

# Intelligent Software Engineering Tools for NASA's Crew Exploration Vehicle

Michael Lowry

NASA Ames Research Center, Moffett Field, CA, 94035

[Michael.R.Lowry@nasa.gov](mailto:Michael.R.Lowry@nasa.gov)

**Abstract.** Orion is NASA's new crew exploration vehicle. The Orion Project will be using a state-of-the-art model-based software development process. This model-based software development process is new for the human space program, and implies both new opportunities and risks for NASA. Opportunities include gaining early insight into designs in the form of executable models, and formulation of requirement verification conditions directly at the model level. Risks include auto-generated code. This paper describes intelligent software engineering tools being developed by NASA. The tools interface directly to the model-based software development process, and provide the following capabilities: early analysis to find defects when they are inexpensive to fix, automated testing and test suite generation, and innovative methods for verifying auto-generated code.

## 1 Prologue

Following the termination of the Apollo program, the human space program has for decades remained confined to low-earth orbit even as robotic vehicles explored the other planets and moons of our solar system. After the disaster of the space shuttle Columbia breaking up over North America, the United States reassessed its human space program. Instead of retreating after the Columbia disaster, NASA was directed to reach again beyond the confines of Earth and to acquire the capability for human exploration of our solar system – and beyond. The first step is to replace the space shuttle with a crew exploration vehicle that is suitable for travel beyond low-earth orbit. This is the objective of the Orion project.

The space shuttle, originally conceived of as a space ferry, was an engineering marvel for the 1970s when it was designed and developed. The avionics alone set many precedents, including digital fly-by-wire and the use of fault-tolerant configurations of general-purpose computers. The core software combined a real-time operating system tightly coupled to cross-checking of computations in a configuration of four computers running identical software and a backup system running dissimilar software to provide real-time computing fault tolerance. This was needed to ensure two-fault tolerance required for digital fly-by-wire.

However, the operation of the space shuttle has never achieved its goal of routine, cost-effective, airline-style operation. In addition, a crew vehicle with a large cargo capacity requires significant expense to meet human-rating requirements as compared

to a simpler and smaller crew-only vehicle augmented with separate cargo vehicles. In retrospect, the space shuttle is also more dangerous than originally thought.

NASA plans to replace the space shuttle with Orion, a capsule that has been likened to ‘Apollo on Steroids’. It is suited for travel beyond low-earth orbit, while avoiding dangerous design aspects of the space shuttle. Since it is much smaller and less massive than the shuttle, while carrying a comparable crew, it will be significantly less expensive to launch and operate. NASA selected Lockheed as the prime contractor, in part because of an innovative model-based software development process. The proposed core of this process are auto-coders that translate from Unified Modeling Language (UML) and related modeling languages to code, and also expected reuse of portions of the Boeing 787 software.

NASA has limited experience in the human spaceflight program with these methods of software development. However, NASA research centers have over the past decade been developing intelligent software verification and validation technologies. These technologies are now being interfaced to the software development process for Orion. This paper first overviews the expected software development process for the Orion project, and then describes the software verification and validation technologies developed by NASA research centers. Cited papers provide more technical detail than is possible in this paper. The paper also describes how these technologies are being adapted to Orion’s software development process, in order to provide NASA better capabilities for oversight.

## 2 Orion Software Development

Lockheed has chosen an innovative software development process for Orion with a high degree of tool support from requirements development through auto-generation of code. The current software development plan calls for a *model-driven architecture* (MDA), where the design is developed primarily with a UML (Unified Modeling Language) tool, with secondary support from Mathworks’ modeling languages. The Mathworks languages include Matlab, Simulink, Stateflow, and a variety of tool-boxes; these will be collectively called Matlab in this paper. The implementation in C++ will be generated automatically through an adapted version of a commercial auto-coder for UML, in combination with a Matlab compiler. The commercial UML auto-coder is designed to be modified by a development organization in order to target a specific operating platform. For the Orion project, the operating platform is a real-time operating system that is compliant with ARINC 653 standards for Integrated Modular Avionics. Integrated Modular Avionics provides the capability for different *partitions* to execute on the same computer, as if they were executing on separate computers; thus providing a level of safety that would otherwise require the power, weight, and costs associated with multiple computers.

This software development process is new to NASA human space exploration. For flight software design and development the model-driven architecture approach is expected to decrease the dependency on target hardware, programming language and architecture. Since Orion is expected to be a core component of NASA’s fleet for decades, it is important to plan ahead for future upgrades to the avionics. The intent of the MDA is to enable redirecting the software to another target platform by changing

the auto-coder and then re-generating the flight software from the models for the new platform. For this to be effective, the intent is for all software (except a fixed reuse core) to be generated ‘pristinely’: no hand modification after auto-coding. This redirection through changing the auto-coder will also enable the same models to be used for simulation and training software. Compared to previous baselines this approach is also expected to decrease cost and schedule, reduce coding defects, simplify integration issues, and support rapid development through model simulation and debugging.

NASA research centers are developing intelligent analysis tools that can be used for NASA oversight, and are adapting them to this innovative software development process. One objective is to verify that the software works correctly over a wide range of both nominal and off-nominal scenarios. Doing this analysis at the model level has advantages in both finding defects early and in scaling the analysis to the large software systems expected for Orion. There are two related objectives to this model analysis. The first is to thoroughly exercise the different execution paths through the software. This objective has been extended to white-box testing: automating the generation of suites of test cases that provide coverage for exercising the execution paths through the software. The second related objective extends this to black-box testing technology: clustering the behavior of the system under simulation according to nominal and off-nominal behavior, and automatically testing the system over a wide range of mission parameters in order to determine governing factors between nominal and off-nominal behavior. The final objective described in this paper is to independently verify the output of the auto-coder for compliance with safety policies, and to generate documentation suitable for humans doing technical reviews.

In order to explain the expected use of these tools, the following sub-sections provide a synopsis of the UML/MDA software development approach. A key observation is that this approach naturally results in system software structured as a set of interacting finite state machines. This structuring facilitates the scaling and use of model-checking technology for model analysis. A second observation is that the adaptable auto-coding approach itself provides a means for interfacing between UML models and model-checking technology: the auto-coder is adapted to the model-checking platform. The core model-checking platform is based on a virtual machine technology that further facilitates this adaptation.

## **2.1 Model-Driven Architecture**

The Object Modeling Group (OMG) defines model-driven architecture as the separation of system application logic from the underlying platform technology. This enables platform-independent models of a system’s function and behavior, built using UML, to be implemented on many different target platforms. The platform-independent models represent the function and behavior of a software system independent of the avionics platform (encompassing hardware, operating system, programming language, etc.) used to implement it. Platform dependence has been a perennial challenge for long-lived aerospace systems, such as the International Space Station (ISS). For example, the upgrade from Ada 83 to Ada 95 for the ISS required a substantial re-engineering effort. Orion is expected to be used through the Lunar Sor-

tie and Lunar Habitat phases of NASA's exploration program, and likely into Martian exploration.

## 2.2 Software Development with UML

This subsection simplifies the relevant aspects of UML-based software development from the viewpoint of the NASA analysis tools. The interested reader can find more details in the extensive literature on UML and object-oriented design including the UML variant chosen by the Orion project – executable UML (xUML) [1].

The UML software development process begins by grouping concepts into semi-independent domains that are largely self-cohesive. A domain reflects subject matter expertise, for example Guidance, Navigation, and Control (GN&C). Within each domain, the *classes* of objects and their static relationships are then defined in a class diagram. Each object class then has attributes defined, for example in GN&C a vehicle class would have attributes attitude and velocity. The static relationships between object classes are also defined, for example spacecraft are a specialization of vehicle.

Once the static class diagrams are defined, then the dynamic aspects of the domain are defined, such as *operations*, which are procedures that operate on objects and related objects. Of particular interest are *statecharts* that define the transition of an object through a succession of states. For example, an Orion capsule could be modeled as an object of class *spacecraft* that has states including pre-launch, launch, various phases of ascent, orbit insertion, docking with the space station, etc. Most objects transition through their states through interaction with other objects; such as the launch pad infrastructure signaling Orion that the countdown has reached zero, and Orion in turn signaling the rocket booster to ignite. These interactions are modeled through formal definitions of signals between state machines. UML allows a rich variety of actions that can be taken when signals are sent and received, and the objects enter, remain in, and exit states. Some variants of UML allow these actions to be defined in a conventional programming language. The xUML variant chosen by Lockheed defines these actions through an Action Semantics Language (ASL), which is then auto-coded to a programming language.

Verification of interacting state machines against logical and temporal properties is done through model-checking. For Orion software, the operations and actions that define the behavior of the xUML state machines will be auto-coded to an object-oriented programming language. Thus the model-checker needs to handle object-oriented software in a manner that permits scaling to large software systems.

## 3 NASA Analysis Tools and their Interface to the Orion Software Development Process

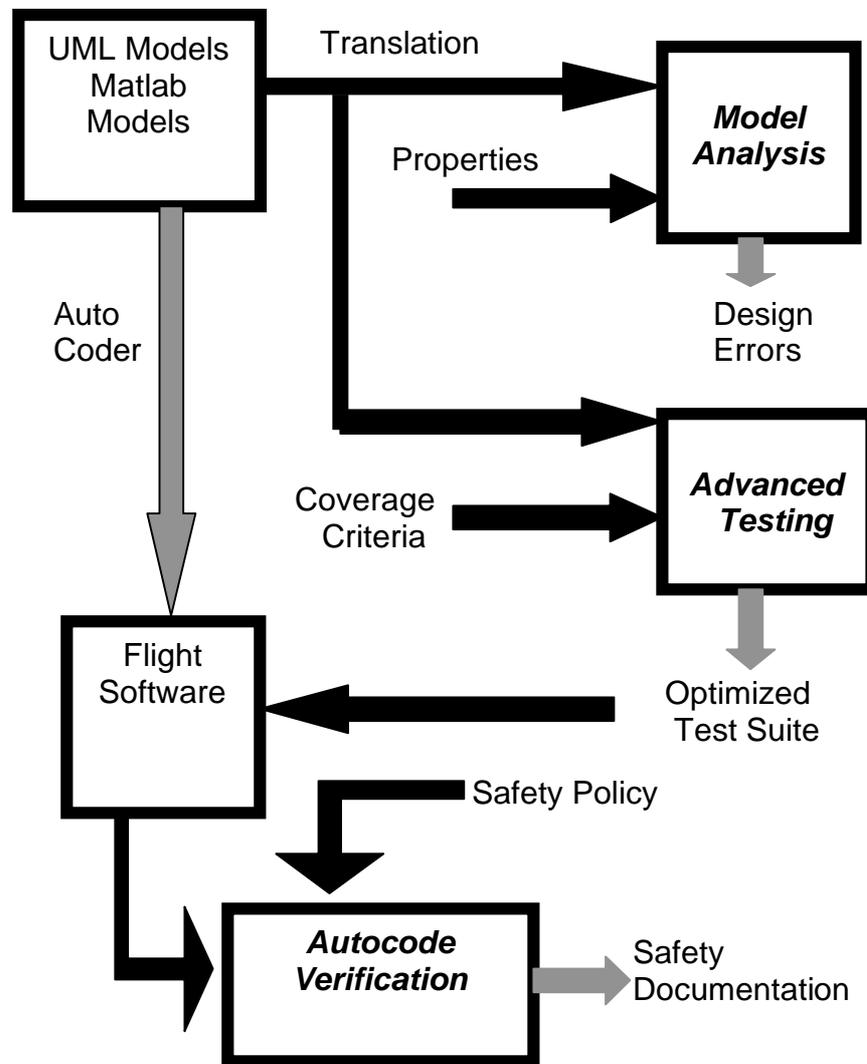
NASA's intelligent analysis tools for model-based software development draw upon core intelligent system methods including automated symbolic reasoning, constraint solving, intelligent search, automated theorem proving, and automated data analysis. The tools interface directly to Orion's model-based software development process (MDA/UML development process), providing the following capabilities:

- Early analysis at the model level to find defects when they are inexpensive to fix. The model-based analysis does a sophisticated graph search through the state space of a software system formulated in UML to automatically find defects against required properties.
- Automated testing and test suite generation that ensures coverage. This includes verification testing based upon white-box coverage criteria of paths through the software, and validation testing for black-box determination of the robustness of the system under varying mission parameters. The white-box testing relies on symbolic reasoning and constraint solving, while the black-box testing relies on intelligent search and machine data analysis.
- Innovative methods for verifying auto-generated code. The auto-code verification independently checks the C++ code against safety policies, and also develops detailed documentation that can be used during a code review. This mitigates the trust that needs to be put into the auto-coder.

On the left of figure 1 below is a fragment in the middle of Lockheed's Orion tool chain: an auto-coder that maps from UML models and Matlab models to C++ flight software. This fragment is preceded by requirements and design steps, and succeeded by compilation to partitions in the target ARINC 653 platform, and many stages of testing. The later stages of testing on the target hardware with a simulated mock-up of the Orion environment are expensive and require unique facilities, so it is important that defects are found as early as possible in the process to avoid costly rework downstream. All of the steps in the tool chain are embedded in a configuration management system. On the right of figure 1 are the NASA intelligent analysis tools, described next in the following sub-sections.

### 3.1 Model Analysis

*Model Analysis* compares the UML and Matlab models against properties specifying verification conditions and then generates design error reports in the form of counterexamples. The models are translated to the model-checking target environment through an alternative auto-coder in the same family of commercial auto-coders that are adapted to generate the flight, training, and simulation software. The model-checking target environment is Java Pathfinder [2,3]: a Java virtual machine that is optimized for state-space exploration through dynamic execution of software encoded as Java byte code instructions. Java Pathfinder dynamically checks a software system against both generic properties (such as consistency and completeness of transitions) and specified verification properties. The verification properties are derived from requirements and design specifications.



**Fig. 1.** On the left, a portion of Lockheed's Orion software development process. On the right, NASA's intelligent software analysis tools.

Java Pathfinder is an explicit state model checker with some similarity to SPIN [4]. Explicit state model checkers use an explicit representation of individual states. They run a backtracking search over the graph of the state space, saving state information for matching on subsequent nodes in the search. If a new node matches a previously generated node, then the search from the new node is terminated. For software, the transitions correspond to instruction executions, and the states correspond to the value of program variables across all threads and processes. Explicit state model checkers are particularly good for asynchronous concurrent systems and reactive embedded systems, where the system interacts with the environment. The environment is itself modeled as a finite state machine, e.g., a software simulation, operating in parallel with the system.

Orion software falls into the category of a reactive embedded system with rich constructs (ASL) for the transition relations, and complex states corresponding to Object-oriented (OO) software. Java Pathfinder was built from the ground up to handle object-oriented (OO) software; it has highly effective means of compactly representing state information for OO software executions. In addition, UML's use of state machines for modeling the dynamics of a system provide an abstraction layer that has been modeled as an extension of the Java virtual machine at the core of Java Pathfinder; providing further computational savings in analyzing Orion software. There is a considerable literature on methods for handling the computational complexity of model-checking, many of these methods are built-in to Java Pathfinder.

The architecture of Java Pathfinder also enables changing the operational semantics of the byte-code instructions on the fly; the default semantics are the same as standard Java virtual machines. This flexibility enables plugging-in operational semantics for other analysis tools, which is used for verification testing.

### **3.2 Advanced Verification Testing – Test Suite Generation**

There are two components to advanced testing: verification testing and validation testing. Verification testing is done through an alternate operational semantics with an extension of Java Pathfinder [5]. Instead of executing the byte codes on concrete inputs, the byte codes are executed on symbolic inputs. The result is to accumulate symbolic expressions denoting the value of variables at successive points in the program execution. When a conditional statement is encountered, the condition becomes a symbolic constraint on variables. For a simple if-then-else statement, a path condition is generated on the variables of the conditional expression for the then branch, and the negation of the path condition is generated for the else branch. The symbolic execution accumulates path conditions through nested conditionals and guarded iterative constructs; the constraints in the accumulated path conditions are then solved to generate test vectors. This is done for the different paths and conditions in order to ensure that the generated test vectors provide sufficient coverage, as described below.

Verification testing takes as input the same translated models as model analysis, and in addition a specification of desired coverage criteria. Examples of coverage criteria include statement coverage (execute each program statement at least once), and branch coverage (execute all combinations of conditional branches, taking into ac-

count nesting structure). Symbolic execution [6,7] then determines the cumulative path conditions for the different points in the code required for the specified coverage. For each such point in the code, a vector of input values are then generated that satisfy the constraints. The result is a set of vectors that comprise a test suite that provides the specified coverage criteria. This set of vectors can be optimized to eliminate duplication. The test suite can be applied at many different stages of testing from the model-level through high-fidelity hardware testing. Verification testing as described here has mainly been applied to automatically generating test suites for individual units. Methods for extending it to larger portions of software are under development, including innovative approaches to integrating with system-level validation testing.

### **3.3 Advanced Validation Testing**

The software validation testing technology performs large validation test runs (optionally on computer clusters or supercomputers) of a system under simulation, followed by automated machine analysis of the test logs to cluster the results, identify anomalous runs, and identify predicates on the inputs that separate nominal from off-nominal runs. It largely automates the laborious and expensive traditional process of human validation testing and test-log analysis, filtering the large amount of data to something which is much more manageable by human engineers. It thus enables validating a simulated system over a much wider range of scenarios, defined by variations in mission parameters, thereby providing assurance that the system does what is needed and more sharply defining under what range of off-nominal parameters the system is no longer robust. Conceptually, it is an extension of Monte-Carlo analysis – where a simulated system is tested under a statistical distribution of parameters, leading to a scatter-plot of results.

The innovation over standard Monte-Carlo analysis is two-fold. First, combinatorial testing techniques are used to factor pairs or triples of parameters into varying sets in order to identify critical parameters. This mitigates the combinatorics of the number of tests, which then allows searching over a much larger range of statistical variation than the two or three sigma that is standard for Monte Carlo testing. The objective of this extended variation is to determine system robustness through explicit identification of failure boundaries. The coverage criteria for advanced validation testing are specified through settings for the combinatorial and Monte-Carlo test generation. The second innovation is the automated machine analysis of test runs. Clustering algorithms based on expectation-maximization are generated automatically through AutoBayes [8] – a program synthesis system for data analysis. Treatment learning provides learning predicates on parameters that separate nominal from off-nominal behavior.

This technology has already been applied to analysis of ascent and re-entry simulations for Orion [9]. At present, several iterations with manual input are required to identify factors that determine off-nominal behavior – e.g., atmospheric re-entry points that lead to off-range landings. The manual input is to adjust test generation ranges and factoring. Under development are methods to automate the adjustments for successive iterations based on machine data analysis.

### 3.4 Autocode Verification

Auto-coders are a critical component of Orion's Model-Driven Architecture, enabling avionics platform re-targeting over the decades-long expected lifetime of Orion. While auto-coders have historically been used for rapid prototyping and design exploration, their use in safety-critical domains is more recent. One approach in safety-critical domains is to treat the auto-generated code as if it were manually developed for purposes of verification and test, however this approach provides only limited immediate productivity gains over manual development. Another approach is to *qualify* the code generator, which requires the same certification standards as the production flight code but enables analysis activities on the model-level to receive formal verification credit. However, code generator qualification is expensive and needs to be completely redone for every upgrade to the auto-coder and every adaptation and reconfiguration for a project.

A third approach is to exploit information about the auto-coder to automate portions of the analysis and documentation needed for certifying the generated code. This is the approach being taken with the NASA autocode verification tool. Certain aspects of the safety certification for flight-readiness are being automated, through a Hoare-style verification of auto-generated code against safety policies, with automated generation of detailed documentation that can be used during a code review. The technology has already been demonstrated on a number of safety policies relevant to Orion, including programming-language safety conditions (e.g., variable initialization before use, and memory access safety) to domain-specific safety conditions (e.g., correct handling of physical units and co-ordinate frames [10]). The documentation of conformance with safety policies is generated after an automated, formal proof that the code meets the safety requirements, and provides an understandable hyperlinked explanation suitable for code reviews.

The algorithm synopsis [11,12] is to work backwards through the generated code from a safety postcondition, first heuristically generating annotations that exploit information about the autocoder (such as stylized code fragments that encode coordinate transformations), and then collecting verification conditions. These are then submitted to an automated theorem prover, which typically succeeds in discharging the verification conditions. The approach is fail-safe, in that the annotations are not trusted and if incorrect will lead to failure to prove the verification condition. The approach is loosely motivated by proof-carrying code.

## 4 Summary

NASA's Orion project is using an MDA software development approach for NASA's new crew exploration vehicle. NASA research centers have adapted intelligent software engineering analysis technology to this model-based approach in order to provide tools for insight and oversight of Orion software development. The tools include model-checking to find design errors in UML and Matlab models, verification testing technology that automatically generates test suites providing white-box testing coverage for units and subsystems, validation testing to find mission parameter ranges that distinguish nominal versus off-nominal behavior, and autocode verification technol-

ogy that automates aspects of the safety certification of the MDA auto-generated code.

**Acknowledgement:** The author wishes to acknowledge the many years of work by members of NASA's Intelligent Software Design project, whose present members at NASA Ames include Ewen Denney, Karen Gundy-Burlet, Masoud Mansouri-Samani, Peter Mehlitz, Corina Pasareanu, Thomas Pressburger, and Johan Schumann.

## References

1. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilke, I.: Model Driven Architecture with Executable UML. Cambridge University Press, Cambridge (2004)
2. Java PathFinder; <http://javapathfinder.sourceforge.net>
3. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal 10(2) (2003)
4. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Menlo Park (2003)
5. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: Proc. of the 13<sup>th</sup> International TACAS Conference (2007)
6. Kurshid, K., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proc. of the 9<sup>th</sup> International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2003)
7. Pasareanu, C., Visser, W.: Symbolic Execution and Model Checking for Testing. Invited talk for Haifa Verification Conference (2007)
8. Fischer, B., Schumann, J.: AutoBayes: A System for Generating Data Analysis Programs from Statistical Model. J. Functional Programming 13(3), 483--508, (2003)
9. Gundy-Burlet, K., Schumann, J., Barrett, T., Menzies, T.: Parametric Analysis of Antares Re-Entry Guidance Algorithms using Advanced Test Generation and Data Analysis. In: Proc. 9<sup>th</sup> International Symposium on Artificial Intelligence, Robotics, and Automation in Space, (2008)
10. Denney, E., Trac, S.: A Software Safety Certification Tool for Automatically Generated Guidance, Navigation and Control Code. In: Proc. IEEE Aerospace Conference (2008)
11. Denney, E., Fischer, B.: A generic annotation inference algorithm for the safety certification of automatically generated code. In: Proc. GPCE '06: 5th International Conference on Generative Programming and Component Engineering (2006)
12. Denney, E., Fischer, B.: Extending Source Code Generators for Evidence-based Software Certification. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (2006)