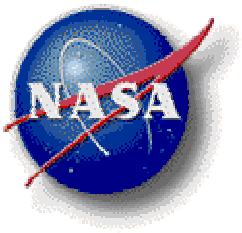
V&V OF ADVANCED SYSTEMS AT NASA

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, 2002

Contributors: Stacy Nelson, CSC
Charles Pecheur, RIACS

APPROVALS

Approvals:		
Task Lead:	Charles Pecheur	
Project Manager:	Marshal Merriam	
Area Manager:	Michael Lowry	

RECORD OF REVISIONS

REVISION	DATE	SECTIONS INVOLVED	COMMENTS
Initial Delivery	10/5/01	Sections 1, 2, 4, 7, 10, and 11	This is a draft only and not intended to be the final deliverable.
Initial Delivery	11/02/01	New Sections: 3, 4.2, 7, 8, 9, 12, 13, 14, 15 and 16 Revised Sections: 1, 2, 10 and 11	This is a draft only and not intended to be the final deliverable.
Initial Delivery	11/30/01	New Sections: 5, 6 and 9 Revised Sections: 2, 16	This is a draft only and not intended to be the final deliverable.
Revisions	12/21/01	All Sections	This is a draft only and not intended to be the final deliverable.
Final Deliverable	1/21/02	Sections 3, 4, 5, 6, 9, 12, 15, 16 and 17	Revisions to incorporate comments from review. This is the final deliverable.

TABLE OF CONTENTS

1. DOCUMENT CONVENTIONS

The following conventions are used throughout this document:

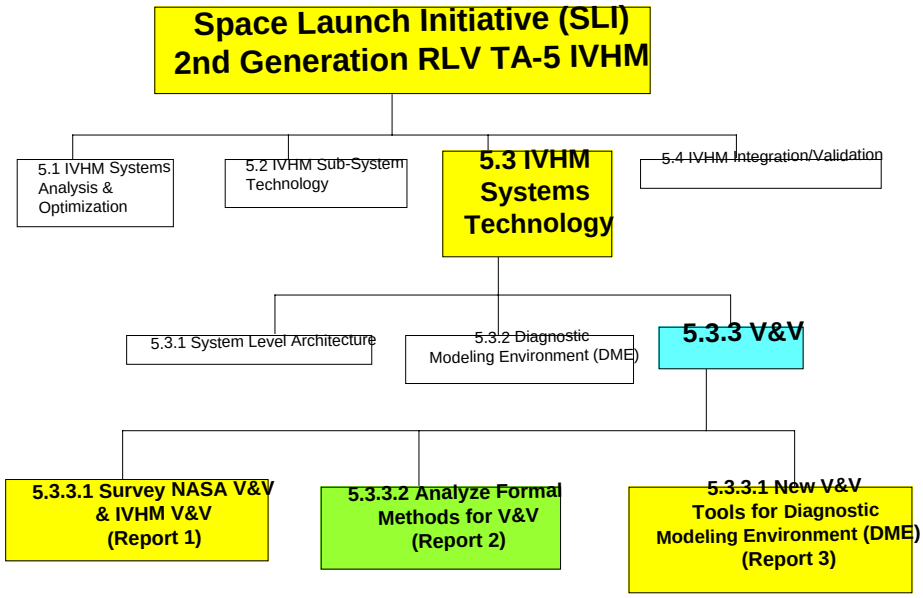
- The term “formal testing” has two meanings. Traditionally, “formal testing” has been used to describe an official test occurring at the end of each life cycle phase and demonstrating that software is ready for intended use. It includes the following:
    - Approved Test Plan and Procedure
    - Quality 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
    - Runtime Monitoring
    - Static Analysis
    - Theorem Proving

  Therefore, to avoid any 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 “Lightweight Formal Methods” used in this report is defined by Martin Feather at Jet Propulsion Lab (JPL) as a special type of runtime monitoring that is relatively easy and speedy to:
    - Acquire information in the proper format for analysis
    - Decide what to analyze
    - Perform analysis
    - Interpret results[21]
- The term “V&V” generally means “Software V&V” 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). In other words:
    - Verification – Build the Product Right
    - Validation – Build the Right Product

2. EXECUTIVE SUMMARY

This report was prepared by the NASA Ames Research Center Automated Software Engineering (ASE) group as the deliverable for Task 5.3.3.2 "Analyze Formal Methods for V&V", highlighted in green on Figure 1: SLI 2nd Generation RLV TA-5 IVHM Project Structure. It is the second of three reports 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 the following:

- Overview of Formal Methods beneficial to V&V of 2nd Generation RLV IVHM
- Description of current use of Formal Methods at NASA and their applicability to 2nd Generation RLV IVHM software
- Guide for incorporating Formal Methods into NASA V&V Standards for certification of airborne software like 2nd Generation RLV IVHM

This report is divided into the following four sections:

- Formal Methods Overview
- Survey – Current NASA Formal Methods
- Formal Methods Applicable to 2nd Generation RLV IVHM
- Incorporating Formal Methods into NASA Standards

Report 2 uses Report 1 as a foundation and builds on that foundation by recommending new Formal Verification processes and methods for V&V of Advanced Software planned for 2nd Generation RLV IVHM.

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.

2.1.1. FORMAL METHODS OVERVIEW

This section provides an overview of Formal Methods including benefits and challenges of each. Specific recommendations for Formal Methods applicable to 2nd Generation RLV IVHM is based on information in this section and discussed in Section 5.

The term "Formal Methods" refers to the use of techniques from logic and discrete mathematics (discrete structures like set theory, automata theory, formal logic as opposed to continuous mathematics like calculus) in the specification, design and construction of computer systems and software. The objective of Formal Methods as a V&V technique is to reduce reliance on human intuition and judgment by providing more objective and repeatable tests.

Formal Methods and their associated benefits and challenges are listed below:

- **Runtime Monitoring** - evaluating code while it runs or scrutinizing the artifacts (event logs, etc) of running code

  Benefits

  - Requires a relatively small incremental effort over traditional testing
  - Combines the ease of testing with the power of Formal Methods
  - Can locate difficult to find error potential for problems that test engineers may not envision

  Challenges

  - Logic-based monitoring can add overhead to the normal execution of programs
  - While detecting difficult to find errors, error pattern runtime analysis can detect problems that do not exist (false positives)
  - Runtime monitoring observes the current program execution, but does not observe all possible runs so coverage is limited to actual program execution

- **Static Analysis** - detects runtime errors and unpredictable code constructs without executing code

  Benefits

  Verification can begin earlier in the Software Life Cycle resulting in early detection/resolution of problems and reduction in development cost

Challenges
The biggest challenge for Static Analysis is generation of false positives. However, while the number of false positives may seem large in some cases, subsequent errors can be the result of an initial or upstream error. Correcting this error can eliminate some false positives. [1]

- **Model Checking** - automated technique for verifying finite state concurrent systems. The model checker evaluates the model by beginning with the initial states and repeatedly applying transitions to reach all possible states.

  Benefits

  - Fast, automated method for exploring all relevant execution paths of non-deterministic systems like 2nd Generation RLV IVHM. This is very important because it is virtually impossible for humans to conceive every test scenario required to verify a non-deterministic system in a plausible time frame for software development.
  - Can backtrack to explore alternative paths from a common intermediate state, avoiding the costly reset between tests required in traditional scenario based testing
  - Detects problems in the early stages of development; thereby greatly reducing overall development costs. [17]

  Challenges

  - Models must be translated into special model checking language (NASA ARC is developing tools to automate this process for Livingstone models)
  - A model checker can run out of memory before exploring the entire program. This is called state-space explosion. (Techniques to mitigate state space explosion are discussed in Section 3.4.)

- **Theorem Proving** – use of logical induction over the execution steps of the program to prove system requirements. In other words, system requirements can be translated into complex mathematical equations and solved by verification experts. Solving these equations proves that the system is accurate.

  Benefits
  Can use the full power of mathematical logic to analyze and prove properties of any design

  Challenges
  Requires significant effort and expertise making Theorem Proving suitable for analysis of small-scale designs by verification experts only.[2]

2.1.2. SURVEY – CURRENT NASA FORMAL METHODS

A survey of current Formal Methods used at NASA was conducted to support planning and analysis for V&V of the 2nd Generation RLV IVHM.

Three programs were selected as being representative of current NASA Formal Methods. They are listed below:

- Formal V&V of Remote Agent (Model Checking)
- Verification of Plan Models (Model Checking)
- RA EXEC Lightweight Formal Methods (Runtime Monitoring)

The following subsections contain a summary of survey results.

2.1.2.1. Formal V&V of Remote Agent (Model Checking)

Formal V&V using the SPIN Model Checker on Remote Agent systems had the following results:

- Original verification (occurred at the beginning of development) found five concurrency errors early in the design cycle that developers acknowledge could not have been found through traditional testing methods
- Quick-response verification performed after a deadlock occurred during the 1999 space mission, resulted in finding a concurrency error. This error was similar to the errors found in the original test showing that Formal Methods testing can improve the safety and reliability of future missions by finding errors that traditional testing methods cannot.[3]

All involved parties regarded the formal methods verification effort as a very successful application of model checking. According to the Remote Agent programming team, the effort had a major impact, locating errors that would probably not have been located otherwise and identifying a major design flaw.

2.1.2.2. Verification of Plan Models (Model Checking) [19]

Research is underway to develop tools and techniques for verifying the Heuristic Scheduling Testbed System (HSTS) Plan Models and the Planning engine. A translation tool has been developed called DDL2UPPAAL for translating HSTS models into UPPAAL (tool for modeling, simulation and verification of real-time systems).

DDL2UPPAAL model translation works well for models of a limited size and complexity. Techniques to verify larger systems appear promising and require additional investigation.

2.1.2.3. RA EXEC Lightweight Formal Methods (Runtime Monitoring)

Twenty anomalies found early in the Software Life Cycle indicate that using Lightweight Formals Methods can improve the accuracy and reduce the cost of software development.

The Lightweight Formal Methods Database Approach provides a straightforward, simple tool for conducting consistency and completeness checks by leveraging the power of a database as a reasoning engine. It provides flexibility to quickly construct new tests and to refine previous tests, as necessary.

Adapting the Lightweight Formal Methods Database Approach to the existing available information was relatively easy and pilot studies indicate that it is feasible to quickly analyze large amounts of data. A more sophisticated user interface could make Lightweight Formal Methods Database Approach easier to use.

2.1.3. FORMAL METHODS APPLICABLE TO 2ND GENERATION RLV IVHM

The purpose of Section 5, "Formal Methods Applicable to 2nd Generation RLV IVHM" is to discuss which Formal Methods are beneficial to the IVHM system planned for the 2nd Generation RLV.

In addition to traditional testing, these Formal Methods are applicable to 2nd Generation RLV IVHM for the following reasons:

- **Runtime Monitoring** - requires a relatively small incremental effort over traditional testing and can locate difficult to find error potential
- **Static Analysis** – can enable verification to begin earlier in the Software Life Cycle resulting in early detection/resolution of problems and possible reduction in development cost
- **Model Checking** - provides a fast, automated method for exploring all relevant execution paths of non-deterministic systems like 2nd Generation RLV IVHM. This is very important for several reasons:

- It is virtually impossible for humans to conceive every test scenario to verify a non-deterministic system in a plausible time frame.
- By verifying all possible execution paths through a system at the same time, the costly reset between tests required in traditional scenario based testing can be avoided.
- Problems can be detected in the early stages of development; thereby reducing overall development costs. [17]

- **Theorem Proving** - applicable to 2nd Generation RLV IVHM, but may call for a sizeable budget because Theorem Proving requires significant effort and expertise. It is more suitable for analysis of small-scale designs by verification experts.[4]

2.1.4. INCORPORATING FORMAL METHODS INTO NASA STANDARDS

This section describes recommendations for incorporating formal methods into the NASA Standards so 2nd Generation RLV IVHM can be certified to fly.

It includes the following:

- Prerequisites for successful integration of formal methods into a program following NASA standards
- Recommendations for where to add formal methods in the Software Life Cycle organized into the following categories:
  - Planning for Formal Methods
  - New V&V Processes
  - New Technical Methods
  - Cost Considerations
- Suggestions of metrics to measure the effectiveness of Formal Methods

3. FORMAL METHODS OVERVIEW

The term "Formal Methods" refers to the use of techniques from formal logic and discrete mathematics in the specification, design and construction of computer systems and software. The word "formal" derives from Formal Logic and means "pertaining to the structural relations between elements".[38]

The objective of Formal Methods is to reduce reliance on human intuition and judgment by replacing the subjectivity of informal and quasi-formal review processes (traditional testing) with a more objective and repeatable exercise. [38]

Formal Methods have been categorized by levels of formalization that correlates to the effort versus assurance provided by the Formal Verification. For example, runtime monitoring requires less effort than Model Checking, but also provides less assurance of system accuracy.[5]

Figure 2: V&V Effort versus Assurance shows the effort versus benefit of the most commonly used V&V techniques. As shown, the spectrum of activities ranges from testing to runtime monitoring to Formal Methods like static analysis, model checking and theorem proving.

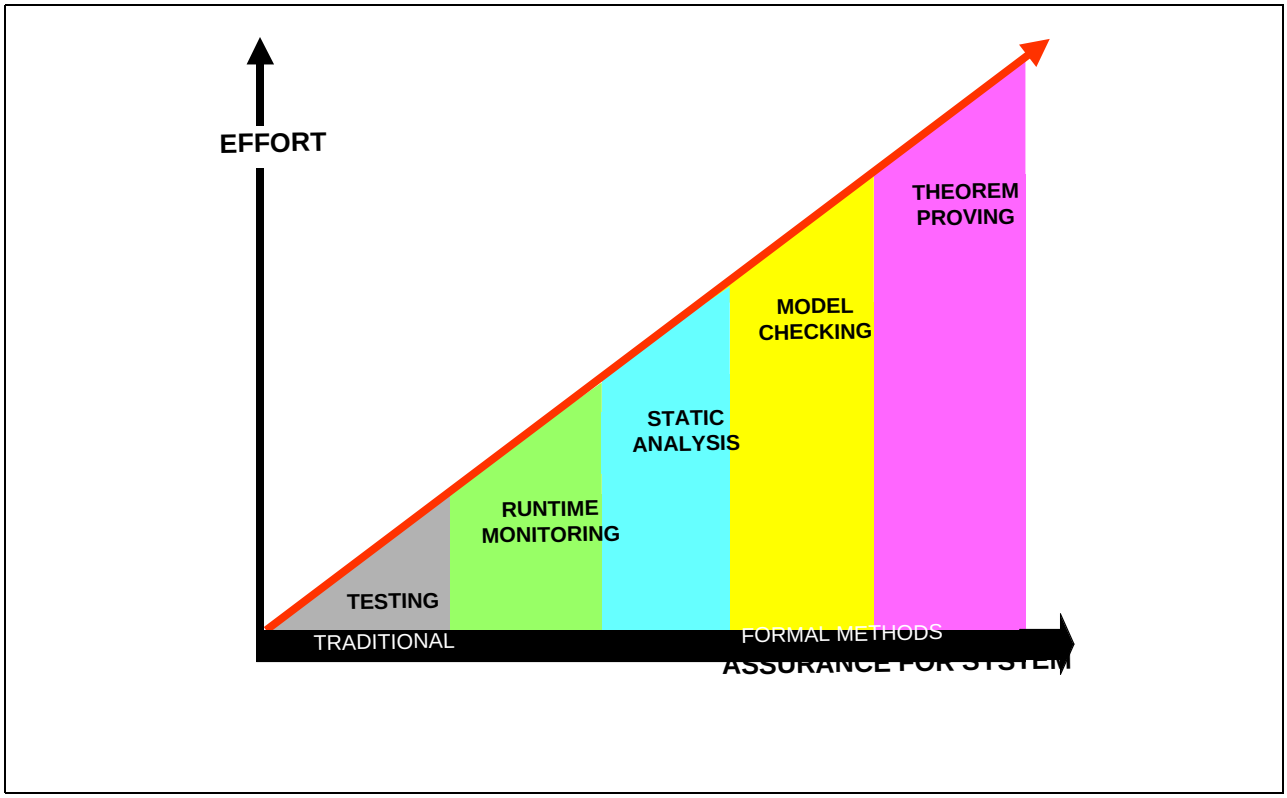


Figure 2: V&V Effort versus Assurance [5]

Note: *Report 3, New V&V Tools for Diagnostic Modeling Environment (DME), contains a detailed description of new tools developed at NASA ARC to automate model checking. Additionally, research is underway in the Automated Software Engineering (ASE) group at NASA ARC to further automate Runtime Monitoring and Static Analysis. This research is discussed in Section 3.2 and 3.3.*

Each V&V activity shown in Figure 2: V&V Effort versus Assurance is explained in the following sections.

3.1. Testing

Traditional V&V is based on test scenarios developed by engineers. Normally, these scenarios are activated by human operators who attach a test harness to a control channel and build tests that exercise each of the scenarios on various fidelity test beds from simulators to actual hardware.

Benefits

Traditional testing techniques are well known and have been used successfully in the past for V&V of traditional software.

Challenges

The design and maintenance of test suites is a difficult and expensive process. It requires a good understanding of the system to be tested to ensure that a maximum number of different situations are covered using a minimum number of test cases. [17]

Running these tests is also a time-consuming task because the test-bed must be initialized before each test and regression testing must be performed. In the development of complex systems, it is quite common that testing the software actually requires more resources than developing it. [17]

While verification of traditional software is difficult, verification of Advanced Systems like 2nd Generation RLV IVHM is even more challenging for reasons listed in Table 1: Reasons for Difficulty of V&V of Advanced Systems

Table 1: Reasons for Difficulty of V&V of Advanced Systems

Reason	Explanation
Hard to test	Autonomous systems like Deep Space Remote Agent close control loops and arbitrate resources on-board, making it more difficult to plug in test harnesses and run detailed tests that drive the system through a desired behavior
Large number of tests	The range of situations to be tested is much larger. An autonomous system implicitly incorporates the response scenario to any combination of events that might occur. Its reaction depends on the current configuration of the system or its past history (in the case of adaptive systems). All these factors multiply exponentially with the size of the system making it virtually impossible for a team of humans to accurately develop a test suite to rigorously exercise an Advanced System like 2nd Generation RLV IVHM in a timely manner.
Concurrency	An autonomous controller is typically multi-threaded meaning that different concurrent parts of the controller execute together. As different parts are scheduled in different ways, the controller may react in different ways to the same stimuli.[17] Therefore, a successful test does not guarantee that the system will behave correctly because the same input sequence can lead to different execution sequences. Another test could fail due to uncontrollable changes of circumstances. This is a well-known issue in designing concurrent systems. [17] The Deep Space One (DS1) Remote Agent (RA) experiment provided a striking example. After a year of rigorous testing, RA was put in control of DS1 on May 17, 1999. A few hours later, RA had to be stopped after a deadlock was detected. Analysis revealed the deadlock was caused by a highly unlikely race condition between two concurrent threads in RA's executive. The scheduling conditions that caused the problem to manifest never happened during testing. [17] But a similar problem was caught by a Formal Methods experiment described in Section 4.1.

3.2. Runtime Monitoring

Runtime Monitoring means evaluating code while it runs or scrutinizing the artifacts (event logs, etc) of running code.

Two approaches to runtime monitoring are described below:

- Lightweight Formal Methods – Database Approach
- Monitoring Java Programs with Java PathExplorer (tool developed at NASA ARC)

3.2.1. Lightweight Formal Methods – Database Approach

In this report, Lightweight Formal Methods is a term used by Martin Feather at Jet Propulsion Lab (JPL) to describe a specific type of runtime analysis that is relatively easy and speedy to:

- Acquire information in the proper format for analysis
- Decide what to analyze
- Perform analysis
- Interpret results[21]

One Lightweight Formal Method that meets the above criteria is the Database Approach. The Database Approach provides a method for storing program logs, interface diagrams or other pertinent data in a database then querying this database for specific information.

The Database Approach analyzes code artifacts, like logs, rather than monitoring an actual program execution. Steps to use the Database Approach are listed below:

1. Design V&V objectives
2. Select a database tool
3. Create a schema to hold information to be analyzed
4. Load data (parse, if necessary)
5. Develop database queries based on V&V objectives
    a. Positive Approach: query result set includes a list of expected results that can be verified
    b. Negative Approach: an empty query result set indicates a successful test. In other words, a query is written to find anomalies. If no anomalies exist, then the result set is empty and the item being tested works properly.
6. Analyze results of queries [21]

Two case studies using Lightweight Formal Methods are described in Section 4.2.

Benefits
The Database Approach leverages the power of a database as the underlying reasoning engine. This approach provides more coverage than simulation-based testing and requires a relatively small incremental effort over traditional testing.

Challenges
Coverage is limited to a single trace through the system rather than an exhaustive search of all possible system executions.

3.2.2. Monitoring Java Programs with Java PathExplorer[6]

Java PathExplorer (JPaX) is a tool for monitoring systems while they execute. Test engineers can use JPaX to automatically instrument code and observe the inner workings of software while it runs. This technique provides information about safety critical systems like 2nd Generation RLV IVHM. It can be used during development to provide more robust verification. It can also be used during operations to further optimize systems as they mature.

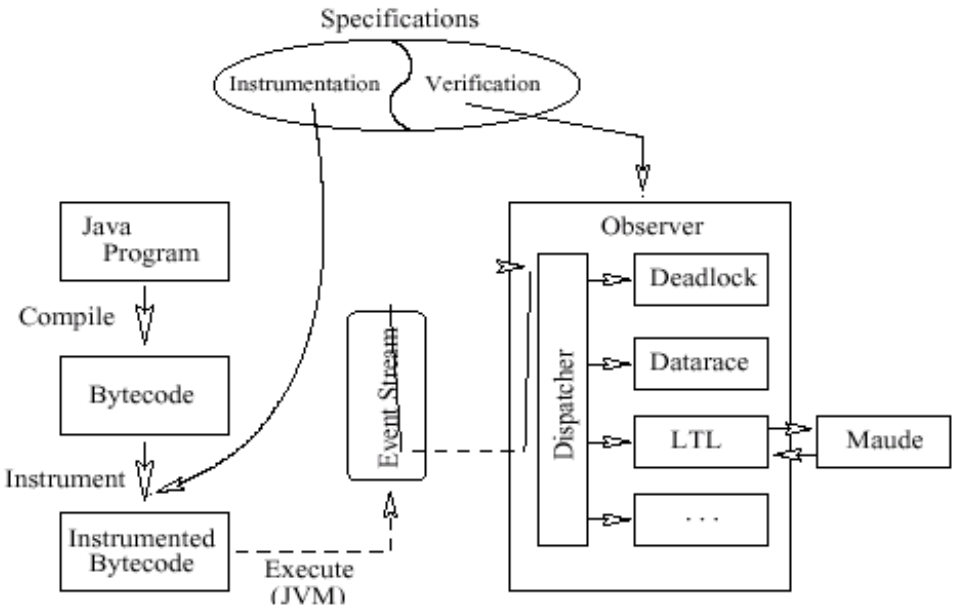


Figure 3: Overview of Java PathExplorer (JPaX)

JPaX consists of the three modules described below:

1. Instrumentation – performs a script-driven automated instrumentation of the program to be verified

2. Observer – performs two kinds of verification:
   - Logic-based monitoring – checking execution events against a user-provided requirement specification. These specifications are defined in Maude, a modularized specification and verification system. See Appendix E for more information about Maude.

     JPaX supports linear temporal logic (LTL), both future time and past time. LTL is explained in Appendix F. Future time LTL provides execution traces as models making it convenient for program monitoring. Past time is useful for verification of safety properties.

   - Error pattern analysis – explores an execution trace and detects potential problems such as error-prone programming techniques like locking practices that may lead to data races and/or deadlocks. The important and appealing aspect of error pattern analysis algorithms is that they find error potentials even in the case where errors do not explicitly occur in the examined execution trace. Therefore, an obscure error that test engineers did not conceive could be detected. The trade off is error pattern analysis may sometimes find errors that do not exist.

     JPaX contains two algorithms focusing on concurrency errors:

- "Eraser" data race analysis algorithm – a data race occurs when two or more concurrent threads access a shared variable, at least one access is a `write,` and the threads use no explicit mechanism to prevent the accesses from being simultaneous. The Eraser keeps track of thread locks and variables to find data race conditions.
- Deadlock analysis algorithm – deadlocks occur when multiple threads take locks in different order. For example, a classic deadlock condition occurs when:
  - Thread A acquires Lock 1 while Thread B acquires Lock 2
  - Thread A retains Lock 1 and acquires Lock 2 while Thread B retains Lock 2 and acquires Lock 1

  JPaX keeps track of locks during program execution to find deadlock errors.

3. Interconnection – receives information about potential errors and transmits them to the observation module

To use JPaX, a test engineer inputs:

- Java program byte code
- Specification script defining what kind of error pattern detection algorithms should be activated and what kind of logic based monitoring should be performed

JPaX provides either null (if no errors are found) output or a set of warnings. A test engineer can observe events and check them against the following user-provided data:

- High-level requirements like temporal logic formulae
- Low-level, error-detection procedures like concurrency related problems such as deadlock and data race algorithms.

The following article provides an in-depth discussion of JPaX:

- Klaus Havelund, Grigore Rosu. *Monitoring Java Programs with Java PathExplorer.* To be published in Electronic Notes in Theoretical Computer Science.

C++ Experiment
Because JPaX is modular, the Java instrumentation module can be replaced with a C++ module to monitor C++ code. An experiment was conducted with the NASA Ames Robotic group on about 90,000 lines of C++ code.

The Java instrumentation module was replaced with a C++ instrumentation module and the experiment checked for deadlocks. It located a deadlock potential that had not been previously detected during other testing.

Benefits
JPaX combines the ease of testing with the power of Formal Methods so engineers can avoid some of the pitfalls of the ad hoc nature of testing and the complexity of theorem proving and model checking.

It can also find potential errors, even in the case where errors do not explicitly occur, in the examined execution trace.

Challenges
Logic-based monitoring can add overhead to the normal execution of programs. While detecting difficult to find errors, error pattern runtime analysis can detect problems that do not exist (false positives).

3.3. Static Analysis

Static Analysis detects potential runtime errors and other undesirable code constructs by analyzing the program structure at compile-time, without executing the software. So instead of testing, crashing, debugging and testing again, static analysis provides a list of errors that can be fixed during development - thereby, reducing development costs. Common static analysis techniques include data flow analysis, control flow analysis, abstract interpretation and type and effects analysis[7].

Type checking, as performed by most modern programming language compilers, is a simple form of static analysis; however, static analysis can also detect the following types of errors:

- Attempt to read a non-initialized variable – read access to non-initialized data may cause non-determinism. Static Analysis tools locate code sections using data that is not initialized.
- Access conflicts for unprotected shared data
- Referencing through null or out-of-bound pointers
- Out of bound arrays - out of bound array errors occur when an index goes outside the range of an array. Static Analysis checks whether the loop incrementing the index can exceed the array size.
- Division by zero – Static Analysis can check that a division equation is properly coded with `if` statements to prevent the denominator from equaling zero. It can also provide a list of possible denominator variables to check the equation.
- Invalid arithmetic operations (square root of negative number) – Arithmetical exceptions caused by procedural entities like modulo computation, square root and logarithm can be checked using Static Analysis.
- Overflow, underflow of arithmetic operations for integers and floating-point numbers – overflow and underflow occur in numerical computation when a result is not compatible with the variable that stores it. Therefore, it cannot be represented in memory. Static Analysis locates and reports these problems.
- Unreachable (dead) code – Static Analysis can locate and report codes segments that are never executed. For example: `if` statement never executed because its condition is never met.
- Illegal type conversion – occurs when a result does not match its assignment
- Unpredictable behavior of multi-threaded applications with shared data. Depending upon the order in which threads read and update shared data, different results can occur for the same input.[8]

Benefits

Because Static Analysis verifies code before runtime, verification can begin earlier in the Software Life Cycle. Finding anomalies before testing and debugging reduces development effort and cost. Because it does not need to explore the actual executions of the system, static analysis can be applied to fairly large programs and still provide formal guarantees that the code is free of the specified defects.

Challenges

The biggest challenge for Static Analysis is generation of false positives sometimes due to over-approximation. For example, assume that the analysis only tracks the sign of some integer variables. If a positive and a negative value are added, the algorithm cannot tell the sign of the result and will consider both alternatives to err on the safe side. One of them may lead to error that corresponds to no actual feasible execution of the real program.

While the number of false positives may seem large in some cases, subsequent errors can be the result of an initial or upstream error. Correcting this error can eliminate some false positives.

3.4. Model Checking

Model Checking is an automated technique for exhaustively verifying finite state concurrent systems[9]. It has been successfully applied to analyze parts of Deep Space One Remote Agent. In order to explain Model Checking, the following terms must be defined:

- Formal Model – computer model of system
- State – snapshot of the system that captures values of variables at a particular instant in time
- Transition – the change described by the state before an action occurs and the state after the action occurs

To use a model checker, an engineer must create a model of the system. This model represents valid states and transitions of the system.

The model checker evaluates the model beginning with the initial states and repeatedly applying transitions to reach all possible states.[17] If a property violation occurs, the model checker reports the error via an execution trace called a counterexample. Counterexamples show where the violation occurred.

Model checkers verify properties of dynamic operations described using the Temporal Logic formalism. Two kinds of Temporal Logic are described in Appendix F:

- Linear Temporal Logic (LTL) – time is described in a linear fashion with no branching
- Computational Tree Logic (CTL) – time is described in a branching fashion

Three Model Checking tools are mentioned in this report and have been used at NASA:

- SPIN – an explicit model checker that enumerates all individual states to be verified. This approach is limited by the size of the state space of the system. Although SPIN provides many optimizations to cover very large state spaces (millions of states and more), it is unlikely to scale well for very complex models.
- SMV – a symbolic model checker that uses efficient data structures (binary decision diagrams, or BDDs) to represent and process sets of states in a single operation. The symbolic processing allows SMV to explore much larger state spaces than explicit state tools such as SPIN. However, SMV is still limited by the complexity of the generated BDD structures, which can vary wildly and are hard to optimize.
- UPPAAL - a toolbox for validation and verification of real-time systems described as networks of timed automata

Tools, called MPL2SMV and JMPL2SMV, are being developed to automate SMV model creation for Livingstone models making verification of these models fast and accurate. Report 3, *New V&V Tools for Diagnostic Modeling Environment (DME),* includes details about these tools.

Benefits

Model Checkers are particularly well suited for exploring the relevant execution paths of concurrent systems (with multiple processes running in parallel) like 2nd Generation RLV IVHM. Instead of executing the real code, a model checker executes an abstract model in a highly efficient way. [17] This is very important because it is virtually impossible for humans to conceive every test scenario required to verify a non-deterministic system in a plausible time frame for software development.

Furthermore, the model checker can backtrack to explore alternative paths from a common intermediate state, avoiding the costly reset between tests required in traditional scenario based testing. It stores and compares states to detect those already explored, thus exploring all states exactly once even in the case of looping executions. [17]

Finally, a model checker controls the scheduling of concurrent components of the model and will therefore explore all possible execution sequences even for the same input sequence. It can be applied at an earlier stage in the design long before a testable product is available as shown in Figure 4: Repair Cost of Design Defects.

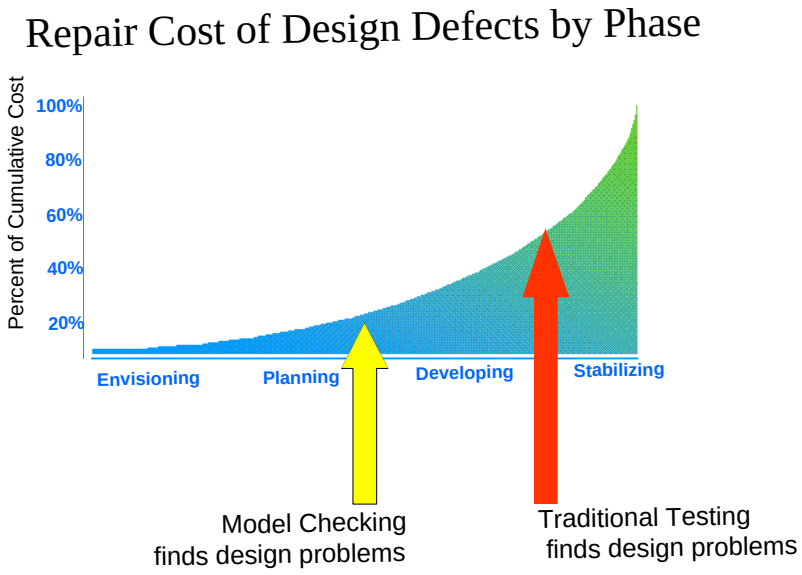


Figure 4: Repair Cost of Design Defects[10,17]

Challenges

There are two challenges associated with model checking:

- Previously, model development required a manual effort to translate a model into a special model checking language like PROMELA (for SPIN) or SMV. Recently, various tools have been developed to apply model checking to application modeling and programming languages, either directly or through automated translation to the model checking language. These tools significantly reduce model development time. In particular, NASA ARC tools to automate translation of Livingstone models into the SMV language are discussed in detail in Report 3, *New V&V Tools for Diagnostic Modeling Environment (DME)*.
- State space explosion – Because of the way software components interact with each other and because data structures can have different values, it is common for a model checker to run out of memory before exploring the entire state space[11].

The following tools/techniques mitigate state space explosion:

> 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. They allow efficient computations of operations needed for model checking.[9]

A BDD is a graph structure used to represent a Boolean function, that is, a function f over Boolean variables $b_{1,\ldots},b_n$ with Boolean result f ($b_{1,\ldots},b_n$). 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* V (*b* /\ *c*) is shown in the following figure:

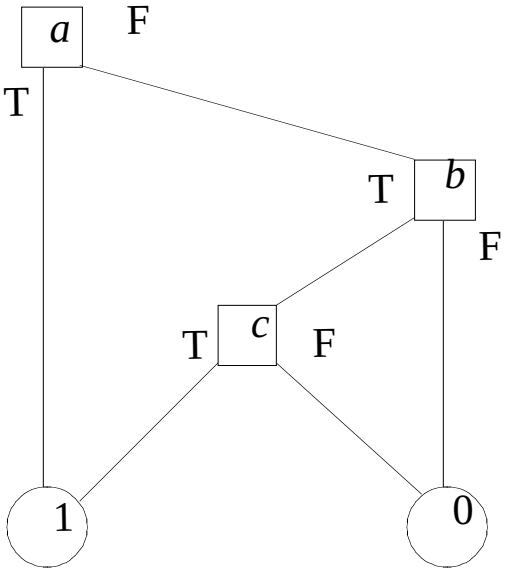


Figure 5: BDD for *a* V (*b* /\ *c*)[39]

The order of the variables is important when developing a BDD because different variable orders can generate a simpler or more complex graph for the same state space. The BDD will also differ depending upon the way the calculation is expressed.

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

- Partial Order Reduction is another technique used to reduce state space size. Verifying software presents special challenges for model checking because software tends to be less structured than hardware and concurrent software is usually asynchronous (activities taken by different processes are performed independently); therefore, the state space for software can be very large due to all the orderings of independent actions.

  Partial Order Reduction decreases the state space by exploiting the independence of concurrently executed events. Two events are considered independent when executing them in either order results in the same global state.

  A common model for representing concurrent software is the interleaving model where all of the events in a single execution are arranged in a linear order called an interleaving sequence. Concurrently executed events appear arbitrarily ordered with respect to one another. However,

most logics, including temporal logic, can distinguish between interleaving sequences in which two independent events are executed in different orders. Because of this, all possible interleavings of such events are normally checked resulting in a large state space.

Partial order reduction makes it possible to decrease the number of interleaving sequences and therefore, reduce the explored state space. When the two interleaving sequences differ only in the order in which concurrently executed, and the permuted events do not affect the verified property, it is sufficient to check one of them.

Compositional Reasoning - The purpose of compositional reasoning is to shift the burden of verification from the global level to the component level, so that global properties are established in terms of verified component properties. As a simple example, consider a communication protocol consisting of a transmitter and a receiver, exchanging messages over some communication network. To show that the data is transmitted correctly from the sender to the receiver, compositional reasoning could be used as follows:

- First, check that data is correctly transferred from the transmitter to the network
- Second, verify that data is properly transferred from one end of the network to another
- Finally, verify that data is correctly transferred from the network to the receiver[12]

These properties may be checked using only the specifications of the transmitter, network and receiver respectively, possibly introducing assumptions that each component makes on the other's behavior. Thus, having verified each part of the system, it is apparent that the complete system has also been verified.[9]

In theory, this type of approach provides a scalable solution to software verification for industrial-size designs.[13] However, compositional reasoning involves a few challenges:

- Problems may arise when a subsystem cannot satisfy a local property without system context. This is called the "assume-guarantee" paradigm. In this style of reasoning, component properties can only be guaranteed given assumptions about the component environment. However, by appropriately combining the set of assumed and guaranteed properties of system components, it is possible to establish system correctness without constructing the system global transition graph.
- Property decomposition is not a straightforward task. Proving local properties typically requires expert user-guidance to add context constraints (assumptions) to the behavior of a component. These need to be detailed enough to establish the desired property, but not too detailed to cause state-explosion.[12]

- Abstraction can also reduce state space size by reducing model complexity. Abstraction is based on the observation that the specifications of systems include data paths usually involving fairly simple relationships among the data values in the system. Mapping between the actual data values in the system and a small set of abstract values reduces model complexity. By extending this type of mapping to states and transitions, it is possible to produce an abstract version of the system under consideration. The abstract system is often much smaller than the actual system and as a result it is usually much simpler to verify properties at the abstract level.

  Two different abstraction techniques are described below:

  - Cone of Influence Reduction – decreases state space by eliminating variables that do not influence the variables in the specification. Therefore, checked properties are preserved but the size of the model is smaller.
  - Data Abstraction – involves mapping between actual system data and a small set of abstract data values, then extending this mapping to states and transitions in order to obtain an abstract system that simulates the original system but is much smaller. [9]

It is generally not possible to have an exact abstraction of all operations performed on abstract variables, so abstraction algorithms have to conservatively over-approximate the concrete state space covered by the abstract model, possibly leading to false positives. This is the same phenomenon as described for static analysis in Section **Error! Reference source not found.**.

- Symmetry reduction – some systems, like a network of many indistinguishable processes, exhibit significant symmetry. For example: a mutual exclusion protocol where two processes p_1 and p_2 are symmetric, consider the states:

  ($\{p_1, C_2\}$, turn = 2) and ($\{p_2, C_1\}$, turn = 1) i.e. process 2 critical and process 1 trying, or the reverse.

  These states are symmetric by swapping the index values 1 and 2.

  Facts, like those described above, can be used to reduce system models. Having symmetry in a system implies the existence of nontrivial permutation group that preserves the state transition graph. Such a group can be used to define an equivalence relation on the state space of the system and avoid re-exploring symmetrical states. The reduce model can be used to simplify the verification of properties of the original model expressed by a temporal logic formula.[9]

3.4.1. Recent Trends in Model Checking

Recent trends in Model Checking are described below:

- Model checking of executable code - instrumentation of executable code (to control branching and allow backtracking to extract and compare states) provides a method for checking code without constructing a model. Tools using this technique include VeriSoft, Java PathFinder and Livingstone PathFinder. Livingstone PathFinder is described in Report 3, *New V&V Tools for Diagnostic Modeling Environment (DME).*

  Verisoft is a tool for systematically exploring the state spaces of system composed of several concurrent processes executing arbitrary code written in full-fledged programming languages like C or C++. The state space of a concurrent system is a directed graph that represents the combined behavior of all concurrent components in the system. By exploring its state space, VeriSoft can automatically detect coordination problems between the processes of a concurrent system.

  VeriSoft controls and observes the execution of all the concurrent processes of the system and can reinitialize their executions. It can automatically detect coordination problems (deadlocks, divergences…) between the concurrent processes of a system and check for assertion violations. Since states of programs written in programming languages can be very complex because of pointers, dynamic memory allocation, large data structures of various shapes , recursion…) VeriSoft does not attempt to compute any representation for the reachable states of the system being analyzed and hence performs a systematic state space exploration without storing any intermediate states in memory.

  VeriSoft eliminates one major obstacle to a wider use of model checking techniques, namely the need to build a model of the system. However, it does not store any states in memory and cannot detect cycles in state space being explored so it is restricted to checking safety properties. This may not be troublesome since the main goal is to be able to systematically and efficiently search a meaningful portion of the state space within a reasonable amount of time in order to detect unexpected behaviors of the system.[14]

- Temporal Queries – a temporal query is a temporal logic formula in which a placeholder "**?**" appears exactly once. Solving the query amounts to finding some canonical condition that, when substituted for the placeholder, makes the whole formula true. For example, the question “If I

have a fault *x*, what does it take to recover?" may be phrased as the (informal) temporal query "**always** (fault x => (**?** => **eventually** recover))".

Not only, do temporal queries expand the debugging function, they provide a model-understanding function. By using temporal queries, the model checker becomes more than a verification tool. A user can hypothesize about a behavior of the system, express the behavior as a temporal-logic formula and use the model checker to validate the hypothesis.

Temporal queries are a very recent and still largely unexplored idea[15]. While research is still required to refine the query-mechanism, temporal queries may be a helpful addition to the model checker of the future.

- Bounded model checking – Bounded model checking with satisfiability (SAT) solving is a new variant of symbolic model checking that can explore large state spaces with reduced memory requirements. It searches for counterexamples to an assertion using up to a fixed number of transitions and can be applied to both safety and liveness properties, but application to safety properties shows the most promise.

  For example, a simple, yet very important type of safety property is an invariant, a property that must hold in all reachable states. Obviously, if a sequence of states can be found that begins at an initial state and ends in a state where the supposed invariant is false, that property is not an invariant.

  Although SAT solving requires exponential time in the worst case, early experimental results suggest that counterexamples can be found with remarkable efficiency with bounded model checking, on designs that would be difficult for BDD based model checking**Error! Bookmark not defined.**. Also, when compared to BDD-based model checking, bounded model checking seems to require less user manipulation like finding a suitable manual ordering for BDD variables. While by-hand adjustments could be necessary in bounded model checking, they are not generally required.

  The robustness and the capacity increase of bounded model checking make it attractive for industrial use. However, the disadvantages of bounded model checking are that the method lacks completeness and the types of properties that can currently be checked are very limited. Additionally, it has not been shown that this method can consistently find long counterexamples or witnesses. However some of these drawbacks are being addressed by industry experts.

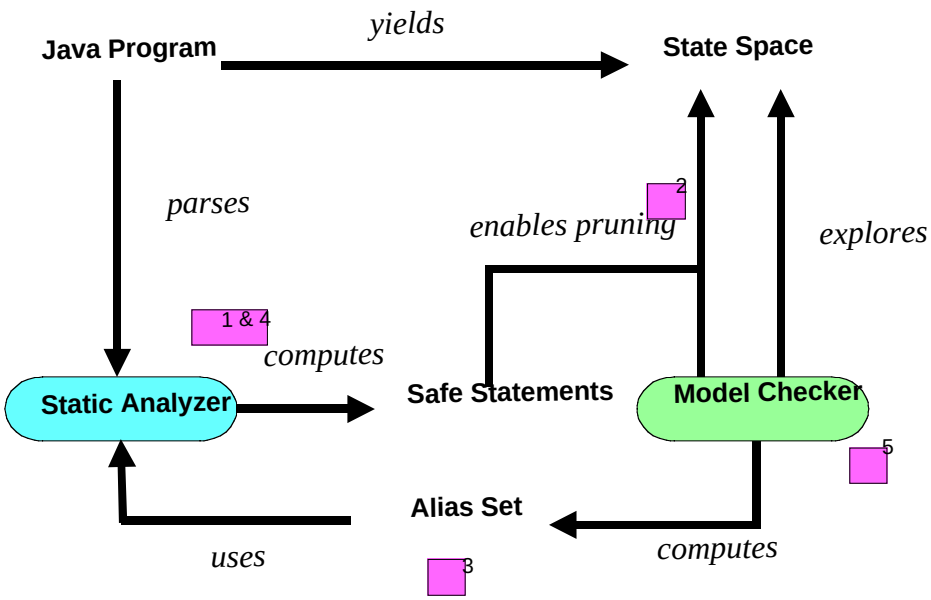
  Essentially, there are two steps in bounded model checking:

  1. The sequential behavior of a transition system over a finite interval is encoded as a propositional formula
  2. The formula from step one is given to a propositional decision procedure, i.e., a satisfiability solver, to either obtain a satisfying assignment or to prove there is none.

  Each satisfying assignment that is found can be decoded into a state sequence which reaches states of interest. In bounded model checking only finite length sequences are explored. For more information see Bounded Model Checking Using Satisfiability Solving[16]

3.5. Hybrid Approach: Combining Static Analysis & Model Checking

Combining a static analyzer with a model checker permits taking advantage of one's strengths to cancel the other's weakness. The following figure shows the Iterative Partial Order Reduction Process for combining the static analyzer and model checker.



Iterative Partial Order Reduction Process:

1 The static analyzer computes partial order information

2 The model checker uses the partial order information to reduce the state space

3 During exploration, the model checker computes aliasing information that it gives to the static analyzer

4 The static analyzer uses the aliasing information to refine its analysis resulting in new partial order reduction information that is fed to the model checker

5 The model checker updates its partial order reduction and the cycle repeats itself

Figure 6: Iterative Partial Order Reduction Process[11]

Benefits
Combining Static Analysis and Model Checking takes advantage of the strengths of each to offset their weaknesses. A static analyzer has no knowledge of the values of variables at run time. It is subject to approximations inherent in the traditional dataflow framework (widening or k-limiting) to identify possible runtime variables.

Therefore, one of the biggest weaknesses of the static analyzer is the precision of its results are directly conditioned by the precision of the aliasing analysis it performs. Unfortunately, most practical alias analysis algorithms are imprecise.

On the other hand, the strength of the model checker is that it can compute precise aliases since the state exploration process is equivalent to executing the program under all possible interleavings or executions.

Challenges
The main challenge when combining Static Analysis and Model Checking is implementing this hybrid approach in a way that does not overwhelm the Model Checker by providing too many safe statements.

3.6. Theorem Proving

Theorem Proving is the use of logical induction to prove system requirements. [32] Appendix A includes an example of Theorem Proving.

The *Formal Methods Specification and Verification Guidebook for Software and Computer systems Volume 1: Planning and Technology Insertion* contains case studies describing the application of Theorem Proving for the following:

- Space Shuttle flight control (Jet Select) requirements
- Space Shuttle on-orbit Digital Auto Pilot (DAP) requirements
- Space Shuttle Change Request (CR) concerning integration of Global Positioning System (GPS)
- Space Shuttle Three Engine Out CR. The Three Engine Out task is executed each cycle during powered flight until either a contingency abort maneuver is required or progress along the powered flight trajectory is sufficient to preclude a contingency abort even if three main engines fail.
- International Space Station Alpha (ISSA) – requirements for Failure, Detection, Isolation and Recovery (FDIR)
- Cassini CDS Fault Protection (FP) software requirements [32]

Benefits
In principle, Theorem Provers can use the full power of mathematical logic to analyze and prove properties of any design. For example, the Prototype Verification System (PVS) system has been applied to several NASA applications described in the above-mentioned case studies.[32] Appendix B contains more information about PVS.

Challenges
Theorem Provers require significant effort and expertise making them suitable for analysis of small-scale designs by verification experts only.[17]

3.7. Summary

The term “Formal Methods” refers to the use of techniques from logic and discrete mathematics in the specification, design and construction of computer systems and software. The objective of Formal Methods as a V&V technique is to reduce reliance on human intuition and judgment by providing more objective and repeatable tests.

Formal Methods have been categorized by levels of formalization that correlates to the effort versus assurance provided by the Formal verification. For example, runtime monitoring requires less effort than Model Checking, but also provides less assurance of system accuracy.

Formal Methods are listed below from least effort to most effort required:

- **Runtime Monitoring** - evaluating code while it runs or scrutinizing the artifacts (event logs, etc) of running code
- **Static Analysis** - detects runtime errors and unpredictable code constructs without executing the software so it can be used along with the compiler during development
- **Model Checking** - automated technique for verifying finite state concurrent systems. The model checker evaluates the model by beginning with the initial states and repeatedly applying transitions to reach all possible states.
- **Theorem Proving** - builds a computer-supported proof of a requirement by logical induction over a formal specification.

Traditional testing has serious limitations for V&V of safety-critical, Advanced Systems like 2nd Generation RLV IVHM. Formal Methods provide more assurance of accurate system design.

In the past, Formal Methods required significant expertise and effort; however, tools are under construction at NASA to reduce this effort by automating the process. These tools will be discussed in Report 3, *New V&V Tools for Diagnostic Modeling Environment (DME).*

4. SURVEY – CURRENT NASA FORMAL METHODS

The purpose of the Survey – Current NASA Formal Methods is to provide actual examples of Formal Validation and Verification (V&V) processes used at NASA.

Three recent or ongoing NASA programs were selected as being relevant to IVHM of 2nd Generation RLV. They are listed below and described in subsequent sections of this report:

- Formal V&V of Remote Agent (Model Checking)
- Verification of Plan Models (Model Checking)
- RA EXEC Lightweight Formal Methods (Runtime Monitoring)

A formal survey was developed consisting of the questions listed in Table 2 – Survey Questions below. These questions were derived from the following documents, as well as previous Formal V&V experience:

- *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion*[32]
- *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume II: A Practitioner's Companion*[38]

Table 2 – Survey Questions

Number	Question
1	Explain why Formal V&V was necessary
2	Note program maturity (For example: software flew, earthbound experiment etc)
3	Describe program software subject to Formal V&V
4	Note organization performing Formal V&V
4	Explain the Formal V&V efforts
5	Describe the success of Formal V&V

Answers to survey questions were obtained from reading project documents and NASA V&V guidelines and interviewing key project team members. References to project documents are listed in the Reference Section in numerical order as cited in the report. Special thanks to the individuals listed in Table 3 - Individuals Participating in Survey or Overview of Formal Methods for their time in answering questions and providing project documents:

Table 3 - Individuals Participating in Survey or Overview of Formal Methods

Program	Individual	Location
Formal V&V of Remote Agent (Model Checking)	Klaus Havelund	NASA ARC
RA EXEC Lightweight Formal Methods (Runtime Monitoring)	Martin Feather	NASA JPL
Verification of Plan Models (Model Checking)	Lina Khatib	NASA ARC
Hybrid Approach: Combining Static Analysis & Model Checking	Guillaume Brat	NASA ARC

4.1. Formal V&V of Remote Agent (Model Checking)

Two Formal Verification experiments were conducted on Deep Space One Remote Agent: one before flight and another after a deadlock occurred during flight.

4.1.1. Why Formal V&V

With the increasing power of flight-qualified microprocessors, NASA is experimenting with a new generation of non-deterministic flight software that provides enhanced mission capabilities. A prime example is the Deep Space One Remote Agent (RA) autonomous spacecraft controller. RA is a complex concurrent software system employing several automated reasoning engines using artificial intelligence technology. The verification of this complex software is critical to its acceptance by NASA mission managers.

4.1.2. Program Maturity

RA software flew DS1 between May 17 and May 21, 1999 [18]

4.1.3. Program Software Subject to Formal V&V

RA is unique and differs from traditional spacecraft commanding because ground operators can communicate with it using goals like "during the next week take pictures of the following asteroids and thrust 90% of the time". It is a model-based system composed of the three AI technologies listed below:

- Planner-Scheduler - generates plans that RA uses to control the spacecraft
- Smart Executive (EXEC) - The EXEC is goal oriented, for example: EXEC's goal: "keep device A on from time X to time Y" will result in EXEC issuing a command to turn device A on if EXEC detected that device A was off between time X and Y.[3] The Smart Executive or EXEC is responsible for the following:
    - Requesting and executing plans from the planner
    - Requesting/executing failure recoveries from MIR
    - Executing goals and commands from human operators
    - Managing system resources
    - Configuring system devices
    - System-level fault protection
    - Achieving and maintaining safe-modes as necessary
- Livingstone or MIR (Mode Identification and Reconfiguration) observes the EXEC issuing commands, receives state information from the spacecraft and uses a model-based inference to deduce the state of the spacecraft and provide feedback to EXEC. It also serves as a recovery expert by taking EXEC constraints and using declarative models to recommend a single recovery action to EXEC

Formal verification was conducted on the EXEC.

4.1.4. Organization Performing Formal V&V

NASA AMES Research Center Autonomy Systems Engineering Group (ASE)

4.1.5. Formal V&V[3]

Two different Formal Verification efforts were conducted on RA, before and after flight, using different technologies in very different contexts.

Formal Methods – Before Flight

In April-May, 1997 (while RA was in the developmental stages) a model was created for the RA EXEC using the SPIN model checker. SPIN is a tool for analyzing the correctness of finite state concurrent

systems. (***Note:*** *Appendix C contains more information about SPIN.)* To use SPIN, a concurrent software system must be modeled using the PROMELA modeling language. The SPIN Model Checker examines all program behaviors to decide whether the PROMELA model satisfies the stated properties. If a property is not satisfied, an error trace is generated to show the sequence of executed statements from the initial state to the state that violates the property.

The RA modeling effort took about 12 person-weeks during a six calendar week period. The verification effort took one week. Between 3,000 and 200,000 states were explored using between 2-7 MB of memory and running between 0.5 and 20 seconds.

This test resulted in discovery of the five errors listed below:

- One error breaking the release property (defined as *"a task releases all of its locks before it terminates"*)
- Three errors breaking the abort property (defined as *"if an inconsistency occurs between the database and an entry in the lock table, then all tasks that rely on the lock will be terminated, either by themselves or by the daemon in terms of an abort"*)
- One non-serious efficiency problem where code was executed twice rather than once

Four of these errors were classic concurrency errors because they arise due to processes interleaving in unexpected ways. One error was similar to the error that deadlocked DS1 in flight. That error caused the abort property to be violated. The SPIN error trace demonstrated the following situation:

> The daemon is prompted to perform a check of the lock table. It finds everything consistent and checks the event counters to see whether there have been any new events while it was running. If not, the daemon decides to call `wait-for-events`. However, at this point an inconsistency is introduced and a signal sent by the environment causing the event counter for the database event to be increased. This is not detected by the daemon since it has already made the decision to wait. The daemon waits and the inconsistency is not discovered.

Proposed solution to the problem: Enclose the test and wait within a critical section that does not allow scheduling interrupts to occur between the test and the wait.

Formal Methods – After Flight
Shortly after the anomaly occurred during RAX on Tuesday May 18, 1999, the ASE team at NASA Ames decided to run a "clean room" experiment to determine whether technology currently used and under development could have discovered the bug. The experiment was set-up as follows:

- "Front-end" group tried to spot the error by human inspection. They identified about 700 lines of problematic code of tractable size for a model checker
- Problematic code was handed over to "Back-end" group with no hint regarding the error
- "Back-end" group further scrutinized the code and created a model of suspicious parts in Java. They used the Java Pathfinder (a translator from Java to a PROMELA model) and SPIN to expose the error.

The error was a missing critical section around a conditional wait on an event. It is a loop that starts with a `when` statement whose condition is sequential-or statement that *states if the event counter has not been changed (*1*) then wait else proceed (*2*).* This behavior is supposed to avoid waiting on the event queue if events were received while the process was active; however, if the event occurs between (*1*) and (*2*) it is missed and the process goes to sleep. Because the other process that produces those events is itself activated by events created by this one both end up waiting for each other – a deadlock situation.

Result
All involved parties regarded the formal methods verification effort before flight as a very successful application of model checking. According to the RA programming team, the effort had a major impact, locating errors that would probably not have been located otherwise and identifying a major design flaw prior to the in-flight Remote Agent experiment.

4.1.6. Success of Formal V&V

Formal Methods testing using the SPIN Model Checker had the following results:

- Original verification (occurred at the beginning of development) found five concurrency errors early in the design cycle that developers acknowledge could not have been found through traditional testing methods
- Quick-response verification performed after a deadlock occurred during the 1999 space mission, resulted in finding a concurrency error. Because this error was similar to the errors found before flight (original verification), it proves that Formal Methods testing can improve the safety and reliability of future missions by finding errors that traditional testing methods cannot.[3]

Suggestions for Improving Formal V&V
Tools for automatically generating a model will make model checking easier and more accurate. The NASA ARC ASE group is currently developing such tools and they will be discussed in Report 3, *New V&V Tools for Diagnostic Modeling Environment (DME)*.

4.2. Verification of Plan Models (Model Checking)[19]

Research is being conducted at NASA to investigate the benefit of using Model Checking to verify planning technologies like the Planner-Scheduler in Remote Agent.

4.2.1. Why Formal V&V

Artificial Intelligence planning technologies like the Planner-Scheduler in Remote Agent called HSTS (Heuristic Scheduling Testbed System) are very complex. They elicit and automatically manipulate system level constraints. Therefore, the HSTS models must be carefully verified. This section discusses current research being conducted at NASA ARC using model checking to verify these models.

4.2.2. Program Maturity

RA software including HSTS flew DS1 between May 17 and May 21, 1999 [20]

4.2.3. Program Software Subject to Formal V&V

HSTS is a general constraint-based planner and scheduler. In order to understand how HSTS works, the following terms must be defined:

- Tokens are used to represent intervals of time over which a state variable is in a certain state.
- Compatibilities are temporal constraints that may involve durations between end points of tokens. For example:
    - "Token 1 meets token2" indicates that the end of token1 should coincide with the start of token2
    - "Token1 before [3,5] token2" means that the distance between the end of token1 and the start of token1 is between 3 and 5
- DDL (Domain Description Language) – object oriented language for specifying Plan models. Figure 7 shows an example of a model defined in DDL.
- Plan Model - description of the domain given as a set of objects and constraints. A Plan Model is specified in an object-oriented language called DDL (Domain Description Language). It is based on a collection of objects that belong to different classes. Each object is a collection of state variables (also known as timelines). At any time, a state variable is in one, and only, one state. This state is identified by a predicate. Each predicate is associated with a set of compatibilities.
- Plan - a complete assignment of tokens for all state variables that satisfy all compatibilities, ranges of duration and disjunction of constraints. In simple terms, a Plan achieves the specified goal and satisfies the constraints in the Plan Model.

```
State Variables:  Rover, Rock
Rover predicates: atS, gotoRock, getRock,gotoS
Rock predicates:  atL, withRover

Compats:
1. Rover.getRock  dur[3,9]
      AND
        metby Rover.gotoRock
        meets Rock.withRover
      OR
            meets Rover.gotoS
            meets Rover.gotoRock
2.  Rover.atS     dur[0,10]
      AND
        metby Rover.gotoS
        meets Rover.gotoRock
3.  Rover.gotoRock      dur[5,20]
      AND
        OR
            metby Rover.atS
            metby Rover.atRock
        meets Rover.getRock
4.  Rover.gotoS
      AND
        metby Rover.getRock
        meets Rover.atS
5.  Rock.atL
      meets Rock.withRover
6.  Rock.withRover
      metby Rover.getRock
```

Figure 7: Model Defined Using DDL

HSTS consists of a planning engine that takes a Plan Model and a goal as input and produces a Plan.

Figure 8 shows an HSTS plan of a Rover getting a Rock sample. The initial state of the Rover is at the spacecraft or *atS.* The initial state of the Rock is at its location or *atL*. The goal is to have the Rock with the Rover or *Rock.withRover*. The plan for Rover is the following sequence:

- atS (begin at the spacecraft)
- gotoRock (go to the Rock)
- getRock (get the Rock)
- gotoS (return to the Spacecraft)

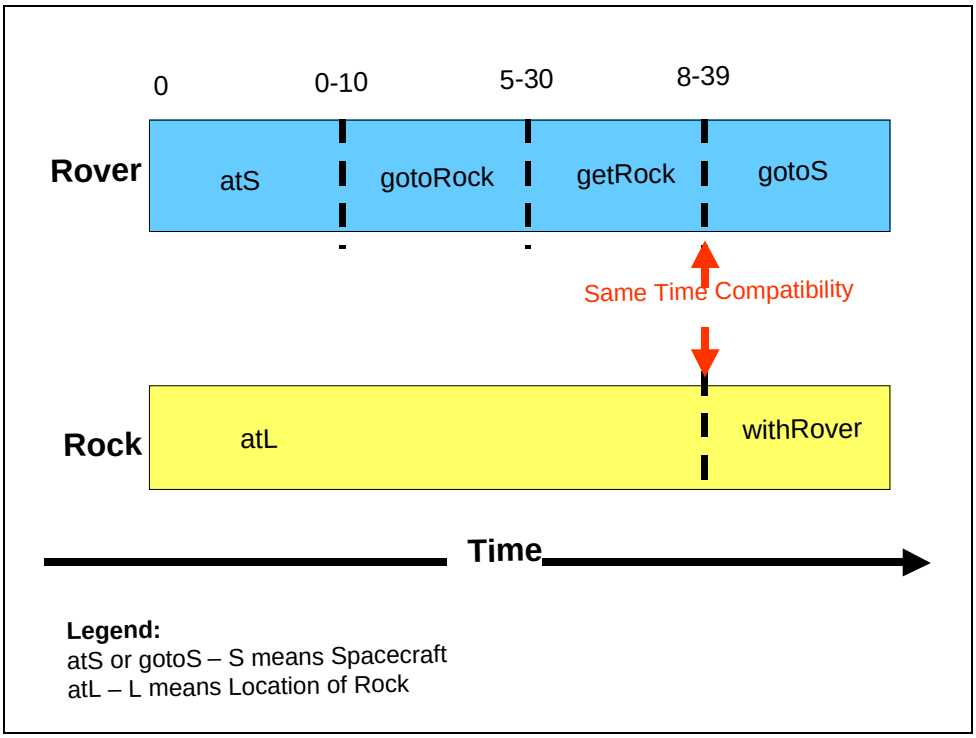


Figure 8: HSTS plan of a Rover getting a Rock

The plan includes duration constraints of the compatibility. For example, getRock token is between 3 and 9 time units. As a result, the goal of Rock.withRover may be satisfied in 8 to 39 time units.

4.2.4. Organization Performing Formal V&V

NASA AMES Research Center Autonomy Group

4.2.5. Formal V&V[3]

Research is being conducted at NASA to investigate the benefit of using the UPPAAL reasoning engine to verify HSTS models as well as verifying the HSTS reasoning engine. The first step is to find a mapping from HSTS models into UPPAAL. Next, a set of properties should be carefully constructed and checked.

Mapping HSTS models to UPPAAL

UPPAAL is a tool for modeling, simulation and verification of real-time systems. It was selected because it can represent time and is comparable to HSTS since they are both constraint-based systems.

The UPPAAL verifier is a symbolic model checker. Space reduction is accomplished by both local and global variable reduction. A UPPAAL model consists of a set of timed automata, a set of clocks, global variables and synchronizing channels.

HSTS models can be automatically translated to UPPAAL via a tool called DDL2UPPAAL. DDL2UPPAAL represents state variables as a UPPAAL automaton where each value of the state variable is represented as a node. Transitions of the automaton represent value ordering constraints of the corresponding state variable. Duration constraints are implemented through communication channels.

Properties for Verification
The following can be verified using DDL2UPPAAL:

- Ensuring correctness and detecting inconsistencies in the HSTS plan models. For example, detecting violations of mutual exclusion properties of predicates that are useful for detecting an incomplete specification of compatibilities in an HSTS model.
- Checking reach-ability of predicates to find inconsistencies in an HSTS model
- Goals in HSTS can be mapped into properties in UPPAAL so execution traces in UPPAAL correspond to plans in HSTS. This technique can also be used to verify the HSTS engine.

4.2.6. Success of Formal V&V

Initial research indicates that plan models can be mapped into timed automata to facilitate automated verification.

HSTS model translation from DDL to UPPAAL works well for models of a limited size and complexity. Techniques to verify larger systems, like building abstract models, appear promising and require additional investigation.

After translating a plan model, properties can be checked for inconsistencies and incompleteness in the model. Additionally, the model checking search engine can be used as an independent problem solving mechanism for verifying the planning engine. This is possible because goals can be mapped into properties and traces correspond to plans.

Suggestions for Improving Formal V&V
Research is currently underway to identify a set of verification properties that guarantee a certain degree of coverage for HSTS models and the Planning engine.

Research is also being conducted to analyze the benefits and limitations of using a model checker for HSTS verification.

DDL2UPPAAL is being enhanced to handle a larger subset of DDL so larger and more complex planning models may be verified.

4.3. RA EXEC Lightweight Formal Methods (Runtime Monitoring)

In this report, Lightweight Formal Methods is a term used by Martin Feather, Ben Smith and others at Jet Propulsion Lab (JPL) to describe a specific type of runtime analysis. One Lightweight Formal Method is called the Database Approach. The Database Approach provides a method for storing program logs, interface diagrams or other pertinent data in a database then querying this database for specific information.

4.3.1. Why Formal V&V

Critical software systems, like 2nd Generation IVHM, warrant high levels of assurance regarding correctness of their design and implementation while maintaining a cost effective V&V effort. Lightweight Formal Methods refer to formal methods that can be rapidly deployed while yielding beneficial V&V results.[21]

4.3.2. Program Maturity

RA software flew DS1 between May 17 and May 21, 1999 [22]

4.3.3. Program Software Subject to Formal V&V

Two pilot studies were conducted to investigate the feasibility of Lightweight Formal methods. The first pilot focused on analyzing software module interfaces of DS1 RA. The second pilot focused on analyzing test logs from the simulation test-bed of DS1 RA.

4.3.4. Organization Performing Formal V&V

Jet Propulsion Laboratory

4.3.5. Formal V&V

The Lightweight Formal Methods approach discussed in this report meet the criteria of rapid deployment and beneficial V&V results by leveraging the power of a database as the underlying reasoning engine. A typical database provides the following advantages:

- Flexible query language
- User-definable schema for expressing data relationships
- Support for loading data
- Powerful report generation capability

In the Database Approach, information to be analyzed is loaded as data into a database and the properties to be analyzed are cast as database queries.

The two pilot studies were conducted to evaluate the Lightweight Formal Methods Database Approach:

1. Pilot Study 1 – Analyzing Software Interfaces
2. Pilot Study 2 – Analyzing Test Logs

Pilot Study 1 – Analyzing Software Interfaces
DS1 RA is divided into several major software modules, which communicate via message passing. Because separate teams were responsible for each of these modules, there was a greater risk of failure caused by message passing anomalies. To promote communication between teams in the early stages of development, a diagram was drawn of each interface of each module. Figure 9: Smart Executive Interfaces shows one of these diagrams.

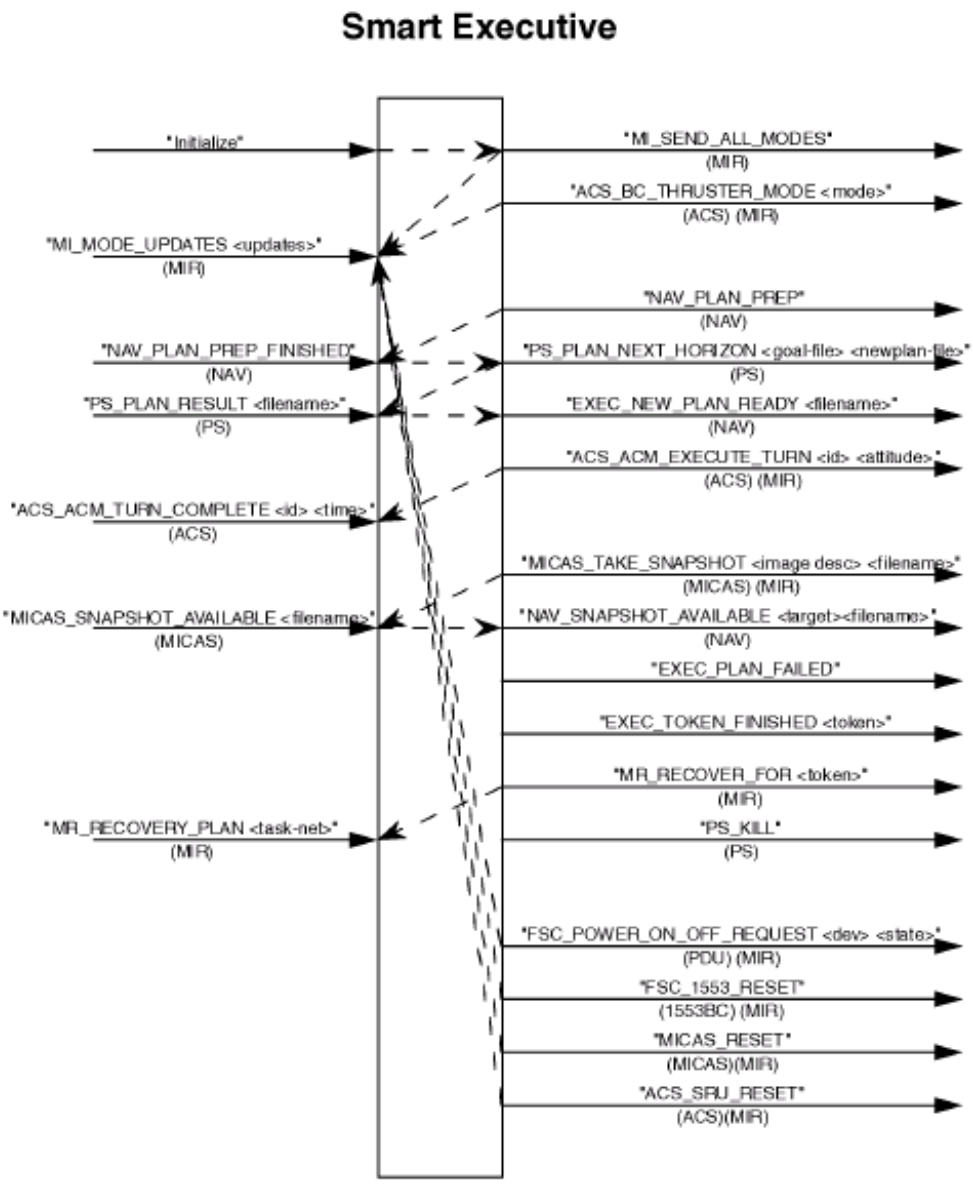


Figure 9: Smart Executive Interfaces
IEEE Transactions on Software Engineering, Vol. 24, No. 11, November 1998, p. 952

Diagram conventions:

- All possible input messages are shown as incoming arrows on the left
- All possible output messages are shown as outgoing arrows on the right
- Names of message types are shown in capital letters above arrows. Parameters, if any, are shown in angle brackets (i.e. <…>)
- For each incoming arrow, names of modules from which that type of message may originate are shown below the arrow
- For each outgoing arrow, names of modules to which that type of message is sent are shown below the arrow
- Cause-effect relationship between message types

  - Explicit: Shown as dotted line going from an incoming message to an outgoing message to indicate that receipt of such an incoming message may cause the module to produce the outgoing message
  - Implicit: Shown as a dotted line going from an outgoing message to an incoming message to indicate that the sending of such an outgoing message may, via actions of other software modules, lead to receipt of such an incoming message

Pilot Study 1 Objectives
V&V objectives for Pilot Study 1 are listed below:

- Find dangling outgoing message types. For example, a message type on an outgoing arrow of module X listed as going to module Y, but not listed on module Y's diagram as an incoming message type from module X
- Find dangling incoming message types. For example, a message type on an incoming arrow of module A listed as coming from module B, but not listed on module B's diagram as outgoing from module A
- Find mismatched parameters. For example, a message type whose list of parameters in one module is not identical to its list of parameters in some other module
- Find miraculous implicit cause-effect link. For example, a link from an outgoing message type to an incoming message type where there is no chain of explicit cause-effect links
- Find omitted implicit cause-effect link

Pilot Study 1 Formal Methods
The pilot study team selected AP5 (a research quality advanced database tool developed at the University of Southern California) and developed a schema to hold information about software interfaces. Data was loaded semi-automatically from PostScript files containing the interface diagrams.

Database queries were written based on V&V objectives. For example, the objective *"Find dangling outgoing message types"* was verified by retrieving a list of outgoing arrows that are not related to anything.

Results
Using Lightweight Formal Methods, the Pilot Study team found the following categories of anomalies:

- Inconsistency - contradictory information in the set of software interface diagrams
- Incompleteness - missing information

Analysis revealed that some false alarms occurred due to a need for further refinement of the database queries. For example, incompleteness resulted because the entire system had not been diagrammed. A simple refinement of the database queries made it possible to further categorize incomplete anomalies as follows:

- Internal incompleteness – missing information that should have been present
- External incompleteness – missing information due to lack of a software interface diagram

Another example of a false alarm was *missing implicit cause-effect links* that could be deduced from other links in the same diagram.

After fine-tuning the database queries to eliminate false alarms, 20 genuine anomalies were found.

Pilot Study 2 – Analyzing Test Logs
Before flight, DS1 RA software was tested in a simulation test-bed. Each test generated logs of the software's behavior including messages passed between software modules. The DS1 RA test team studied these logs to ensure the software was working properly. This was a very time consuming process because the logs were highly detailed and very lengthy (several thousand messages or more). To expedite log review the Lightweight Formal Methods Database Approach was used.

Pilot Study 2 Objectives
The pilot study team determined that two kinds of information could be analyzed from the test logs:

1. Whether the controlled spacecraft adheres to all its flight rules
2. Whether the message flow between software modules follows expected protocols. Protocol example: a command message is followed by the corresponding confirmation message before another command message is sent

Pilot Study 2 Formal Methods
The pilot study team selected AP5 (a research quality advanced database tool developed at the University of Southern California) and developed a schema to hold information from the test logs.

Data was parsed from the log and loaded into the database. To speed the data loading process, only information from relevant messages was loaded.

Analysis was achieved by expressing a flight rule or message protocol as a database query that would retrieve all instances of messages in violation of that condition. For example: one flight rule says: *"when the Ion Propulsion System (IPS) is not thrusting, be in Reaction Control System (RCS) control mode".* This could be violated either by turning off thrusting while not in RCS control mode or turning off RCS control mode while not thrusting. Two queries (listed here in pseudo-query language) were written to check for these violations:

1. Find an IPS_thrust_off message that occurs while RCS control mode is off
2. Find an RCS_control_mode_off message that occurs while IPS is not thrusting

Results
No genuine anomalies were discovered during this pilot study, most likely because for the patterns and flight rules studied, the design was functioning correctly. However, this pilot study demonstrated the feasibility of rapid analysis of large amounts of data.

4.3.6. Success of Formal V&V

The Lightweight Formal Methods Database Approach provides a straightforward, simple tool for conducting consistency and completeness checks by leveraging the power of a database as a reasoning engine. It provides flexibility to quickly construct new tests and to refine previous tests, as necessary.

Adapting the Database Approach to the existing available information was relatively easy and pilot studies indicate that it is feasible to quickly analyze large amounts of data.

While it is possible a CASE tool would provide some of the same types of information, CASE tools can be expensive and are not always available.

Finally, 20 anomalies were found early in the Software Life Cycle indicating that using Lightweight Formals Methods can improve the accuracy and reduce the cost of software development.

Suggestions for Improving Formal V&V
A user interface could be developed so a test engineer can select a single test condition and have the appropriate queries be automatically generated.

For example, a test engineer could request a test of the flight rule, *"when the IPS is not thrusting, be in RCS control mode",* and two queries (listed in section 4.3.5) would be generated and run automatically.

5. FORMAL METHODS APPLICABLE TO 2ND GENERATION RLV IVHM

The purpose of this section is to discuss which Formal Methods used at NASA are beneficial to the IVHM system planned for the 2nd Generation RLV.

Assumptions
In order to properly evaluate the applicability of Formal Methods to 2nd Generation RLV IVHM, the following assumptions were made:

- IVHM will include functionality described in the following document:

  2nd Generation RLV, TA-5 IVHM, NASA In-House Task 1, IVHM Concept of Operations Document

- Verification and validation of diagnostic systems such as IVHM fall into the general approaches summarized below:
  - Formal methods should be applied to the functional model to verify that certain universal and problem-specific conditions are met for all applicable states of the model
  - Simulation testing should be applied to the model by inputting selected parameters and checking the resulting diagnosis against known or trusted results. For example: trusted results for a Livingstone model could be obtained by checking one Livingstone model against a second Livingstone model

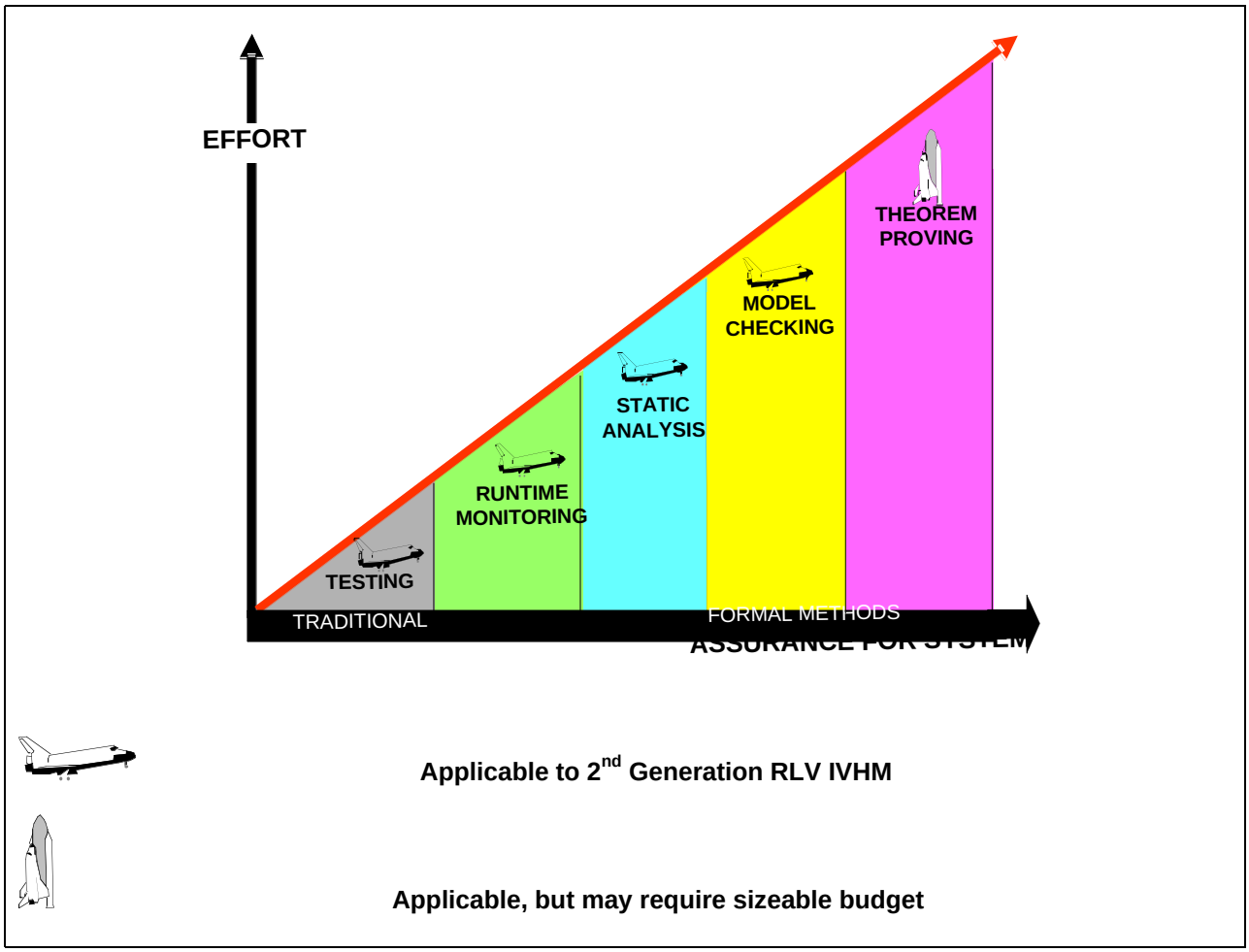
  (Suggested by Dr. Steve Brown from Northrop Grumman in working notes in *Model-based Verification of Diagnostic Systems* included in Appendix A of Report 3*, New V&V Tools For Diagnostic Modeling Environment (DME)*)

Applicability
Applicability was determined by reviewing the following documents:

- NASA V&V guidebooks
- Formal V&V activities described in the survey:
  - Surveys of the Formal V&V of Remote Agent (Model Checking)
  - Verification of Plan Models (Model Checking)
  - Lightweight Formal Methods (Runtime Monitoring)

Applicable Formal Methods may be incorporated into the V&V Plan for the 2nd Generation RLV IVHM system.

The following diagram summarizes Formal Methods applicable to 2nd Generation RLV IVHM and the effort versus assurance of each method.



An explanation of how each applicable Formal Method benefits the 2nd Generation RLV IVHM system follows:

Testing - benefits of traditional testing and the applicable NASA standards were discussed at in detail in Report 1, *Survey Of NASA V&V Processes/Methods.*

Runtime Monitoring - requires a relatively small incremental effort over traditional testing and can locate difficult to find potential for errors

Static Analysis – can enable verification to begin earlier in the Software Life Cycle resulting in early detection/resolution of problems and possible reduction in development cost

Model Checking – provides a fast, automated method for exploring all relevant execution paths of a system and its environment. This is very important for systems like 2nd Generation RLV IVHM, because it is virtually impossible for humans to conceive every test scenario required to verify the system in all possible situations in a plausible time frame for software development.

For a given coverage, Model Checking can be much faster than testing, by avoiding the costly reset between tests and by tracking explored states to avoid redundant executions. It also detects problems in the early stages of development; thereby greatly reducing overall development costs. [17]

Model Checking has proven very useful for the validation of domain models used in model-based diagnosis systems such as Livingstone, which is being considered for 2nd Generation RLV IVHM.

Theorem Proving - Requires significant effort and expertise making Theorem Proving more suitable for analysis of small-scale designs by verification experts.[23]

6. INCORPORATING FORMAL METHODS INTO NASA STANDARDS

The purpose of this section is to provide guidance on identifying changes necessary to integrate Formal Methods into the Software Life Cycle described in the following NASA standards[24] for airborne software:

- NASA Procedures and Guidelines (NPG) 2820.DRAF1, NASA Software Guidelines and Requirements.

  Note: *This document, NPG 2820 DRAF 1, references the following IEEE (Institute of Electrical and Electronics Engineers) documents:*

  - *12207.0 - Standard for Information technology – Software Life Cycle Processes (March, 1998)*
  - *12207.1 - Standard for Information technology – Software Life Cycle Data (April, 1998)*
  - *12207.2 - Standard for Information technology – Software Implementation Considerations (April, 1998)*
- NASA Procedures and Guidelines (NPG) 8730.DRAFT 2, Software Independent Verification and Validation (IV&V) Management
- *Software Considerations in Airborne Systems and Equipment Certification,* Document No RTCA (Requirements and Technical Concepts for Aviation) /DO-178B, December 1, 1992.

Note: *Report 1, Survey Of NASA V&V Processes/Methods, Section 4.1.1 Overview of NASA Standards contains details about NASA Standards for airborne software.*

This section includes the following:

- Prerequisites for successful integration of formal methods into a program following NASA standards
- Recommendation for where to add formal methods in the Software Life Cycle
- Suggestions of metrics to measure the effectiveness of formal methods

6.1. Prerequisites

Effective integration of Formal Methods into a NASA program requires the program to establish a well-defined process with the following characteristics:

- Discrete Software Life Cycle phases clearly defined and documented
- Work products (deliverables) specified for each phase
- Analysis procedures established to ensure correctness of work products
- Scheduled reviews of major work products

6.2. Where to Add Formal Methods

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 traditional 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 as shown in the following figure. As the legend reflects, boxes filled with yellow (light gray if printed via black and white printer) indicate model checking. Boxes filled with blue hashing (gray hashing) indicate static analysis and boxes filled with magenta (dark gray) indicate runtime monitoring.

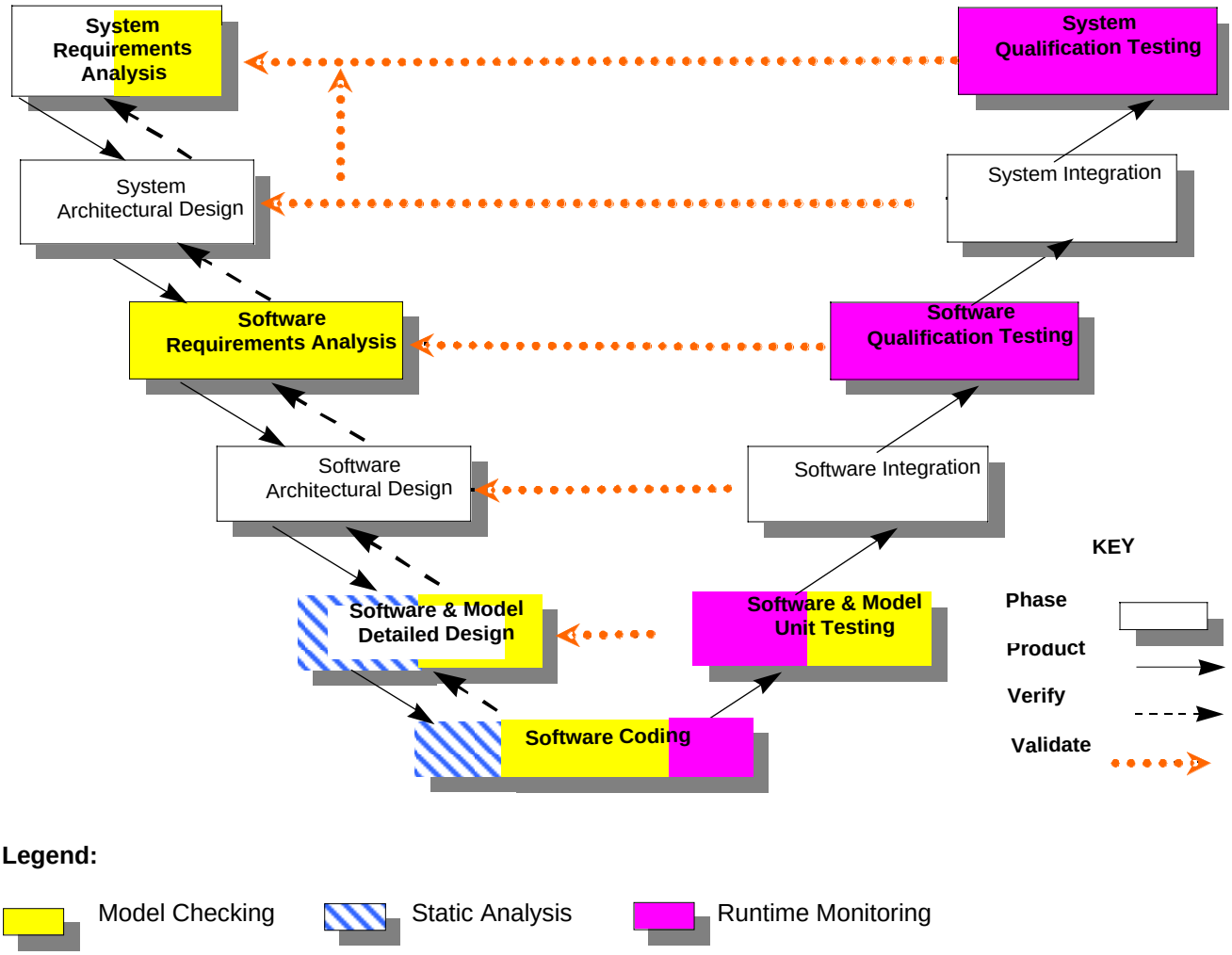


Figure 10: Formal Methods Inserted into Software Life Cycle

Specific techniques of integrating Formal Methods into the Software Life Cycle are explained in the following sections:

- Planning for Formal Methods
- New V&V Processes
- New Technical Methods
- Cost Considerations

6.2.1. Planning for Formal Methods

Planning for Formal Methods includes activities at two levels in the life cycle. At the beginning of the program, staffing requirements and enhanced project guidelines must be considered:

- Staffing for Formal Methods:
    - Formal Methods Planning Team: at least one person knowledgeable in formal methods and one person knowledgeable about the application domain
    - Formal Methods V&V Team: must have formal methods expertise or be provided with hands-on training
- V&V Guidelines: V&V guidelines, standards, and conventions, both for documentation and specification, should be developed early and followed carefully

After the Formal V&V team has been selected, they must plan how and when to use Formal Methods. The following table provides planning steps for Formal Methods:

Table 4: Planning Formal V&V

Planning Step	**Notes**
Define scope of Formal Methods effort and identify Formal Methods expertise and application domain expertise	Is this a Trial, Partial or Full Scale Project? Both application domain and Formal Methods expertise are essential
Choose application for analysis with Formal Methods	Criticality assessments, assurance considerations, and architectural characteristics are among the key factors used to determine which subsystems or components to analyze with formal methods. Since large systems are typically composed of components with widely differing criticalities, the extent of formal methods use should be dictated by project-specific criteria. For example, a system architecture that provides fault containment for a critical component through physical or logical partitioning provides an obvious focus for formal methods and enhances its ability to assure key system properties.
Select Formal Methods	The choice of a Formal Methods technique is dictated by the application. For best results use several complementary methods. For example: • Model Checking explores the properties of mode-switching or other complex control logic and could be easily combined with Static Analysis and/or Runtime Monitoring • Theorem Proving verifies correctness of fault-tolerant algorithms and could be used in conjunction with any of the other Formal Methods

Planning Step	**Notes**
Select Formal Methods Tools	The process of selecting a formal methods tool is similar to selecting any other software tool; the usual considerations of documentation, tutorials, ease of use, etc. apply. Effective technical support is also important. Some additional considerations are described below: • Can the tool answer the questions posed to it effectively? • Does the tool apply to the language being used for development? • Does the tool offer a reasonably comprehensive library of standard types, functions, and other constructions? • What, if any, editing and document preparation tools does the tool provide? • Are there facilities for cross-referencing, browsing, and requirements tracing? • Is there support for incremental development across multiple sessions and for change control and version management? • Specification Language: Is the language adequately expressive for the given application? Does it have the following: o Well-defined semantics o Modern programming language constructs (including support for abstraction, modularity, and encapsulation) o Familiar and convenient syntax o Strong typing o Encapsulation o Parameterization o Built-in model of computation o Executable subset or other provision for animating specifications o Support for state exploration, model checking, and related methods

Planning Step	**Notes**

Select Formal Methods Tools *(continued)*	• Model Checker:
	  • Does the Model Checker have a translation program to automate translation from application domain languages (such as Livingstone models) into the appropriate syntax?
	  • Does the Model Checker optimize state space to prevent running out of computer memory while performing verification?
	  • Does the Model Checker provide potential errors or error logs in an easy to understand format?
	  • Can the Model Checker support state space reduction?
	  • Is the model checker flexible enough to provide exhaustive and focused coverage?
	  • Is model checker powerful enough to cope with the complexity of the problem?
	• Theorem Prover:
	  • Does the formal methods tool offer a theorem prover or proof checker?
	  • If so, how is the theorem prover controlled and guided?
	  • Is there automated support for arithmetic reasoning, efficient handling of large propositional expressions, and rewriting?
	  • What support is there for developing and viewing the proof?
	  • Can lemmas be used before they are proved and can new definitions be introduced and existing definitions modified during proof?
	  • How is the proof presented to the user (e.g., user input or canonical expressions, with or without quantifiers)?
	  • Are the foundations (i.e., all axioms, definitions, assumptions, lemmas) of the proof identified?
	  • Are there facilities for editing proofs and is it reasonably easy to re-verify a theorem after slight changes to the specification?
	A comprehensive list of Formal Methods tools is maintained by Jonathan Bowen and is available at: http://www.afm.sbu.ac.uk

Implement Formal Methods Training	View training as an investment
Develop Formal Methods Project Guidelines	Considerations analogous to those for conventional software
Track & Document Formal Methods Process Changes	Revise the process and documentation with project feedback, as appropriate

6.2.2. New V&V Processes

For each formal method used, consider replacing or enhancing traditional processes in the appropriate Software (SW) Life Cycle Phase with the activities described in the following table.

Formal Methods	**Applicable SW Life Cycle Phase**	**Formal Verification Activities**
Any	System Requirements Analysis SW Requirements Analysis	Perform a new development activity called "formalization" during which a new work product called a "formal specification" is created. This can be a separate product or an addition to an existing work product such as a requirements document. Documenting requirements reduces confusion later in the project and promotes customer approval of the software and system. Creating a formal specification enables the application of formal methods at later stages. It can also increase the accuracy of requirements and promote communication between developers and test engineers.
Model Checking (Theorem Proving)	System Requirements Analysis SW Requirements Analysis	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.
Model Checking (Theorem Proving)	System Requirements Analysis SW Requirements Analysis	Perform an Official Review of the formal specification to check the coverage, correctness, and comprehensibility of the formal specification. ***Note:*** *Refer to Section 3 of Report 3, New V&V Tools For Diagnostic Modeling Environment (DME), for a discussion on correctness and reliability criteria for IVHM.*
Any	System Requirements Analysis SW Requirements Analysis	Enhance traceability tools and techniques to track new products such as formal specifications and proofs, and their relationships to existing products
Model Checking	SW Requirements Analysis SW & Model Detailed Design	Perform a new analysis activity called "modeling", producing a new work product called a "formal model". Modeling the software or system allows for model checking to be applied. Some recent model checking tools are applicable directly to design languages such as UML, either natively or by translation to their own modeling language.
Model Checking	SW Requirements Analysis SW & Model Detailed Design	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.

Formal Methods	Applicable SW Life Cycle Phase	Formal Verification Activities
Model Checking	SW Requirements Analysis SW & Model Detailed Design	Perform an Official Review of the model to check for correctness ***Note:*** *Refer to Section 3 of Report 3, New V&V Tools For Diagnostic Modeling Environment (DME), for a discussion on correctness and reliability criteria.*
Static Analysis	SW & Model Detailed Design 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. Static Analysis may also be applicable at the later stages of the Detailed Design phase.
Model Checking	SW Coding 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.

6.2.3. New Technical Methods

The technical methods required to use new verification tools for Advanced Software like 2nd Generation RLV IVHM are similar to traditional automated testing methods. Like traditional testing tools, they require specific types of input and generate output like reports listing potential errors.

Specific details about the MPL2SMV, JMPL2SMV and Livingstone PathFinder V&V tools are contained in Report 3, *New V&V Tools For Diagnostic Modeling Environment (DME).*

6.2.4. Cost Considerations

Cost-effectiveness of using formal methods depends on the following:

- Characteristics of the project
- Productivity of the staff
- Nature of the work environment
- Availability of resources
- Potential for re-use

Due to the difficulty of using statistical techniques to analyze software engineering methods[25] reliable data on formal methods cost and effectiveness is hard to come by, although available data strongly suggests that judiciously applied formal methods are cost-effective.

Two examples of cost-effective use of Formal Methods at NASA are listed below:

- According to the Formal Methods Specification and Verification Guidebook For Software and Computer Systems Volume I: Planning and Technology Insertion, data collected in the *Formal Methods Demonstration Project for Space Applications – Phase I Case Study: Space shuttle Orbit DAP Jet Select*, JPL Document D-11432, December 22, 1993 indicates that the act of formally stating specifications is generally cost-effective. For critical applications, the act of proving key properties also appears to be cost-effective, although there is less data to support this claim.[32]
- The RA modeling effort took about 12 person-weeks during a six calendar week period. The verification effort took one week. All involved parties regarded the formal methods verification effort as a very successful application of model checking. According to the RA programming team, the (model checking) effort had a major impact, locating errors that would probably not have been located otherwise and identifying a major design flaw.[3]

6.3. Metrics

Potentially useful metrics include the following:

- 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.)

Specific to Model Checking

- Amount of time spent in model development (both human and CPU time)
- Amount of coverage

Specific to Theorem Proving

- Number of pages of English description that were used as the basis for the formal specification, along with a subjective indication of their level of detail and completeness (e.g., high, medium, low)
- Number of lines of formal specification produced
- Amount of time spent in developing specifications, including properties and proofs

7. 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*

8. 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[26]
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[27].
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[28] (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[29]
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[31]
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.[30]
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[31]
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)

9. FOR MORE INFORMATION

Papers located at http://ase.arc.nasa.gov/pecheur/publications:

- Charles Pecheur, Reid Simmons. *From Livingstone to SMV: Formal Verification for Autonomous Spacecrafts*. In Proceedings of 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.

- Charles Pecheur. *Verification and Validation of Autonomy Software at NASA*. NASA/TM 2000-209602, August 2000.

- 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.

- 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. http://ase.arc.nasa.gov/pecheru/publi.html

Papers located at http://ase.arc.nasa.gov (click on authors name to view their website)**:**

- Klaus Havelund, Grigore Rosu. *Monitoring Java Programs with Java PathFinder.* To be published in Electronic Notes in Theoretical Computer Science.

- Guillaume Brat and Willem Visser. "Combining Static Analysis and Model Checking for Software Analysis".

- Khatib, L., Muscettola, N., and Havelund, K., *Mapping Temporal Planning Constraints into Timed Automata*, Time-01 (IEEE press), the Eighth International Symposium on Temporal Representation and Reasoning, Cividale Del Friuli, Italy, 2001.

- Khatib, L., Muscettola, N., and Havelund, K., *Verification of plan models using UPPAAL*, First Goddard Workshop on Formal Approaches to Agent-Based Systems, Greenbelt, Maryland, 2000. (to appear in a special issue of Lecture Notes in Computer Science)

Other Publications:

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.

NASA Guidebooks:

- *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion*, National Aeronautics and Space Administration, December 1998

- Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume II: A Practitioner's Companion, National Aeronautics and Space Administration, December 1998

Books:

- Beatrice Berard, 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

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

Websites:

Deep Space One Website: http://nmp.jpl.nasa.gov/ds1/

10. APPENDIX A: FORMAL METHODS EXAMPLE – THEOREM PROVING[32]

Note: This example, including references, was copied directly from *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion*, National Aeronautics and Space Administration, December 1998, pp. 7-10.

The example illustrates the use of formal specifications to model a system, to enhance the consistency of the specification, and to suggest the role of proof in establishing desired system properties. The purpose of this discussion is to provide a concrete, albeit small and highly simplified example. Readers interested in a more detailed tutorial discussion should consult [Butler[33]], [Weber-Wulf[34]], and [Wordsworth[35]]. Those interested in more realistic or industrial-scale applications can find excellent discussions in papers, technical reports, and books, including [Miller1[36]] and [Bowen2[37]].

Consider the following typical informal requirements expressed in English:

> A tank of cooling water shall be refilled when its low level sensor comes on. Refilling consists of adding 9 units of water to the tank.
>
> Notes:
> - The maximum capacity of the tank is 10 units of water.
> - From one reading of the water level to the next reading of the water level, 1 unit of water will be used.
> - The low level sensor comes on when the tank contains 1 unit of water or less.

The above statement contains several descriptions, including two key notions: the water level in the tank and the water usage. Formally, these notions can be modeled as follows (statements 1 and 2):

> **1** **`level`** is represented by a restricted integer type: a number between 0 and 10, inclusive
> **2** **`usage`** is represented as the integer constant 1

That is, `level` describes an amount of water that the tank may hold at any point in time and `usage` describes the amount of water used during one cycle.

The primary requirement is that 9 units of water will be added to the tank whenever the level is less than or equal to 1. This can be more precisely[∇] stated as (statement 3):

> **3** Function **`fill`** takes, as input, a water level and returns, as output, a water level. Given an input of `L` units of water, `fill` returns `L`+9 if `L` is one or less, otherwise it returns `L`.

That is, we claim that `fill(L)` accounts for any filling of water in the tank.

A commonsense property of this system is that, at the next cycle, the new water level will be the current water level, plus any amount that was added, minus the amount that was used. That is, given `L` as the current level of water, the level at the next cycle should be given by statement 4:

```
4     level = L + fill(L) - usage
```

One approach to checking this specification is to ensure that each reference to a level of water is consistent with the definition of `level`, i.e., it should always be a number between 0 and 10. It turns out

[∇] This specification is given in a form of structured English so that the reader can easily follow it without having to learn a formal specification language. Such specifications are more precise than those written in conversational English but are still less precise than those written in a formal specification language.

that the specification for `fill` given in 3 above is consistent with the definition of `level` if the following two logical statements are true:

```
5     FORALL levels L
            (L <= 1) IMPLIES THAT
                  (0 <= L + 9) AND
                  (L + 9 <= 10)

6     FORALL levels L
            (0 <= L + fill(L) - usage) AND
            (L + fill(L) - usage <= 10)
```

(Statements 5 and 6 can be derived straightforwardly by means of formal methods techniques. Many formal methods tools can produce such expressions automatically from a set of system definitions.) The following statements (statements `5.1` and `5.2`) constitute an informal proof that the *first* `FORALL` statement (statement 5) is true:

5.1 L+9 >= 0 because L >= 0 (and the sum of any two numbers greater than zero is greater than zero)

5.2 L+9 <= 10 because L <=1 (and any number less than or equal to 1 plus 9 is less than or equal to 10)

However, the *second* `FORALL` statement (statement 6) is not true. Consider the case when L is 9:

L + fill(L) - 1 = L+L-1 = 9+9-1 = 17 (which is not <= 10)

So clearly, something is wrong. Upon closer examination, it is found that statement 4, our expression for the water level at the next cycle, is in error:

```
4     level = L + fill(L) - usage   (incorrect)
```

This statement is inconsistent with the definition of `fill` because `fill` returns the new level of water, not just the amount of water added. The (corrected) expression for `level`, denoted by 4', is simply:

```
4'    level = fill(L) - usage  (correct)
```

and the (corrected) `FORALL` statement (statement 6) is:

```
6'    FORALL levels L:
            (0 <= fill(L) - usage) AND
            (fill(L) - usage <= 10)
```

This example illustrates the following:

- **Formal Specification**: Modeling informal English statements using mathematical expressions
- **Type Checking**: Checking that all types of items are used consistently (e.g., level)
- **Stating Properties**: Identifying and defining expected behavior of the system (e.g., the expected new level in the tank).
- **Proving Logical Conditions**: Constructing logical proofs which show that a given condition holds under all possible situations.

This example also illustrates how formal analysis can expose errors and inconsistencies in a specification. In the example, the name chosen for the “fill” function in statement 3 is misleading because

the function returns the “actual level” rather than the “amount added.” Statement 4, although wrong, is consistent with the casual reader’s expectations, so the error is easy to overlook.

In simple cases such as this, an informal inspection of the specification can be expected to find the error. However, the use of formal methods resulted in a systematic and reproducible approach to uncovering the problem. Similar results can be achieved in challenging industrial-scale specifications, where such errors can be obscured within many pages of requirements.

This example does not show how tools can be used to assist in formal analysis. That topic is addressed in Volume II, *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume II: A Practitioner’s Companion*[38]

11. APPENDIX B: PVS (THEOREM PROVING)

Note: *Appendix B was copied from Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion*, National Aeronautics and Space Administration, December 1998, B-16-17

NAME: PVS (Prototype Verification System)

LANGUAGE: Classical, typed higher-order logic with predicate subtypes, dependent typing, and abstract data types.

FEATURES: Customized GNU Emacs interface, parser, typechecker, integrated proof checker, BDD simplifier, prettyprinter, browser, specification libraries, and facilities for status-reporting, cross-reference generation, and LaTeX-printing.

SYNOPSIS: PVS provides an integrated environment for the development and analysis of formal specifications and is intended primarily for the formalization of requirements and design-level specifications, and for the rigorous analysis of difficult problems. PVS has been applied to algorithms and architectures for fault-tolerant flight control systems, to problems in real-time system design, and to hardware verification.

PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms and theorems. Definitions are guaranteed to provide conservative extension. Libraries of proved specifications from a variety of domains are available.

PVS offers a rich type system, strict typechecking, and powerful automated deduction with integrated decision procedures for linear arithmetic and other useful domains, and a comprehensive support environment. A PVS specification is typically expressed using type constraints that are enforced through automatically generated proof obligations, many of which are automatically discharged by the system. The expressive specification language allows concise and natural specifications across a wide range of problem domains. The proof checker provides direct control by the user for the higher levels of proof development, and powerful automation for the lower levels, using a collection of primitive inference procedures that can also be combined by the user to develop higher-level proof strategies. Proofs yield scripts that are displayed in a readily understood format and can be edited and reused. Context is preserved across sessions.

DOCUMENTATION:

S. Owre, N. Shankar, J. M. Rushby, “User Guide for the PVS Specification and Verification System (Beta Release),” Computer Science Laboratory, SRI International. Three volumes: Language, System, and Prover Reference Manuals.

These and other manuals, papers, and technical reports both by the formal methods group at SRI and outside users are documented in the SRI WWW page and available by anonymous FTP.

FTP: ftp to ftp.csl.sri.com, connect to directory /pub/reports
WWW: http://www.csl.sri.com/sri-csl-fm.html
User group: pvs@csl.sri.com

TOOL REQUIREMENTS: PVS is implemented in Common Lisp and runs on most modern workstations; the requirements are a Unix machine that runs Gnu Emacs and a Common Lisp compiler with integrated CLOS. If typeset specifications are of interest, LaTeX and an appropriate viewer must also be available. The standard version of PVS is implemented in Allegro Lisp and runs on Sun SPARCstations. PVS requires about 20 megabytes of disk space, 50 megabytes of swap space, and 32 megabytes of real memory.

AVAILABILITY: PVS is available by tape or by FTP (ftp://ftp.csl.sri.com/pub/pvs). All installations of PVS must be licensed by SRI. There is no license fee and no charge for a PVS system obtained via FTP. A nominal distribution fee is charged for tapes and for nonstandard versions. Requests should be addressed to pvs-request@csl.sri.com or to one of the following contacts.

John Rushby, N. Shankar, Sam Owre

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Email: {rushby, shankar, owre}@csl.sri.com
Phone: +1-415-859-5456/5272/5114
Fax: +1-415-859-2844

12. APPENDIX C: SPIN (MODEL CHECKING)

Note: *Appendix B was copied from Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion*, National Aeronautics and Space Administration, December 1998, B-28-29

NAME: Spin

LANGUAGE: PROMELA (Process Meta Language) is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and Hoare's language CSP. It contains the primitives for specifying asynchronous (buffered) message passing via channels, with arbitrary numbers of message parameters. It also allows for the specification of synchronous message passing systems (rendezvous). Mixed systems, using both synchronous and asynchronous communications, are also supported.

The language can model dynamically expanding and shrinking systems: new processes and message channels can be created and deleted on the fly. Message channel identifiers can be passed from one process to another in messages.

Correctness properties can be specified as standard system or process invariants (using assertions), or as general linear temporal logic requirements (LTL), either directly in the syntax of next-time free LTL, or indirectly as Buchi Automata (expressed in PROMELA syntax as Never Claims).

FEATURES: Spin can be used in three basic modes:

- As a simulator, allowing for rapid prototyping with random, guided, or interactive simulations;
- As an exhaustive state space analyzer, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search);
- As a bit-state space analyzer that can validate even very large protocol systems with maximal coverage of the state space (a proof approximation technique).

SYNOPSIS: Spin is a widely distributed software package that supports the formal verification of distributed systems. The software was developed at Bell Laboratories in the formal methods and verification group.

Spin has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

DOCUMENTATION:

Gerard J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall 1991.

Online Documentation:
Documentation for the basic version of Spin consists of a concise manual, a set of online manual pages, some research papers, and a book (also available online). Basic and more advanced usage of Spin, such as language features and validation modes for proving linear temporal logic formulas, are described in the book. The more recent extensions of the tool are described in the papers.

The book contains an explanation of the code and describes the main validation strategies. More details, and help with installation and usage of this software is always available from the author. Questions can also be posted to the Spin users list.

TOOL REQUIREMENTS: The Spin software is written in ANSI standard C, and is portable across all versions of the UNIX operating system. It can also be compiled to run on any standard PC running a Windows 95 or Windows NT operating system.

AVAILABILITY: The software can be downloaded from the following address: http://netlib.bell-labs.com/netlib/spin/whatispin.html

The current version is Spin 3.1, which runs on any Unix workstation, as well as Windows 95 or Windows NT.

13. APPENDIX D: SMV (MODEL CHECKING)

This appendix contains an overview and information about current versions of SMV. It also discusses future research directions for SMV.

13.1. Overview of SMV

Note: *The following overview of SMV was copied from Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion, National Aeronautics and Space Administration, December 1998, B-26-27.*

NAME: SMV (Symbolic Model Verifier)

LANGUAGE: The input language, SMV, is a relatively high-level description language that provides modular hierarchical descriptions and definition of reusable components. The specification language, Computation Tree Logic (CTL), is a propositional, branching-time temporal logic.

FEATURES: Symbolic model checker verifies finite state systems described in the SMV language against specifications written in CTL. Implemented with BDDs (reduced, ordered Binary Decision Diagrams), SMV can handle both synchronous and asynchronous systems, and arbitrary safety and liveness properties.

SYNOPSIS: The SMV system has been distributed widely and used to verify industrial-scale circuits and protocols, including the cache coherence protocol described in the IEEE Futurebus+ standard and the cache consistency protocol developed at Encore Computer Corporation for their Gigamax distributed multiprocessor.

SMV is designed to provide largely automatic verification of finite state system descriptions that run the gamut from completely synchronous to completely asynchronous, and from detailed to abstract. The SMV input language offers a set of basic data types consisting of bounded integer subranges and symbolic enumerated types, which can be used to construct static, structured types. CTL provides a concise syntax for expressing a rich class of temporal properties including safety, liveness, fairness, and deadlock freedom. SMV uses a BDD-based symbolic model checking algorithm to avoid explicitly enumerating the states of the model. With carefully-tuned variable ordering, the BDD algorithm yields a system capable of verifying circuits with extremely large numbers of states. Examples of the scalability of this approach include a pipelined ALU with over 10^{120} states and an asynchronous stack with over 10^{50} states.

DOCUMENTATION:

J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, “Symbolic Model Checking for Sequential Circuit Verification,” *IEEE Transactions on Computer-Aided Design*, Vol. 13, No. 4, April 1994.

K. McMillan. *Symbolic Model Checking*. Kluwer, Boston, MA, 1993.

13.2. Current Versions of SMV

Currently, three versions of SMV exist:

- Carnegie Melon University (CMU) SMV (http://www-2.cs.cmv.edu/~modelcheck/smv.html)
- NuSMV (http://nusmv.irst.itc.it)
- Cadence SMV (www-cad.eecs.Berkeley.edu/~kenmcmil/smv)

13.2.1. CMU-SMV

This is the original SMV program developed at Carnegie Mellon University by Ken McMillan in Prof. Clarke's group.

Several versions of CMU-SMV are available including a new version for Windows NT and a recent upgrade for Unix. Enhancements include new options for printing and controlling model checking capabilities and new features like the following:

- Partitioning of BDDs
- Increased the number of BDD levels to 100K with reordering and 2G without reordering.
- Increased the number of program variables to 5000 from 500
- Bug fixes
- Source code modularization
- Signal handling to catch more UNIX signals and print to the standard report

See http://www-2.cs.cmu.edu/~modelcheck/smv.html for more information and to download SMV tools.

TOOL REQUIREMENTS: Windows NT or Unix (SPARC, Intel x86 Linux, DECstation Ultrix)

13.2.2. NuSMV

NuSMV is a symbolic model checker developed as a joint project between:

- Formal Methods group in the Automated Reasoning System division at ITC-IRST (Istituto Trentitno di Cultura)
- Model checking group at Carnegie Mellon University
- Mechanized Reasoning Group at University of Genova
- Mechanized Reasoning Group at University of Trento

It is a result of reengineering and extending SMV. NuSMV includes:

- New text interface and GUI
- Extended model partitioning techniques
- LTL model checking
- New modular architecture

A detailed overview of NuSMV (including architecture) is located at http://nusmv.irst.itc.it/NuSMV/papers/cav99/html/paper.html.

A comparison between CMU-SMV and NuSMV is located at http://nusmv.irst.itc.it/NuSMV/papers/sttt_j/html/node46.html.

NuSMV version 2 is available for downloading at http://irst.itc.it. It features new:

- SAT-based Bounded Model Checking algorithms. Bounded model checking searches for counterexample to an assertion using up to a fixed number of transitions. A satisfiable (SAT-based) instance indicates the presence of a counterexample of the given length or shorter.
- Partitional algorithms
- Cone of Influence reduction – remove variables in a range of variables that are not relevant

TOOL REQUIREMENTS: Windows NT or Unix

13.2.3. Cadence SMV

Cadence SMV was re-developed by Ken McMillan at Cadence Berkeley Laboratories. It comes equipped with two modeling languages, extended SMV and Synchronous Verilog.

Cadence SMV allows several forms of specification including the temporal logics CTL and LTL, finite automata, embedded assertions and refinement specifications. It also has a GUI and source level debugging capabilities.

For more information see: www-cad.eecs.Berkeley.edu/~kenmcmil/smv.

TOOL REQUIREMENTS: Windows 95 or NT, Sparc/Solaris/ Sparc/SunOS, HPUX, MIPS/Irix and i386/Linux

14. APPENDIX E: MAUDE

Maude is a high-performance, modularized specification and verification system supporting both rewriting logic. Rewriting logic is a logic of concurrent change that can naturally deal with state and with concurrent computations. Therefore, it is easy to define new logics, like temporal logics, in Maude.

The Maude rewriting engine can be used as:

- A monitoring engine during program execution. In JPaX, execution events are submitted to the Maude program that evaluates them against the requirement specification.
- Translation engine before execution. In JPaX, the Specification is translated into a data structure optimal for program monitoring. This data structure is sent back to Java and used within the JAVA program to check the events during execution.

For more information about Maude see http://maude.csl.sri.com or http://ase.arc.nasa.gov/havelund *Monitoring Programs Using Rewriting.*

15. APPENDIX F: TEMPORAL LOGIC

Temporal Logic is a form of logic. It is different from prepositional or first-order logic which reason about a given state "of the world". Temporal logic provides the capability to reason about how different states follow each other over time.

Time can be described in a linear fashion (Linear Temporal Logic) or a branching fashion (Computational Tree Logic) as shown in Figure 11: Temporal Logic. For example, temporal logic provides a way to express statements like "the engine will never overheat" or "all faults will eventually be detected". [39]

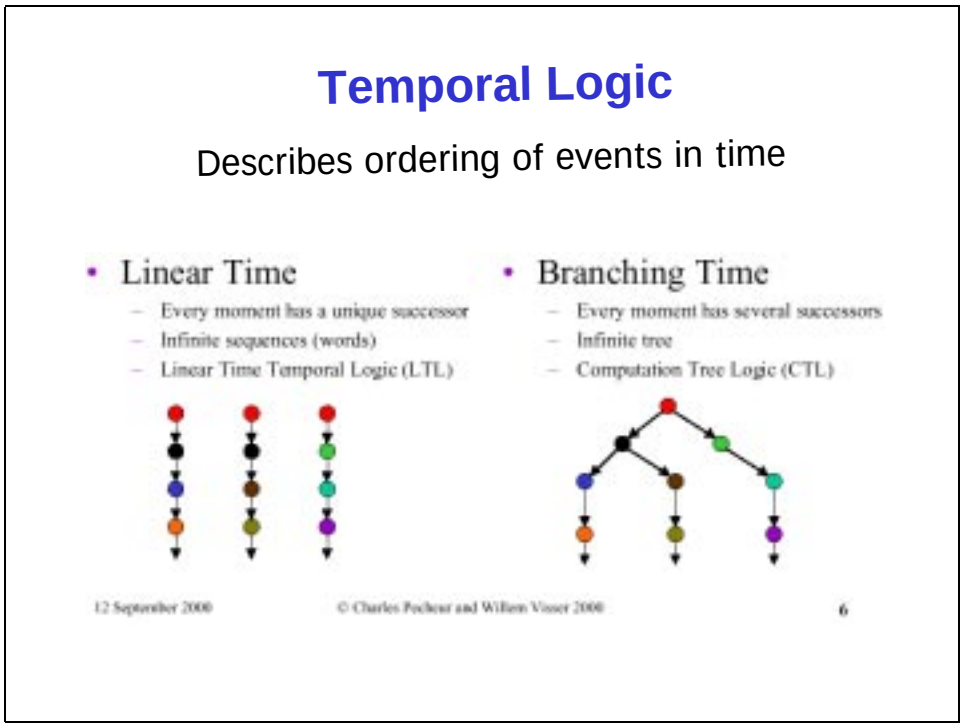


Figure 11: Temporal Logic [40]

The difference between LTL and CTL is how they handle branching in the underlying computation tree. LTL focuses on describing events along a single computation path; whereas, CTL considers all paths possible for a given state. [9] The SMV tool uses CTL and an example of CTL is included in Figure 12: Computation Tree Logic (CTL) [40]

The CTL formulas in Figure 11 are composed of path quantifiers and temporal operators. Two path quantifiers describe the branching structure in the computational tree:

- A – ALWAYS (all computational paths)
- E – SOMETIMES (some computational path)

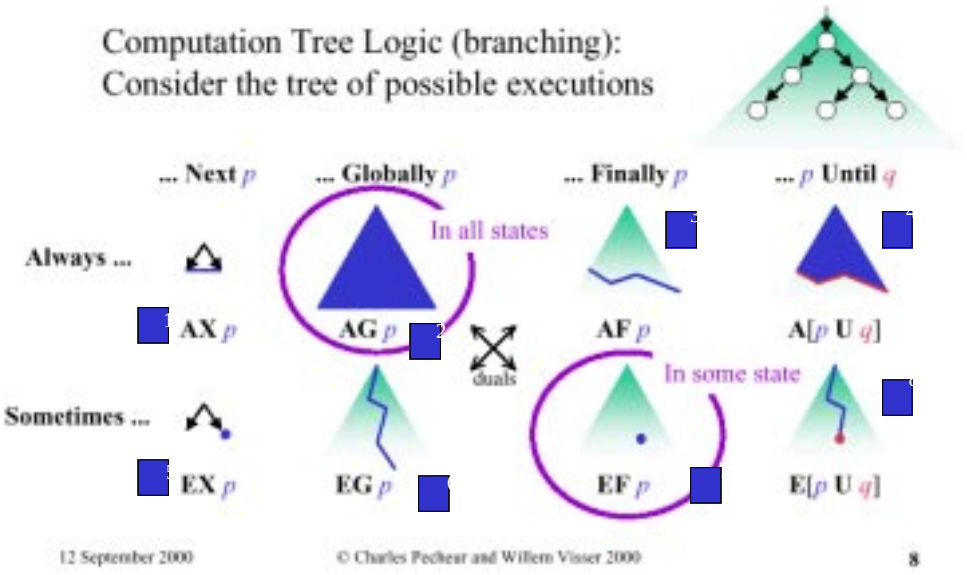
Five basic temporal operators describe properties of a path through the tree:

- X – Next (requires that a property hold in the next state of the path)
- G – Always or Globally (requires that a property hold at every state)
- F – Future or Finally (requires that a property hold for some state in a path. It is the dual of the G operator)

- U – Until (requires that a second property holds and the first property will hold at every preceding state)
- R – Release (requires that a second property hold along a path up to and including the first state where the first property holds. However, the first property is not required to hold. It is the logical dual of the U operator) [9]

LTL has the same temporal operators as CTL, but does not contain path quantifiers, A (ALWAYS) and E (SOMETIMES) because LTL considers time to be linear rather than branching.

Computation Tree Logic (CTL)



1. The property AXp states that the property p will hold at all next states
2. The property AGp states that the property p will hold forever on all execution paths (i.e. ALWAYS)
3. The property AFp states that the property p will occur for all states at some point
4. The property A[pUq] states that the property p will occur until q occurs
5. The property EXp states that the property p will occur in some next state
6. The property EGp states that the property p will occur for some states
7. The property EFp states that the property p will occur in some state
8. The property E[pUq] states that the property p will occur in some states until q occurs

Figure 12: Computation Tree Logic (CTL) [40]

16. REFERENCES

[1] Interview with Guillaume Brat, NASA ARC ASE, November 30, 2001

[2] Charles Pecheur. "Verification and Validation of Autonomy Software at NASA." NASA/TM 2000-209602, August 2000. Submitted to Automated Software Engineering Journal. http://ase.arc.nasa.gov/pecheur/publications

[3] 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. http://ase.arc.nasa.gov/pecheru/publi.html

[4] Reference not used

[5] John Rushby. Disappearing Formal Methods slide 38 in Assurance for Dependable Systems (Disappearing Formal Methods) presented at Safecomp, Budapest, September 2001, TU Vienna, March 2001 and NSA March 2001.

[6] Klaus Havelund, Grigore Rosu. *Monitoring Java Programs with Java PathFinder.* To be published in Electronic Notes in Theoretical Computer Science.

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

[8] *Static Verification of ANSI C Software Using PolySpace C Verifier.* PolySpace Technologies. http://www.polyspace.com

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

[10] Microsoft Solution Framework Instructors Presentation, Principles of Application Development

[11] Guillaume Brat and Willem Visser. *Combining Static Analysis and Model Checking for Software Analysis.* In Proceedings of the 16th International Conference on Automated Software Engineering. November 2001. Coronado Island, California. pp 262-269.

[12] Email dated January 14, 2002 from Dimitra Giannakopoulou, NASA ARC ASE

[13] Kesten, Y. and Pnueli, A. "Modularization and Abstraction: The Keys to Formal Verification", in *Proc. of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS 1998)*. 1998. Springer, Lecture Notes in Computer Science 1450, pp. 54-71. L. Brim, J. Gruska, and J. Zlatuska, Eds.

[14] Patrice Godefriod, Robert S. Hanmer, Lalita Jategaonkar Jagadeesan. *Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft.* Bell Laboratories and Lucent Technologies

[15] William Chan. Temporal-Logic Queries. Proceedings of CAV 2000, pp. 450-463. Lecture Notes in Computer Science, vol. 1855, Springer, 2000.

[16] Edmund Clarke, Armin Biere, Richard Raimi and Yunshan Zhu. *Bounded Model Checking Using Satisfiability Solving.* www.inf.ethz.ch/~biere/papers/ClarkeBiereRaimiZhu-FMSD2001.pdf

[17] Reference not used

[18] Deep Space One Website: http://nmp.jpl.nasa.gov/ds1/

[19] Khatib, L., Muscettola, N., and Havelund, K., *Verification of plan models using UPPAAL*, First Goddard Workshop on Formal Approaches to Agent-Based Systems, Greenbelt, Maryland, 2000. (to appear in a special issue of Lecture Notes in Computer Science)

[20] Reference not used

[21] 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. (Note: for information regarding reprints of this article, please send email to tse@computer.org and reference IEEECS Log Number 107207)

[22] Reference not used

[23] Reference not used

[24] *NASA Procedures and Guidelines NPG: 2820.DRAF1, NASA Software Guidelines and Requirements as of 3/19/01* (Responsible Office: Code AE/Office of the Chief Engineer), NASA Ames Research Center, Moffett Field, California, USA

[25] N. Fenton. "How Effective Are Software Engineering Methods". J. of Systems and Software 22:141-146, 1993. Found in reference [32]

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

[27] 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

[28] 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.

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

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

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

[32] *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion*, National Aeronautics and Space Administration, December 1998

[33] R. W. Butler, *An Elementary Tutorial on Formal Specification and Verification Using PVS*, NASA Technical Memorandum Number 108991, June 1993.

[34] D. Weber-Wulf, "Proof-Movie—A Proof with the Boyer-Moore Prover, in *Formal Aspects of Computing*, 5(2):121–151, 1993.

[35] J. B. Wordsworth, *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, Ltd., 1992.

[36] S. P. Miller and M. Srivas, *Formal Verification of a Commercial Microprocessor*, Technical Report SRI-CSL-95-4, SRI Inter-national, Menlo Park, CA, February 1995.

[37] J. P. Bowen and M. G. Hinchey, *Applications of Formal Methods*, Prentice-Hall International, Ltd., 1995.

[38] *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume II: A Practitioner's Companion*, National Aeronautics and Space Administration, December 1998

[39] Charles Pecheur, Reid Simmons, Peter Engrand. *Formal Verification of Autonomy Models From Livingstone to SMV.* August 31, 2001. Submission to FAABS (Formal Approaches to Agent-Based Systems).

[40] Model Checking for Software. Tutorial presented at the ASE'00 conference, Grenoble, France, 12 September 2000 (with Willem Visser)