# Verification and Validation of Autonomy Software at NASA

Charles Pecheur, RIACS / NASA Ames Research Center
pecheur@ptolemy.arc.nasa.gov

(Submitted to Automated Software Engineering Journal)

**Abstract**

Autonomous software holds the promise of new operation possibilities, easier design and development and lower operating costs. However, as those system close control loops and arbitrate resources on-board with specialized reasoning, the range of possible situations becomes very large and uncontrollable from the outside, making conventional scenario-based testing very inefficient. Analytic verification and validation (V&V) techniques, and model checking in particular, can provide significant help for designing autonomous systems in a more efficient and reliable manner, by providing a better coverage and allowing early error detection. This article discusses the general issue of V&V of autonomy software, with an emphasis towards model-based autonomy, model-checking techniques and concrete experiments at NASA.

## 1 Introduction

NASA's mission of deep space exploration, coupled with Administrator Goldin's directive to do it "faster, better, and cheaper", has created an exciting challenge for the computer science research community: that of designing, building, and operating smart, adaptable, and self-reliant autonomous spacecraft, rovers, airplanes, and perhaps even submarines, capable of coping with harsh and unpredictable environments. As those robotic explorers continue to explore Mars and beyond, the great distances from Earth will require that they be able to independently perform not only navigation tasks and self-diagnosis, but also an increasing amount of autonomous or semi-autonomous on-board science. For example, the Autonomous Controller for the In-Situ Propellant Production facility, designed to produce spacecraft fuel on Mars, must operate with infrequent and severely limited human intervention to control complex, real-time, and mission-critical processes over periods of months or years in poorly understood environments [11].

While autonomy offers promises of improved capabilities at a reduced operational cost, there are concerns about being able to design, implement and verify such autonomous systems in a reliable and cost-effective manner. This article discusses the general issue of V&V of autonomy software, with an emphasis towards model-based autonomy, model-checking techniques and concrete experiments at NASA. Section 2 introduces autonomy software, section 3 discusses how they can be verified, section 4 focuses on verification of model-based autonomy, section 5 presents further applications in verification of autonomy at NASA, and section 6 draws conclusions.

# 2 Autonomous Systems

## 2.1 The Need for Autonomy

Though information technology is taking a more and more important part in our everyday life, we still depend heavily on human intelligence and adaptability when it comes to responding to unforeseen circumstances. Even highly automated systems such as nuclear plants, power distribution networks or assembly lines rely on human operators in critical or anomalous situations. This is particularly true at NASA, where space missions are still almost entirely controlled by human operators on earth. The successful safe return of the crew of Apollo 13 is a famous example where the intelligence of the crew and the huge ground support team were essential in saving the mission with the limited available resources. More automation has been introduced since then, but the tasks performed by software in space missions are rudimentary, and every shuttle, spacecraft or space station module in operation has a full team of highly trained engineers constantly monitoring its health from Earth.

The limited use of software is justified by the very high reliability requirements for space technology: it is very difficult to design, develop and validate software that provides the needed functionalities. However, ongoing progress in software technologies, coupled with the exponential growth of computer performance, now make it possible to go towards more autonomous systems, where a larger part of the control is delegated to autonomy software.

There are two main reasons driving the development of autonomy software: one is budgetary, the other is technical. On the budgetary side, autonomy will reduce the need for human attendance, which is a major cost factor at NASA and elsewhere – this is the same incentive that has driven the ubiquitous appearance of automation in less critical tasks, from dial phones to on-line shopping. Even more importantly, though, autonomy reduces the reliance on the communication link between the system and its operators, opening a full new range of opportunities:

- In space, information takes more and more energy and time to reach its destination as distance increases – up to 20 minutes from Earth to Mars. Local autonomous control software will enable much faster reactions. This can increase productivity, enable new missions or even save the life of a spacecraft.
- Even with negligible communication delays, autonomy can provide computer-speed reaction times in places where human response time would be too slow with respect to the environment, as for example in collision avoidance systems.
- Autonomy also continues to work when no communication is possible at all, because of interference or physical obstacles. In planetary missions, this happens when a spacecraft passes behind a planet; in earth-bound missions, this enables deep underground or submarine explorations.

## 2.2 Model-Based Autonomy

In its simplest form, autonomy software consists in control sequences that allow the controlled system to achieve its particular goal while tolerating a certain amount of uncertainty in its environment. To build this code, the software engineer must use his understanding of the system to anticipate execution scenarios for all the possible combinations of events that may arise, both in the system and in its environment. The development and validation of such scenario-based code

becomes very difficult as the system becomes more complex, due to the combinatory explosion of the number of situations to be handled.

*Model-based reasoning* (MBR) uses artificial intelligence techniques to automate the inference of those scenarios, using an abstract, declarative model of the system, as shown on Figure 1. Applying efficient reasoning rules to this model and to sensor data, an MBR system can infer information about the current state of the physical system, and build sequences of actions to drive it to a desired configuration, even in situations that were not anticipated at design time.

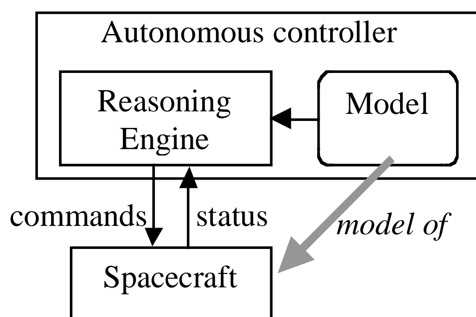


*Figure 1. Model-based autonomous controller*

Instead of using a static, human designed model for controlling a system, some innovative approaches are based on a dynamic model that is progressively improved through a learning process. The initial learning phase is usually done before the system is put in service, but some further adaptation can be done while in operation, for example as the system learns about new unanticipated working conditions. In particular, adaptive techniques include neural networks or genetic algorithms.

### 2.3 Example: Remote Agent

The Remote Agent (RA) is an autonomous spacecraft controller developed by NASA Ames Research Center conjointly with Jet Propulsion Laboratory [18]. Remote Agent comprises three parts, two of which are model-based (Figure 2):

- The Planner and Scheduler (PS) [19] generates flexible plans, specifying the basic activities that must take place. Given a mission goal, such as taking a picture of an asteroid, the planner/scheduler uses a model of the spacecraft's resources to produce sequences of tasks for achieving this goal.
- The Smart Executive (EXEC) [22] receives the plan from the planner/scheduler and commands spacecraft systems to take the necessary actions.
- The Mode Identification and Recovery (MIR), based on the Livingstone model-based health management system [24], uses another model describing both the nominal and failure modes of the different components of the spacecraft. By comparing the observed real state with the state predicted by the model, Livingstone can detect and diagnose failures and suggest recovery actions to the executive.

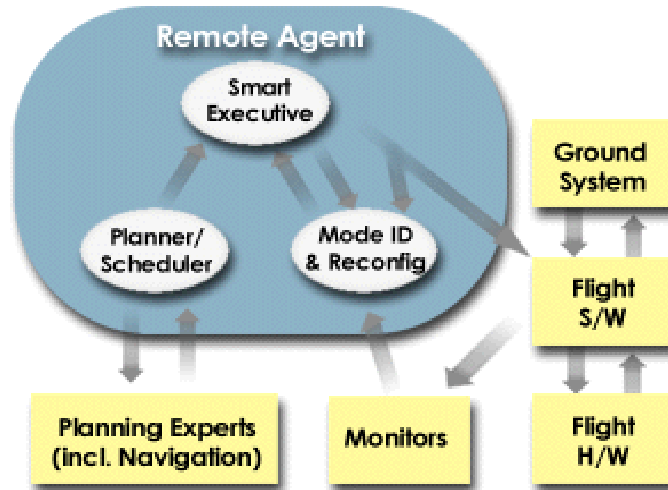


*Figure 2. Remote Agent*

Remote Agent was demonstrated in flight on the Deep Space One mission (DS-1) in May 1999, marking the first control of an operational spacecraft by AI software [20].

# 3 Verification of Autonomy Software

## 3.1 Scenario-Based Testing

Typical software development models are staged into several phases: requirements capture, design, implementation, verification, deployment, maintenance. We are mainly concerned about verification[1], which role is to assert that the implementation performs as expected. Usually, the verification phase is done using *scenario-based testing*. The software component to be verified is embedded into a test harness that connects to the inputs and outputs of that component, and drives it through a suite of *test runs*. Each test run is an alternated sequence of provided inputs and expected outputs, corresponding to one scenario of execution of the tested component. An error is signaled when the received output does not meet the expected one.

Even for simple systems, 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. Running the tests is also a time-consuming task, because the whole program code has to be executed and everything must be re-initialized before each test run. In the development of complex systems, it is quite common that testing the software actually takes more resources than developing it.

---

[1] More generally, one speaks about *verification* and *validation* (V&V), where *verification* checks the implementation against its specification, i.e. "does things right", whereas *validation* checks that the specification itself captures the intended requirements, i.e. "does the right things". This distinction depends on what is considered to be the requirements and the specification; for our purposes, we consider any mechanized evaluation to pertain to verification, and restrict validation to human review with respect to mental concepts. Technically, this paper is about verification, not validation.

In conventional controllers, the code to be tested consists mostly of sequential scenarios, which are activated explicitly by human operators through an open control loop. It is thus quite convenient to attach a test harness to that control channel and build test runs that exercise each of these scenarios.

In contrast, verification of autonomous systems is much more challenging, for several reasons:

- First, autonomous systems close the control loops and arbitrate resources on-board, making it more difficult to plug in test harnesses and write detailed test runs that drive the system through a desired behavior.
- Second, the range of situations to be tested is incomparably larger. In the open loop case, it is up to the intelligence of the experts in control to choose the appropriate response to a situation as it occurs. In the autonomous case, the program implicitly incorporates response scenarios to any combination of events that might occur. The reaction can also depend on the current configuration of the system, or even on its past history in the case of adaptive systems. All these factors multiply exponentially with the size of the system, and a test suite can only exercise a very limited portion of those cases.
- Third, as different concurrent parts of the autonomous controller interact together internally, the controller can now react in different ways to the same stimuli, for example because of differences in scheduling. The consequence is that a successful test run does not even guarantee that the system will behave correctly for the tested scenario, because the same *input* sequence can lead to different *execution* sequences. Another test run could fail due to uncontrollable changes of circumstances. This is a well-known issue in designing concurrent systems.

The recent Remote Agent experiment (RAX) did provide a striking example [20]. After a year of extensive testing, Remote Agent was put in control of NASA's Deep Space One mission on May 17, 1999. A few hours later, RAX had to be stopped after a deadlock had been detected. After analysis, it turns out that the deadlock was caused by a highly unlikely race condition between two concurrent threads inside Remote Agent's executive. The scheduling conditions that caused the problem to manifest never happened during testing but indeed showed up in flight. RAX was re-activated two days later and successfully completed all its objectives.

Note that redundancy, which is the usual solution to increase reliability, is not appropriate for software. As opposed to hardware components, which fail statistically because of wear or external damage, programs fail almost exclusively due to latent design errors. Failure of an active system is thus highly correlated with failure of a duplicate back-up system (unless the systems use different software designs, as in the Shuttle's on-board computers).

### 3.2 Model Checking

*Analytic verification* is the branch of software engineering concerned with establishing, through some mathematically based analysis, that a computer *program* fulfills a formally expressed *requirement*. Two main approaches to analytic verification have been developed:

- *Theorem provers* build a computer-supported proof of the requirement by logical induction over the structure of the program;
- *Model checkers* search all realizable executions of the program for a violation of the requirement.

In principle, theorem provers can use the full power of mathematical logic to analyze and prove properties of any design in its full generality. For example, the PVS system [21] has been applied to many NASA applications (e.g. [6], [3]). However, these provers require a lot of efforts and skills from their users to drive the proof, making them suitable for analysis of small-scale designs by verification experts only. In contrast, model checking is completely automatic, and thus more convenient for verification in on-line software development environments, as opposed to off-line research studies using theorem provers.

The *program* here is some representation of the dynamic, generally concurrent behavior of the application. For tractability reasons, it is usually not the complete code from the implementation but rather some abstract *verification model*[2] of the application, capturing only the essential features that are relevant to the requirements to be checked, expressed in a language that is accepted by the verification tools. Those languages vary among different analytic verification technologies, but the underlying mathematical abstraction is always some kind of transition system: the model defines the *structure of a state* of the system, the set of possible *initial states*, and a *transition relation* defining the allowed moves of the system.

The requirements to be checked can be *invariants* (e.g., a resource cannot be accessed by two processes at the same time), *temporal properties* (e.g., after a process locks a resource it will always eventually release the lock) and even *metric properties* (e.g., the lock will be released within 30 milliseconds). Other approaches work by *comparison between models* (e.g., a distributed cache memory algorithm is equivalent to a local memory model).

In its simplest form, a model checker starts from the initial states and repeatedly applies the transition relation to search all reachable states for a property violation, while remembering explored states to avoid looping. A lot of improvements have been introduced to make this search as flexible and efficient as possible (e.g. partial order reduction [8] or symmetry reduction [15]). When a property violation is found, the model checker reports the execution trace that leads to the violation, which is essential for diagnosing the source of the problem.

*Symbolic model checkers* offer a useful alternative to conventional explicit-state model checking as described above. Instead of generating and exploring every single state, symbolic model checking manipulates whole sets of states at each step, implicitly represented as the logical conditions that those states satisfy. These conditions are encoded into data structures called Binary Decision Diagrams (BDDs) [1], that provide a compact representation and support very efficient manipulations. For example, BDDs for a set of states and for the transition relation can be combined to obtain a BDD for the next set of reachable states. Symbolic model checking can address much larger systems than explicit state model checkers, but does not work well for all systems: the complexity of the BDDs can outweigh the benefits of symbolic computations, and BDDs are still exponential in the size of the system in the worst case. One of the major symbolic model checkers is SMV developed by K. McMillan and E. Clarke at Carnegie-Mellon University (CMU) [2].

Autonomy programs and models have an abstract view of the system they control. The nature of this abstraction, and the corresponding programming or modeling paradigms, can vary according to

---

[2] Not to be confounded with the models used in model-based autonomy, which we will refer to as *autonomy models*. The distinction is mostly one of syntax and purpose; both belong to the same broad family of automata-based formalisms.

the needs of each application. The kind of abstraction used has a critical influence on the complexity and tractability of the verification task:

- *Discrete models* describe the system in terms of transitions between states, where a state describes a stable configuration of the system for some duration of time. Discrete models are usually based on some form of automata.
- *Real-time models* can specify time durations between events, whereas un-timed discrete models only address the order in which transitions occur.
- *Continuous models* represent the continuous change in the system with respect to time, using some form of differential equations.
- *Hybrid models* mix continuous changes and discrete transitions.

Currently available heavy-duty model checkers, such as Spin (Bell Labs) [14] or Murphi (Stanford) [5], are based on discrete un-timed models. Model checking of richer formalisms is still an active field of research: though prototype tools exist for real-time models (Uppaal [16], Kronos [25]) or even hybrid models (Hytech [13]), these tools still do not scale up well to real-size models.

Finally, with adaptive systems that are designed to modify their behavior dynamically, any kind of a priori analytic verification becomes problematic. A solution could reside in run-time incremental verification, where the parts of the system affected by an adaptive re-configuration are verified before the re-configuration is committed. Since the verification has to be performed on-board and during the operation of the autonomous system, this may put very tight time and space requirements on the verification process.

### 3.3 Benefits of Model Checking

Model checkers are particularly well suited for exploring the relevant execution paths of non-deterministic systems with multiple processes running in parallel. Instead of executing the real code, a model checker executes an abstract model in a highly efficient way. Furthermore, the model checker can backtrack to explore alternative paths from a common intermediate state, avoiding the costly reset between test runs. It will automatically detect already explored states, thus exploring all executions exactly once. Finally, it controls the scheduling of concurrent components of the model, and will therefore explore all possible execution sequences even for the same input sequence. For all these reasons, model checking can provide a much better coverage than scenario-based testing, and do so in a much shorter time (millions of states in hours of time).

Model checking can also be applied at an earlier stage in the design, long before a testable implementation is available. The cost for diagnosing and repairing faults grows exponentially as the system is developed: the division bug in early Pentium processors, or the destruction of the first Ariane 5, are well-known illustrations. In contrast, the use of analytic verification techniques[3] allowed Silicon Graphics to design chips that were functional on the first fabrication run, which was unprecedented for the manufacturer. By decreasing the human effort required to find faults and rework software, the software development costs can be reduced and become more predictable.

---

[3] Using symbolic equivalence checking, a simplified form of symbolic model checking that compares the transition relations of two systems [4].

### 3.4 Limits of Model Checking

Model checking is limited by state space explosion: the number of states to be explored grows exponentially in the size of the system. In particular, model checkers do not perform induction, and hence can only verify systems of bounded size, in terms of reachable state space. The major challenge to successful application of model checking is to produce a model that is accurate enough to provide useful information about the system, yet small and abstract enough to produce a state space of finite and tractable size. While this remains the main limiting factor, the state of the art provides tools that can handle very large systems with several million states.

Model checking technology, especially symbolic model checking, is extensively employed in verification of digital hardware, where the limitation to bounded systems is often not a limiting factor. However, software, especially autonomy software, is both qualitatively and quantitatively more complicated than digital hardware. First, computer programs feature complex and often unbounded data structures that induce huge and often infinite state spaces. Second, programs use more elaborate constructs such as dynamic memory allocation, procedure calls, object orientation or dynamic thread creation, that complicate the task of representing and comparing their states in an efficient way.

However, the most complex and time-consuming part in nowadays experiences in model checking of software is not so much in doing the verification but mainly in turning the programs into verifiable models. Typically, no formalized high-level description of the software is readily available; more often than not, the starting point is the program code for the implementation. The verifier thus faces a double challenge. First, he must *translate* his application into the input language of the model checker. Second, he must *abstract* away enough of the original system to obtain a model that will be amenable to model checking within reasonable time and space. Both tasks are arduous and require a deep understanding of both the software being verified and the model checker used to verify it. The net result is that software model checking is currently mostly performed off-track by V&V experts, rather than by field engineers as part of the development process. The translation task can be automated, but the abstraction phase is more complex: though the formal grounds for rigorous abstraction are a topic of active research [10], they have yet to be turned into useful tools.

### 3.5 The Relevance of Model Checking

Classifying model checking as an analytic verification technology carries the idea that it is used to prove that a design is correct. This is an over-optimistic, but also largely incomplete picture.

It is over-optimistic because any verification result is conditioned on the implicit hypothesis that the verified model indeed reflects the design. Theorem proving has the expressive power to carry arguably fully general proofs, though on small problems and with extensive expert guidance. Model checking, on the other hand, suffers from the abstractions and simplifications needed to get to a finite and tractable model. For example, a property that is successfully verified for a system of four components might fail for five or more components. When an error is reported by the model checker, the diagnostic information can be analyzed to trace the error back to the design or to the model. If nothing is found though, a doubt persists. For this reason, model checking is sometimes coined as a *falsification method*, that is, a way to prove systems wrong rather than prove them right.

In this sense, model checking is akin to testing: it does not give absolute proofs of correctness, but increases the confidence level by exploring a quantifiable part of the system's possible configurations. Model checking will give a much wider coverage than testing for a much lower cost. On the other hand, it will overlook a whole lot of implementation details that can only be tested on the final implementation. Model checking does not replace testing but complements it: testing can be focused on later stage issues such as interface compatibilities, while model checking will find concurrency bugs that are very hard to track down with testing.

Even more importantly, though, model checking is a very powerful and flexible software understanding and debugging tool. It gives the possibility to explore a program, look for a particular configuration, guide the search with a specific property, all this while automatically going back and forth through all possible alternative executions. It allows more errors to be found early in the design and thus fixed at little cost, resulting in improved software quality.

# 4 Verification of Model-Based Systems

Model-based autonomous systems present a particularly tough challenge for model checking techniques. Usual procedural programming languages such as Ada, C++ or Java can be fairly easily translated and abstracted into transition systems for modeling purposes [12]. In contrast, the reasoning engines used in MBR perform complex computations using large-size data structures capturing their knowledge about the model.

The issues of correctness of the general-purpose reasoning engine and the application-specific model are very different ones that we can address separately. The autonomy model can be considered as a high-level program that is "executed", in a somewhat unusual way, by the reasoning engine. The reasoning engine is a more complex, but also more stable and better understood part. It is typically built around a couple clearly identified algorithms that have been subject to careful scrutiny and documented in technical publications. For this type of essentially sequential algorithm, theorem proving tools seem more appropriate. This verification work, however, is to be done once and for all by the designers of the model-based infrastructure.

## 4.1 Verification of Autonomy Models

From the point of view of the autonomy application developer, the reasoning engine should be viewed as a stable, trustable part, just as a C programmer trusts his C compiler. His main concern is the validation of the autonomy model he is writing with respect to the real system that this model represents. This can be addressed by converting this model into a verification model that can be model-checked against expected properties of that system.

Autonomy models are themselves high-level descriptions, as opposed to low-level programming code found in more conventional software development. This is one of the main benefits of using model-based approaches. It is also beneficial to analytic verification: for systems of comparable complexity, an autonomy model is more likely to be tractable for model-checking after translation but without further abstraction, whereas a controller developed using conventional programming techniques would require important simplifications to be amenable to model-checking. This makes an approach based on pure translation, which is much easier to implement, viable for practical use. For the verification step to be completely transparent to the developer of autonomous models, three translations have to be provided, as illustrated in Figure 3:

- The *model* is translated from the autonomy syntax to the verification syntax. This also produces a correspondence map between elements of the two models (components, variables, events, etc.);
- In the *property* to be verified, elements of the autonomy model are replaced by corresponding elements of the verification model. For convenience, this translation can also convert the concrete syntax for properties, so that logic operators can be expressed in the autonomy syntax as well. Moreover, new property constructs can be defined and expanded into those supported by the model checker.
- Conversely, in the *diagnostic traces* returned by the model checker, elements of the verification model have to be replaced back by the corresponding elements of the autonomy model.

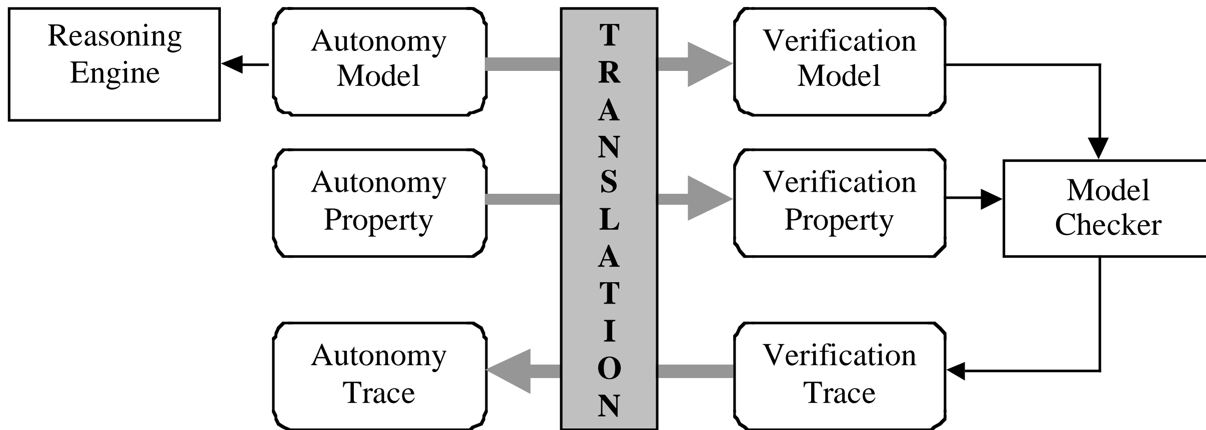


*Figure 3. Translation for Model Checking of Autonomy Models*

Unlike usual computer programs, autonomy models are not intended to describe sequences of operations but rather to provide a description of possible operations from which such sequences can be inferred when needed. The model itself allows for a very wide range of behaviors, with a relatively weak coupling between successive steps. In terms of model checking, this tends to produce a broad but shallow state space: the graph of reachable states has a big branching factor, corresponding to the numerous possible behaviors, but all states are reached within a relatively small number of steps.

Autonomy models such as those used in Remote Agent also use a declarative style, where elements are described as a combination of logic statements. This supports abstraction and modularity, but is exposed to completeness and consistency concerns: if a statement is too weak, improper executions can appear; if it is too strong, no execution can be possible. In terms of states and transitions, inconsistency manifests itself as a deadlock state, inconsistency as a form of non-determinism. Both can be detected by model checking, though not all model checkers can easily express determinism.

### 4.2 An Example: Symbolic Verification of Livingstone Models

After exploratory experiments on models used by the Remote Agent Planner/Scheduler (see Section 5.3), the principles for verification of autonomy models set forth in the previous section

have been applied to verify models for the Livingstone health management system using symbolic model checking.

Livingstone is used to monitor the health of a complex device such as a spacecraft. It tracks the commands issued to the device and monitors device sensors to detect and diagnose failures. To achieve this, Livingstone relies on a model of the device that describes, for each component, the nominal and abnormal functioning modes, how these modes are affected by commands and how they affect sensors. All sensor data is processed by a set of monitors that turns physical measures into a-priori defined discrete values such as *high*, *medium* and *low*. The Livingstone model thus represents a combination of concurrent discrete, finite-state transition systems.

The ASE group, in collaboration with Prof. Reid Simmons at Carnegie Mellon University (CMU) has developed a translator from Livingstone models to the SMV symbolic model checker [2]. The essence of the translation is fairly straightforward, due to the similar synchronous concurrency model used in both Livingstone and SMV (i.e. all components take a lock-step transition at each step). The main difficulty comes from discrepancies in variable naming conventions between the Lisp-like syntax of Livingstone and the Pascal-like syntax of SMV.

To express properties to be verified, SMV supports the powerful temporal logic CTL. In the Livingstone model, such properties are encapsulated in special declarations and written in a Lisp-like style that is consistent with the rest of the Livingstone model. Pre-defined specification patterns and variables can also be used for common properties such as consistency, reachability of given component modes or existence of a broken component. These declarations are captured and converted into SMV syntax by the translator. For example, a property

```
(all (globally (implies (on (heater h1))
                        (high (temp t1)))))
```

in the Livingstone model could be translated into the following SMV statement:

```
AG ((h1.mode = on) -> (t1.temp = high))
```

Work is in progress at CMU for converting SMV diagnostic traces back into Livingstone syntax, thereby completing the bridge between Livingstone and SMV.

Consistency and completeness are a prime source of trouble for designers of Livingstone models. For example, each mode of each component has a list of associated transition declarations

```
(name :when cond :next mode)
```

that work as guarded commands: "if `cond` holds, then transition to `mode`". For the model to work correctly, it is required that exactly one of the `cond` of each active mode hold at each step. If two transitions are enabled simultaneously, then two next modes are enforced at the same time, resulting in inconsistency. The translator supports predefined properties to check for such errors. When the pre-defined property `:consistency` is given, the translator extracts the guards of all transitions and generates, for each mode, a mutual exclusion property among its transitions.

The translator is being used at NASA Kennedy Space Center by the developers of a Livingstone model for the In-Situ Propellant Production (ISPP), a system that will produce spacecraft propellant using the atmosphere of Mars. The translator allows the model developers at Kennedy to express the properties to be checked in their familiar modeling syntax, then invoke the SMV model checker without writing a single line of SMV syntax. First experiments have shown that SMV can easily process the ISPP model and verify useful properties such as reachability of normal operating

conditions or recoverability from failures. The current version of the ISPP model, with $10^{50}$ states, can still be processed in less than a minute using SMV optimizations (re-ordering of variables). The Livingstone model of ISPP features huge state spaces but little depth (all states can be reached within at most three transitions), for which the symbolic processing of SMV is very appropriate.

### 4.3 Verification of Model-Based Applications

Model checking of an autonomy model only addresses its validity as an abstraction of a physical system, not its adequacy as support for model-based reasoning for which it is designed. Feeding a sensible model into a sound reasoning engine does not guarantee that the desired answer will always be obtained. The problems addressed by model-based systems are of high computational complexity, or even not decidable in general, so the reasoning engines are based on partial, heuristic algorithms that may fail to find a solution even if such a solution exists. For example, a planner might not be able to find a plan to achieve a given goal. It may also happen that, although the model is correct, the engine does not have enough information to give a correct answer. For example, a fault might not be diagnosed because of insufficient sensory information. It is therefore still desirable to perform analytical verification over the whole model-based system, that is, consider both the reasoning engine and the autonomy model it uses.

When we look at the autonomous controller as a whole, the autonomy model is data used by the reasoning engine, as opposed to a dynamic model of its own. In order to apply model checking to the complete application, we need a verification model of the reasoning engine and its data structures, including the autonomy model. Producing such a model would be an arduous and error-prone task. Furthermore, the size of the data structures involved would severely limit the number of states that can be covered.

Nevertheless, an intermediate approach, halfway between testing and model checking, can be considered. We will refer to this approach as *analytic testing*. Like conventional testing, the real reasoning engine is executed inside a testing environment, rather than simulating some abstract model of it. In particular, the environment has to contain a simulator for the controlled system. However, the engine and the environment code are instrumented in order to allow finer control on how the test is executed. Instead of running a handcrafted sequence of test runs, the test driver uses the same kind of systematic exploration algorithm as used in model checkers to drive the system through a whole range of scenarios, while looking for violations of desired properties.

How far this approach can go will depend on the provided instrumentation. At least, the test driver should be able to control scheduling of the different components, both in the engine and in the environment, stop execution at choice points and select which branch is followed (e.g. which fault is simulated). The VeriSoft tool (Lucent) [9] applies this principle to C programs. If the state of the application can also be accessed and modified by the test driver, then it is possible to perform an exhaustive exploration. Otherwise, as in Verisoft, loops cannot be detected and the exploration has to be pruned at an arbitrary depth.

This analytic testing would provide a better accuracy of the verification results, since no translation or abstraction of the verified system takes place. While verification of models can search for potential causes of incorrect reasoning, analytic testing of complete applications will allow to actually check that the reasoning engine tells the right thing. On the other hand, analytic testing will run real code and thus be much more hungry in computing resources, so the search space will have to be narrowed down to a tractable range, typically by focusing on a few typical mission scenarios.

As far as we know, little work has been done until now to address verification of model-based autonomy applications as a whole. In the coming months, we will develop analytic testing technology for the Livingstone system at NASA Ames, as a continuation of the work done on Livingstone models described in the previous section.

# 5 Other Experiences at NASA

In addition to verification of Livingstone models described in Section 4.2, this section presents other examples of verification done on autonomy software at NASA. Most of them have been performed by the Automated Software Engineering group (ASE) at NASA Ames Research Center. They focus on components of Remote Agent, described in section 2.3.

## 5.1 Verification of Remote Agent Executive

A team from the ASE group used the Spin model checker to verify the core services of RA EXEC and found five concurrency bugs [17]. Four of these bugs were deemed important by the executive software development team, which considers that these errors would not have been found through traditional testing. Once a tractable Spin model was obtained, it took less than a week to carry out the verification activities. However, it took about 1.5 work-months to manually construct a model that could be run by Spin in a reasonable length of time, starting from the Lisp code of the executive. The initial models were not sufficiently abstract and simple to be tractable by Spin.

This is a typical case of conventional V&V effort: an existing system is handed to V&V experts who have a hard time understanding the original design and distilling it down to a tractable model, and then find and report concurrency bugs. It stresses the cost of modeling, as opposed to verification. The designers of the Executive would arguably have much less trouble doing the modeling work, since they knew their program much better, but would have a hard time learning Spin and tuning the model to a tractable size.

## 5.2 Search for the RAX Anomaly

Shortly after an anomaly was discovered in RA EXEC during the Remote Agent Experiment (RAX) in May 1999, the ASE team took the challenge of performing a "clean room" experiment to determine whether the bug could have been found using verification and within a short turnaround time. Over the following weekend, a front-end group selected suspect sections of the code, and a back-end group performed the modeling in Java and the verification in Spin, using the group's Java-to-Spin translator Java Pathfinder [12]. The hardest part was to understand the Lisp source code of EXEC. It then took 3 hours to produce a two-page Java program that models the backbone of the concurrent structure of two tasks and reproduces the bug. As it turns out, this bug is a deadlock due to improper use of synchronization events, and is identical to one of the five bugs detected in another part of EXEC with Spin two years before.

The main lesson is not so much in the success of this verification effort than in the correlation with the previous one: it proves that the kind of concurrency bugs that were found and fixed in another part of the system can indeed pass through heavy test screens and compromise a mission. Besides this, it stresses again the difficulty of the modeling phase. It also illustrates the convenience of Java's concurrent programming primitives for modeling purposes.

### 5.3 Verification of Remote Agent Planner/Scheduler Models

Researchers from ASE conducted preliminary experiments in translating Planner/Scheduler models to the SMV, Spin and Murphi model checkers [23]. Those models are composed of a large number of tightly coupled declarative constraints, whose combined effects are difficult to apprehend. Automated validation can find inconsistencies and determine whether implicit properties of the model can be derived from the set of explicit constraints. The experiments were done on a small model of an autonomous robot. A useful subset of the planner modeling language, covering the robot example, could be translated in all three model checkers. No translation tool was built, but the translations were done by hand in a systematic way amenable to automation. The analysis identified several flaws in the model that were not identified during testing. Out of the three model checkers used, SMV allowed the most direct translation and was by far the fastest: 0.05 seconds vs. ≈30 seconds for Spin or Murphi.

This was the first experiment by ASE in model checking of autonomy models. The good performance of SMV can be attributed to different factors:

- The declarative style of Planner models is directly translatable in SMV models, whereas it has to be simulated by a big iteration loop in Murphi and Spin.
- Planner models are weakly constrained: a lot of states can be reached at any time. Spin and Murphi have to enumerate all those states and transitions explicitly, whereas the symbolic computations in SMV involve fairly compact logical expressions.

Planner models also contain timing information that was ignored here since un-timed model checkers were used. The Autonomy group at NASA Ames is currently pursuing further experiments using the Uppaal real-time model checker [16].

### 5.4 Consistency Checking of Remote Agent Traces

While testing the Remote Agent, long logs of exchanged messages were generated. These logs have to be searched for errors by experts, a cumbersome and error-prone task. The Formal Methods group at Jet Propulsion Laboratories has used a database to verify those traces [7]. The traces were checked for consistency with explicit design requirements (so-called *flight rules*), and correct message flow. The messages, and their ordering, were entered as objects in the database, and the flight rules and message flow requirements were then formulated as queries on the database. It took around two minutes to check each query on a set of about 60 traces. All traces were found to meet all the requirements.

This is a different and quite original approach. Analytic verification is performed on test results, not on the system itself, so the results have a more limited significance. On the other hand, it relates to system-level testing of the whole Remote Agent, whereas model checkers can only deal with smaller sub-components. This approach was also much easier and faster to set up than a typical model checking task.

## 6 Conclusions

Because of the internal complexity of autonomous controllers, and the huge range of situations that they can potentially address, scenario-based testing provides a very limited coverage. Model

checking can help to find the concurrency problems that would be overlooked in testing, and fix them earlier in the development and thus at cheaper cost.

For autonomy software based on conventional programming techniques, the analytic verification issues do not differ much from those met in other software systems such as communication protocols or safety-critical controllers. The main obstacle is the translation and abstraction work required to go from a complex piece of software to an abstract model of tractable size.

For model-based autonomy applications, recent experiments have shown successful use of symbolic model checking for the verification of autonomy models. The high level of abstraction of those models allows model checking to be applied on direct translations, without further abstraction. Verification of complete model-based applications is beyond reach of model checking, but an intermediate solution applying model checking principles to support analytic testing has been outlined.

A key factor in the future success of model checking, and other analytic verification technologies, is their close integration in the development environment of autonomous system designers. The principles alone require some learning phase; users are not willing to spend more time to learn the input language of a model checker, and re-write their program in that language. If the model checker becomes just another button next to the source-level debugger, then developers will definitely use it and reap its benefits.

Autonomous software requires even more reliability than current critical control software, because there will be little or no human supervision to detect and act upon unexpected failures at run-time. Analytic V&V can become the key enabling factor for using autonomous systems, providing the necessary level of assurance that the right thing will be happening when no one is watching.

**Bibliography**


[1] R. E. Bryant. "Graph-based algorithms for boolean function manipulation". IEEE Transactions on Computers, C-35 (8), 1986.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, "Symbolic model checking: 10^20 states and beyond", Information and Computation, vol. 98, no. 2, June 1992, pp. 142–70.

[3] Ricky W. Butler , James L. Caldwell, Victor A. Carreno, C. Michael Holloway , Paul S. Miner and Ben L. Di Vito. "NASA Langley's Research and Technology-Transfer Program in Formal Methods". In: 10th Annual IEEE Conference on COMPuter ASSurance (COMPASS '95), Gaithersburg, MD, June 1995

[4] O. Coudert, C. Berthet, J. C. Madre. "Verification of synchronous sequential machines based on symbolic execution". In: Proc. of the Workshop on Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science, vol. 407, Springer-Verlag, 1989.

[5] D. L. Dill, A. J. Drexler, A. J. Hu and C. H. Yang. "Protocol Verification as a Hardware Design Aid". 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525.

[6] B. L. Di Vito. "Formalizing New Navigation Requirements for NASA's Space Shuttle". Lecture Notes in Computer Science 1051, Springer Verlag, 1996.

[7] M. Feather, "Rapid Application of Lightweight Formal Methods for Consistency Analyses", IEEE Transactions on Software Engineering, vol. 24, no. 11, November 1998.

[8] P. Godefroid. " "Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem", volume 1032 of Lecture Notes in Computer Science, Springer-Verlag, January 1996.

[9] P. Godefroid. "Model Checking for Programming Languages using VeriSoft". Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174-186, Paris, January 1997.

[10] S.Graf, H.Saïdi. "Construction of abstract state graphs with PVS". In Proceedings of the 9th Conference on Computer-Aided Verification (CAV'97), Haifa, Israel, June 1997.

[11] A. R. Gross, K. R. Sridhar, W. E. Larson, D. J. Clancy, C. Pecheur, and G. A. Briggs, "Information Technology and Control Needs For In-Situ Resource Utilization", to appear in Proceedings of the 50th IAF Congress, Amsterdam, Holland, October 1999.

[12] K. Havelund, T. Pressburger. "Model Checking Java Programs Using Java PathFinder" To appear in "International Journal on Software Tools for Technology Transfer".

[13] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. "HyTech: A Model Checker for Hybrid Systems". Software Tools for Technology Transfer 1:110-122, 1997.

[14] G. J. Holzmann, "The Model Checker SPIN", IEEE Transactions on Software Engineering, vol, 23, no. 5, May 1997.

[15] C. N. Ip and D. L. Dill. "Better Verification through Symmetry". In: Formal Methods in System Design, Volume 9, Numbers 1/2, pp 41-75, August 1996.

[16] Kim G. Larsen, Paul Pettersson and Wang Yi. "UPPAAL in a Nutshell". In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[17] M. Lowry, K. Havelund, J. Penix, "Verification of AI Systems that Control Deep-Space Spacecraft", in: Foundations of Intelligent Systems, LNAI, Vol. 1325, Springer Verlag, 1997.

[18] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. "Remote Agent: To Boldly Go Where No AI System Has Gone Before". Artificial Intelligence 103(1-2):5-48,August 1998.

[19] N. Muscettola. "HSTS: Integrating planning and scheduling". In Mark Fox and Monte Zweben, editors, Intelligent Scheduling. Morgan Kaufmann, 1994.

[20] P. P. Nayak et al. "Validating the DS1 Remote Agent Experiment". In: Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS-99), ESTEC, Noordwijk, 1999.

[21] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. "PVS: Combining specification, proof checking, and model checking". In: Rajeev Alur and Thomas A. Henzinger, editors, Computer-Aided Verification, CAV '96, volume 1102 of Lecture Notes in Computer Science, Springer-Verlag, pages 411-414.

[22] B. Pell, D. E. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. P. Nayak, M. Wagner, and B. C. Williams. "An Autonomous Spacecraft Agent Prototype". Autonomous Robots 5(1), March, 1998.

[23] J. Penix, C. Pecheur, K. Havelund, "Using Model Checking to Validate AI Planner Domain Models", Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard, December 1998.

[24] B. C. Williams and P. P. Nayak, "A Model-based Approach to Reactive Self-Configuring Systems", Proceedings of AAAI-96, 1996.

[25] S.Yovine. "Kronos: A verification tool for real-time systems". In Springer International Journal of Software Tools for Technology Transfer, Vol. 1, No. 1/2, October 1997.