending autonomous robots to places where it is too dangerous or expensive for humans and/or where human control is difficult or not technically feasible[6] (for example, deep space).
BDD	Binary Decision Diagram – data structure for representing Boolean functions that allows efficient computations and is used for symbolic model checking.
BEAM	Beacon-based Exception Analysis for Multimissions is an end-to-end method of data analysis intended for real-time fault detection. It provides a generic system analysis capability for potential application to deep space probes and other highly automated systems[7]
CAD	Computer Aided Drafting. Software environment for drawing architectural plans and engineering diagrams.
Deterministic Software	Software that always yields the same result for the same input
DSI eXpress	Diagnostics Development Tool for functional modeling to enable convergence of design diagnostics while providing design tradeoff analysis between Testability, Reliability, Maintainability and Availability concerns. For more information: http://www.dsiintl.com/Products/index.asp
Failure	Inability of a component or system to perform, as designed, during operational and maintenance service life. For the purposes of IVHM, the terms fault detection/isolation and failure detection/isolation are interchangeable.
Fairness Property	Fairness property expresses that under certain conditions something will or will not occur infinitely often
Fault	A hardware or software anomaly that propagates into a failure or maintenance event. A fault may be an induced, manufacturing or material defect. In software, a fault is caused by defective, missing or extra instructions or sets or related instructions that result in one or more actual failures or create a problem that results in a maintenance event.
Fidelity	Integrity of testbed. For example: low fidelity testbed may have a simulator rather than actual spacecraft hardware. The highest fidelity testbed is the actual hardware being tested

Term	Definition
FMEA/FMECA	FMEA – Failure Mode Effects Analysis FMECA - Failure Mode Effects and Criticality Analysis FMEA and FMECA aides in determining what loss of functionality occurs due to an unremediated fault state
Formal Testing	The term “formal testing” has two meanings. Traditionally, “formal testing” has been used to describe an official test occurring at the end of each life cycle phase and demonstrating that software is ready for intended use. It includes the following: o Approved Test Plan and Procedure o Quality Assurance (QA) witnesses o Record of discrepancies (Problem Reports) o Test Report With the invention of more advanced software, the term “formal testing” also refers to a type of mathematical testing using Formal Methods. Formal Methods include the following types of tests: o Model Checking o Theorem Proving o Static Analysis o Runtime Monitoring Therefore, to avoid on confusion in this document, the traditional use of “formal testing” has been replaced with the term “official testing”. The term “formal testing” used in this document means formal mathematical testing (i.e. Formal Methods).
Interleaving	Ordering of concurrent events in a program
JMPL	Java-style Model Programming Language for designing Livingstone models
k-limiting	Lists only up to a depth of k or k is the limit of the list
LISP	Functional programming language widely used for AI applications
Liveness Property	Liveness property expresses that under certain conditions something will eventually occur.
Mode Identification	Mode Identification observes commands, receives state information and uses model-based inference to deduce the state and provide feedback
Mode Reconfiguration	Mode Reconfiguration serves as a recovery expert. It takes constraints and uses declarative models to recommend a single recovery action.
Model Checking	Technique for verifying finite-state concurrent systems[9]
Nominal	Expected behavior for no failure, for example: nominal behavior for a valve may be “open” or “shut”
Non-deterministic Software	Software that can yield different results for the same input

Term	Definition
Off-nominal	Unexpected failure behavior, for example: off-nominal behavior for a valve may be “stuck open” or “stuck shut”
Partial Order Reduction	Reduces the number of interleaving sequences that must be considered for model checking
Program	The term “Program” is used as a generic term to describe a mission or project conducted at NASA. For example, this document contains a survey of the Deep Space One Program, rather than the Deep Space One Mission.
Reachability Property	Reachability property states that some particular situation can be reached
Regression testing	Extrapolates the impact of the changes on program, application throughput and response time from the before and after results of running tests using current programs and data.[8]
Safety Property	Safety property expresses that under certain conditions, something never occurs
Software Life Cycle	The Software Life Cycle is defined as the steps or phases required to develop software, starting with a concept and ending with a working product or system
State	Snapshot or instantaneous description of the system that captures values of variables at a particular instant in time[9]
Temporal Logic	Temporal logic is a form of logic that provides the capability to reason about how different states follow each other over time
Transition	The change described by the state before an action occurs and the state after the action occurs
UPPAAL	UPPAAL is a toolbox for validation and verification of real-time systems described as networks of timed automata. The current version of UPPAAL is implemented in C++, Motif and Xforms. The traditionally encountered explosion problems are dealt with in UPPAAL, by a combination of on-the-fly verification, together with a symbolic technique reducing the verification problem to that of solving simple linear constraint systems. In addition, optimization techniques for reducing the time and space requirements of the verification procedure have been implemented.
Validation	Ensuring that each step in the process of building the software yields the right products (Build the Right Product)
Verification	Ensuring that software being developed or changed will satisfy functional and other requirements (Build the Product Right)

8. Appendix A: Model-based Verification of Diagnostic Systems

Working notes by Dr. Steve Brown, Northrop Grumman

Verification and validation of diagnostic[1] systems is very important, because the diagnostics presented may lead to crew/flight/mission safety critical decisions. In addition, the actual costs associated with informed maintenance will be strongly influenced by how well informed that maintenance is.

Sections 16.2 - 16.4 of this paper discuss diagnostic system requirements (nominal and failure modes) in terms of both the infinite-dimensioned client system space and the finite-dimensioned diagnostic system space. Unfortunately, it is difficult to create an explicit detailed representation of modes in either space, because the parameter values associated with each mode are themselves complex functions of other parameters, and some or all of these parameters may be infinitely valued. Sections 16.5 - 16.7 discuss shifting the focus of verification from verification of the system as a whole to verification of a problem-specific model used within the system. In addition, to the extent that the diagnostic problem definition explicitly declares system modes, these modes may themselves be defined by functional models of system behavior. These "definition models" will themselves then require validation against the real physical system.

8.1. Verification to Design Requirements

The NASA Goddard Independent Verification and Validation Group define Verification and Validation as:

Verification *asks, "Is the product being built right?" It is the process of determining whether or not the products of a given phase of the software development cycle fulfill the established requirements.*

Validation *asks, "Is the right product being built?" It evaluates software at the end of the development lifecycle to ensure that the product not only complies with standard safety requirements and the specific criteria set forth by the customer, but performs exactly as expected*[2]*.*

Validation compares the finished product against top level customer requirements, as discussed in Section 2. Verification does the same, except it compares each phase and/or component in the system development process to a set of derived requirements defined for each phase and/or component. *In verification, the correctness criteria applied at the end of each phase are that the system or component meets the design requirements imposed at the beginning of that phase.* Verification is simpler, in that the requirements for each phase are usually more explicit than the overall validation requirements. On the other hand, this adds the problem of validating the requirements generated for each phase.

8.2. Problem Statement in Diagnostic Space {*D*}

The diagnostic problem consists of examining an engineering system (the client) and determining whether its current state corresponds to one or more distinct failure modes[3]. In particular, a diagnostic system attempts to divide all possible states of the system into:

1. States which represent nominal modes of the system (everything responding normally).
2. States which represent off-nominal modes of the system, including:

a) States which represent one or more defined failure modes of the system,
b) States which represent one or more undefined failure modes of the system, and

[1] As usual for TA-5 documents, the term "diagnostic" can generally be taken to also include "prognostics" over various timeframes.
[2] http://www.ivv.nasa.gov/faq/index.shtml
[3] Two modes are distinct if they are distinguishable, given all possible information about the client system. We will assume that by definition that all failure modes are distinct from all nominal (non-failure) modes.

c) States which represent some mixture of a) and b).

In practice, however, a diagnostic system will not have an omniscient view of its client. Instead the diagnostic system will define the client system in terms of a finite set of diagnostic parameters {*D*}, including:

- parameters with known or imposed values
- parameters that can be sensed
- parameters that can be inferred from other parameters (including parameters evolved through prior history of the system).

In general a diagnostic parameter

- may be enumerated, integral, or continuous
- need not be finite in range
- need not be independent of the other diagnostic parameters.

The *prognostic* problem consists of examining an engineering system (the client) and determining whether its current state corresponds to a *precursor* state associated with one or more distinct failure modes. Although the remainder of this discussion will focus on the diagnostic problem, it can readily be extended to prognostics by simply treating the precursor modes as "subcritical" failure modes.

8.3. Verification of the Diagnostic System Requirements

The true client system is typically described by an infinite number of possible engineering parameters, covering both its internal state and its environment (external state). The diagnostic system approximates this infinite parameter space with a finite diagnostic space {*D*}, which implies that the diagnostic system can never exactly express the full range of the client system. General verification of the diagnostic system thus consists of two problems:

1. *Verification of the Diagnostic System Requirements*: Verifying that the diagnostic space, and the problem description defined in terms of that space, are adequate to express the requirements demanded of the actual client system. This requirement can, in turn, be broken down into two sub-requirements:
   a) Verifying that the diagnostic space, and the problem description defined in terms of that space, are adequate to express the requirements demanded of the actual client system *as those requirements are understood at design time*.
   b) Verifying that the actual deployed client system (including its environment) match the design time understanding of that system to an acceptable degree.

2. *Verification of the Implemented Diagnostic System*: Verifying that the implemented diagnostic system fulfills the requirements given *as defined within that diagnostic space*?

In general, verification of the implemented system against the diagnostic system design requirements (2) is a more tractable problem than verifying the design requirements against the actual client system requirements (1).

8.4. Fundamental (Black Box) Requirements

The fundamental requirements of a diagnostic system can, in principal, be defined without regard to the manner in which the requirements are implemented; i.e. treating the diagnostic system as a black box.

The first set of requirements defines correct operation of the diagnostic algorithm(s):

1. *Recognition of Nominal Mode(s)*
   For all possible states of the system *for which nominal mode descriptions have been provided*, the system shall correctly identify all states that correspond to defined nominal modes[4].

2. *Detection of Anomalous Modes*
   For all possible states of the system *for which nominal mode descriptions have been provided*, the system shall correctly identify all states that do not correspond to any defined nominal mode.

3. *Nominal Robustness*
   For all possible states of the system, the system shall correctly identify all states that do not correspond to a defined nominal mode (i.e. ones for which no nominal mode descriptions have been provided). Nominal robustness prevents the system from reporting an anomalous mode simply because it does not know what "nominal" would look like in this state. Instead the system should report "undefined state".

4. *Fault Recognition*
   For all possible states of the system *for which description(s) of a given failure mode have been provided*, the system shall correctly identify all states that correspond to the defined failure mode.

5. *False Alarm Avoidance*
   For all possible states of the system *for which description(s) of a given failure mode have been provided*, the system shall correctly identify all states that do not correspond to the defined failure mode.

6. *Fault Robustness*
   For all possible states of the system, the system shall correctly identify all states for which a given failure mode has not been defined.

   Assuming the above requirements are all met, we can then define an additional *desirable* condition for the system:

   Ideal Fault Isolation
   For all possible states of the system, there should
   - Never be a state which corresponds to one or more defined nominal modes and one or more defined failure modes (nominal/failure ambiguity)
   - Never be a state which corresponds to two or more distinct failure modes (failure ambiguity) unless the ambiguities in question have been specifically allowed.

It is important to recognize that, while requirements 1-6 define correctness of the algorithm, the maximum possible extent to which fault isolation can be achieved is a property of the problem definition itself. The diagnostic system space normally spans a finite number of discrete parameters. It is quite common to have cases in which, for some region(s) of the diagnostic space, a failure mode overlaps one or more different nominal and/or failure modes.

- Given this diagnostic space and problem definition, *no possible algorithm can isolate beyond this ambiguity group.*
- Failure of an algorithm to meet requirements 1-6 can, however, increase the level of ambiguity beyond the minimum possible.

---

[4] This description is not quite as circular as it may appear. For a diagnostic system defined over $\{D\}$, a given mode definition might be valid over a fixed range of some subset of these parameters $\{d\} \subseteq \{D\}$. The mode definition itself would then typically define nominal ranges for the complimentary set $\{D\} - \{d\}$.
Example: Assume the diagnostic space $\{D\}$ = {*launch_phase*, *temperature_1*, *temperature_2*}, where *launch_phase* can take on the enumerated values [*on_pad, lift_off, cleared_tower*].
If $\{d\}$ = {*launch_phase*}, then a particular nominal or failure mode could be defined as limits on the complementary set {*temperature_1*, *temperature_2*} for *launch_phase* = *lift_off* and *launch_phase* = *cleared_tower*. In the subspace corresponding to *launch_phase* = *on_pad* the mode would be undefined.

Thus a feasible requirement on fault isolation must be worded something like:

7. *Fault Isolation*

For all possible states of the system, to the limit possible within the given diagnostic state representation, there shall

- never be a state which corresponds to one or more defined nominal modes and one or more defined failure modes (nominal/failure ambiguity)
- never be a state which corresponds to two or more distinct failure modes (failure ambiguity) unless the ambiguities in question have been specifically allowed.

In principal it is always possible to disambiguate two modes, given additional system information necessary and sufficient to the disambiguation in question. Put another way, if the diagnostic space spanned all possible system parameters, all distinct modes could, by definition, be disambiguated. In practice what this means is that even if a given diagnostic space cannot disambiguate a set of modes, sometimes a second system can by employing a different diagnostic space.

8.5. Derived (Implementation Specific) Requirements

Unfortunately, the only direct way to prove the fundamental requirements 1-7 would be to examine every possible point in the system space (exhaustive search). Even if we restrict ourselves to verifying the diagnostic system against its design requirements, the number of test points in $\{D\}$ typically ranges from large to infinite.

An alternative approach that indirectly verifies the fundamental requirements can be applied when three conditions are met:

1. The diagnostic system employs a specific set of algorithms, methods and procedures that are known to meet these requirements, so long as the problem specific inputs to the system (the "model") are correct.
2. The specific implementation of these algorithms, methods and procedures, is shown to be correct.
3. The problem specific model is shown to be correct.

Validation of any specific algorithm set is beyond the scope of this program. In practice certain algorithms are usually "trusted", and accepted without further validation.

The level of validation typically applied to a specific implementation (diagnostic engine) depends on both its "trust" level and the tools available for validation. Because the diagnostic engine is typically written in a general purpose programming language, it can be very difficult to "prove" its correctness. Engines are normally validated through general SEI-type practices (i.e. written under a formal Software Development Plan, validated through some kind of test suite).

Our specific focus in the TA-5 is on verification of the problem specific model. Under the assumptions listed above, it should be clear that it is this model which contains all of the diagnostic system's knowledge about system specific nominal and failure modes. The way in which that knowledge is represented, and what constitutes verification of that knowledge, depends on the specific form of the model (which is, in turn, a function of the specific algorithm and engine to be applied).

Verification of the model to the engine is primarily a matter of syntax; i.e. validating that the specific representation of the model provided to the engine will be interpreted as intended with reference to the underlying algorithm.

This leaves only verification of the model with respect to the original design problem, where verification is understood to be within the context of the underlying algorithms to be applied.

8.6. Deficient Algorithm/Model Forms

Above, it was stated that the algorithm set applied is required to meet fundamental requirements 1-7, so long as the model itself is correct. In practice we sometimes use diagnostic algorithms that are known to not meet all of these requirements. Such an algorithm may be deficient in one or more of these requirements, yet still be useful for certain applications. In this case "verification" techniques may be used to flush out deficiencies, which then must be either corrected (if correctable) or accepted (if inherent to the algorithm).

8.7. Model Verification and Validation: Functional (System Hardware) Models

In Section 5 it was claimed that, by using trusted algorithms, the diagnostic system verification problem could be reduced to verification of its system specific model. In particular, algorithms such as Model-Based Reasoning (MBR) represent the nominal and failure modes of a system by constructing a functional model of the physical system itself.

- The parameters of this model constitute the diagnostic space $\{D\}$.
- The absence or presence, and perhaps severity of each defined root failure mode is typically a parameter of the system, with the observable effects of the failure on the system (fault signature) generated through causality relations built into the model.
- Nominal modes are simply defined as the system response in the absence of any defined failures.
- The control states of the model $\{d\} \subseteq \{D\}$ typically represent external environment and commanded states. As discussed in Section 4, one of the characteristics of a well behaved model the ability to recognize when the values of $\{d\}$ are out of a range for which the functional model is known to be valid.

Because functional models directly model the physics of the client system, they have some interesting verification characteristics. In particular:

1. The diagnostic system requirements are encoded directly into the model; and this model itself may be the only extant detailed definition of the system's normal and failure modes. If there *is* an external definition of the client system modes, it will typically be in the form of an even more elaborate functional model (which would itself require verification).
2. Validation of the diagnostic system requirements against the client system requirements, at design time or in operational use, consists of validating the functional model against the client system.

Based on the above arguments, we will now claim the following:

Verification of a diagnostic system based on a functional model consists of:

1. *Universal validation (or trust) of the diagnostic algorithms and their specific implementation.*
2. *Specific verification of the problem specific functional model against the physical characteristics of the client system:*
   a) *Verification that the representation of the client system is correct (within the limitations of the given model form).*
   b) *Verification that the representation of the client system is adequate to express the behaviors of interest, including specifically the response of sensed observable parameters to internal and external system* states.

Validating correctness of the functional model can include the use of formal methods to prove certain characteristics and/or invariants that are known to be true of the client system, either because they are true of the client system specifically, or because they are true for *all* physical systems. The physical world, including the client system, is always self-consistent, and there are a number of physical laws (e.g. conservation of mass) which must always hold. This means that large regions of the parameter space

{*D*} can be declared invalid, and any legal transition within the model from a valid state to an invalid one represents an error in the model.

Design verification of the diagnostic system against its client system is more difficult, because it requires comparison to an independent and more trusted representation of that system. Ultimately verifying the design understanding of the system against the physical reality can occur only after a sufficient number of deployed systems have seen service, to establish both baseline behavior and variance over the life of the system.

Finally, although the arguments of this section were focused on diagnostic models that are directly based on functional models, they apply to some extent to any model that uses a functional model in its construction. For example, a neural net trained using a simulation (functional model) of the client system will, in general, not be valid unless the functional model used for training is also valid.

8.8. Conclusions

In the development of diagnostic systems both the overall validation requirements and the specific verification requirements are expected to be of the form:

> Given a set of sensible parameters $\{s\} \subseteq \{D\}$, detect failure mode "x" (if present), isolated to within the fault isolation group {"y"}

Unfortunately, the normal and failure modes of the system will typically be described by reference to a functional model of the client system. Diagnostic systems which themselves operate on a functional model of the client system are, therefore, well suited to solving the stated problem; but are difficult to verify, because they create the detailed requirements specification within their own implementation. They are, in as sense, self-verifying, but only if the detailed requirements can be validated against the actual client system. Two general approaches to this problem have been identified as applicable to the design phase:

1. Formal methods applied to the functional model, used to verify that certain universal and problem specific conditions and invariants are met for all applicable states of the model.
2. When available, simulation-based testing. This involves presenting the diagnostic model with a sequence (one or more) of inputs (control and/or sensible parameters), and then comparing its resulting diagnoses against results which are either known to be correct, or are at least "trusted" to a higher degree than the diagnostic model.

Final validation of the system requires operational deployment of the client system, in order to generate test sets which are certain to be correct. Even in this case, however, the client system may never explore all possible failure states, and discovering incorrectly represented failure states may have catastrophic costs. For these reasons our primary focus will remain on "up front" design verification of the system, despite its inherent difficulty.

9. REFERENCES

[1] Charles Pecheur and Reid Simmons. *Livingstone to SMV Formal Verification for Autonomous Spacecrafts.* In:Proceedings for First Goddard Workshop on Formal Approaches to Agent-Based Systems, NASA Goddard, April 5-7, 2000. To appear in Lecture Notes in Computer Science, Springer Verlag.

[2] Charles Pecheur, Reid Simmons, Peter Engrand. *Formal Verification of Autonomy Models From Livingstone to SMV.* August 31, 2001. Submission to FAABS

[3] Charles Pecheur, *Livingstone PathFinder User Manual Release 1.0,* 2001

[4] Microsoft Corporation. "Windows 2000 Server Resource Kit Online Books", MSDN Library Glossary. 1985-2000.

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

[6] Reid Simmons, Charles Pecheur, Grama Srinivasan. *Towards Automatic Verification of Autonomous Systems.* In: Proceedings of the 2000 IEEE/IRSJ International Conference on Intelligent Robots and systems, IEEE 2000.

[7] Ryan Mackey, Mark L. James, Han Park, Michail Zak. *BEAM: Technology for Autonomous Self Analysis.* IEEE. Updated September 15, 2000.

[8] Jeffrey L. Whitten and Lonnie D. Bentley. *Systems Analysis and Design Methods Fourth Edition Instructor's Edition.* Irwin McGraw-Hill 1998 p. 583.

[9] Edmund M. Clarke, Jr., Orna Grumberg, Doron a. Peled, *Model Checking.* The MIT Press, 2000.