Analytic Verification of Flight Software

Michael Lowry, NASA Ames Research Center
Daniel Dvorak, Jet Propulsion Laboratory, California Institute of Technology

IN THE REALM OF SPACE EXPLO-ration, the biggest obstacle to widespread application of autonomy in flight software is not technical feasibility; it is doubt about its trustworthiness as a replacement for human-in-the-loop decision-making. Autonomous control systems raise difficult *verification and validation* issues. V&V techniques are needed that significantly increase confidence in these decision-making systems.

The key to acceptance of this technology is not hit-or-miss testing but thorough V&V methods that yield guarantees. We've developed such a method that applies two analytic-verification approaches: design-time model checking that guarantees that specific conditions are never violated, and runtime embedded behavior auditors to verify that the implemented integrated system respects similar conditions. Together, they make verification activities part of design and development, not just a back-end step.

The challenges of V&V

Traditional space missions without autonomy are already inherently risky. Charles Perrow identifies two risk dimensions for high-risk technologies: interactions and coupling.[1] Complex interactions are those of unfamiliar or unplanned or unexpected sequences, and are either invisible or not immediately comprehensible. Tightly coupled systems have more time-dependent processes that cannot be delayed or extended. His chart (see Figure 1) identifies space missions as having both characteristics, thus placing them in the quad-

To provide rigorous validation and verification of autonomous software, the authors apply two analytic-verification approaches: design-time model checking and runtime behavior auditing.

rant depicting the riskiest technologies.

Flight-project managers are therefore understandably reluctant to risk a science mission on unproven technologies. Flight-qualification programs for new technology such as NASA's New Millennium and X2000 are essential to overcome this initial hurdle. However, flight-project managers also need to be convinced that any technology can be verified and validated in the specific context of their mission. This poses a special challenge to autonomy software, because traditional V&V approaches are inadequate for it.

Traditional spacecraft control uses sequences: deterministic, time-stamped, linear series of commands. Their roots go back to electromechanical controls similar to those for washing machines, although today sequences are implemented as software instructions. Sequences are validated mainly through manual review by several engineering teams (for example, thermal and power) to ensure that the sequences achieve their goals without posing hazards to the spacecraft. This type of validation is possible precisely because the engineering teams need to consider only one execution path. Even minimal generalizations of straight-line sequencing are viewed as risky. One such example is conditional sequencing,

where a sequence has top-level conditionals—for example, if the spacecraft is out of the planet's shadow, turn the solar panels toward the sun.

In contrast, autonomy software for spaceflight compactly encodes at least millions, if not billions, of execution paths. Traditional approaches to the V&V of sequences cannot scale to this level.

Also, autonomy software is inherently concurrent—that is, multiple processes achieve different goals, or subgoals execute in parallel. Concurrent-task software is easier to program than traditional sequences because the means of achieving each goal can be designed separately. Because of the closed-loop nature of autonomy, each goal being achieved represents a separate thread. However, unintended interactions between threads can lead to failures. These failures are very difficult to find and debug through testing. More thorough means of finding concurrency errors are required.

Analytic verification approaches can meet the V&V challenges of autonomous software. They can scale to handle the complexity of autonomous control, and can calculate whether the concurrent-software designs are correct and monitor their execution.
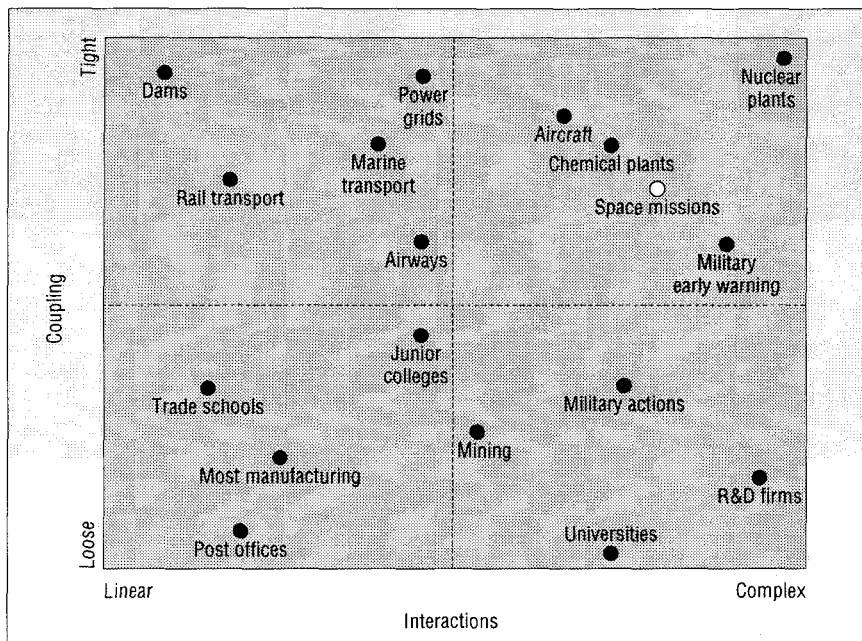
Figure 1. The two dimensions of risk for high-risk technologies.[1] With complex interactions and tight coupling, space missions fall into the quadrant depicting the riskiest missions.

Design-time analytic-verification approaches

Unlike traditional testing, which samples a digital system's behavior, analytic verification (also called *formal verification*) mathematically calculates the system's behavior. Traditional testing is limited because the number of tests required to achieve statistical confidence in the system's reliability grows dramatically—both as a function of the system's complexity and as a function of the desired degree of confidence. The Pentium floating-point bug illustrates this. Under rare circumstances, the floating-point circuitry of the early Pentiums produced an incorrect result. Extensive testing before the Pentium's release did not reveal these circumstances.

However, these circumstances could have been revealed through analytic-verification algorithms. For this reason, after the Pentium floating-point bug, digital-hardware developers have invested heavily in formal-verification techniques to complement simulation and testing. Results have been encouraging and are regularly reported in arenas such as the International Conference on Computer-Aided Verification.[2] Excellent results have also occurred in the verification of computer firmware (for example, complicated hierarchical memory protocols), communication protocols, and operating-system services.

To date, two main approaches to design-time formal verification exist: *computer-based theorem proving* and *model checking*.

Computer-based theorem proving. This approach can verify unbounded (that is, infinite state) systems. Theorem-provers that are often used in software verification include PVS[3] and ACL2.[4] Theorem-proving has two disadvantages. First, it requires an expert's sustained effort over a substantial time period. Second, it provides little direct information if the system is not correct. The inability to find a proof of correctness is usually interpreted as indicating that the expert needs to explore a different approach, not that the proof doesn't exist.

Typically, an expert will interact with a computer-based theorem prover for several work-months to generate a mechanically verified proof of the correctness of some key design or subsystem of a digital system. The effort usually focuses on developing an induction hypothesis that guarantees that if a system starts in a state obeying the correctness criteria, each transition of the system to the next state will guarantee those criteria. Carrying out a successful proof by induction requires the expert to be skilled at finding *invariants* that are always true of the system and that can be added as lemmas to support the induction hypothesis.

Although computer-based theorem provers have become increasingly powerful, performing many of the smaller proof steps automatically, the problem of finding suitable invariants has prevented this approach's automation. Research toward automated invariant generation is encouraging, but completely general algorithms will remain elusive.

Model checking. This approach is a mathematical technique for verifying and debugging concurrent or real-time systems modeled as interacting finite-state machines. Given a model and a *property*, a model checker searches for *traces* of the model that violate the property. Properties can be invariants, temporal properties (that is, defined through modal operators such as `eventually`), or in the case of real-time model checkers, metric time constraints defined through linear relations. A trace is an interleaved sequence of states (or, dually, transitions) of the finite-state machines. If the checker finds no traces violating the property, and the algorithm runs to completion, the property is verified.

Model checkers differ from simulators in that they explore all relevant traces. In other words, they explore all realizable paths through the graph of states that can be reached from the initial state and that match the property being checked. They also enable checking much richer concurrency properties than is typical of simulators. Some model checkers are similar to theorem provers in that they manipulate symbolic descriptions of the transition relation. However, model checkers do not perform induction and hence cannot verify systems with unbounded state. On the other hand, they are completely automatic and thus more practical for verification in a spiral development process than theorem provers.

Design verification using model checking

Model checkers are particularly well-suited to exploring the relevant execution paths of nondeterministic systems with multiple processes running in parallel. This makes them well-adapted to verification and debugging of autonomy software. The number of possible interleavings of the executions of parallel processes has an upper bound proportional to the product of the number of local states traversed in the execution of each process. Anticipating all these interleavings can be difficult for a human designer; model checking can find subtle, pernicious interactions that violate correctness conditions.

For example, an unexpected interaction between a communication process, a weather-data process, and an information-bus process caused the Mars Pathfinder software to enter a quasi-deadlock. This resulted in Pathfinder resetting itself because of the timeout of a

```
funcall-w-m-properties (props closure)          maintain-properties-daemon
    unwind-protect(                                 loop-forever do
        do                                              if check-locks
            critical section(...);                          do automatic recovery;
            achieve-lock-properties(locks);             if not(changed? (...count...))
            funcall(closure);                               then wait-for(...);
        od,                                         od
→
        release-locks(locks));
(a)                                             (b)
```

Figure 2. Simplified extracts of the executive code, converted to an Algol-like syntax: (a) the `funcall-with-maintained-property` construct; (b) the `maintain-properties-daemon`.

watchdog timer, and in the loss of several days' worth of science. A subsequent model-checking replication of the interaction of these three processes duplicated this bug.

In this section, however, we discuss the discovery of bugs during design verification of autonomy software, before they occur during a mission. As an example, we'll briefly describe one such bug that our model checker found in the executive subsystem of the DS-1 Remote Agent.[5]

The correctness criteria given to a model checker are expressed in a temporal logic that includes logical predicates on states and temporal operators on traces. Temporal logics differ in the specific temporal operators they provide. The temporal logic LTL includes the temporal operators []P, meaning that the predicate P is always true of the states in a trace, and <>P, meaning that P is eventually true of a state in a trace. (This logic is neither strictly less nor more powerful than the Tspec language described below for execution-time checking, but a significant area of overlap exists between what properties the two languages can express.)

In the executive code for the DS-1 Remote Agent, the model checker Spin[6] found an error trace that violates an *eventually* property. Specifically, it found an error trace where an executive-task program might abort without eventually releasing its property locks. This can cause a deadlock where other tasks cannot execute because the aborted task has not released the properties they need. The properties locked by a task are typically device states required for the task's successful execution—for example, the requirement that the engine-gimbal actuator be activated when the rocket engine is firing. The error occurred in the executive's central core, which provides services analogous to those of an operating system.

Figure 2 contains simplified extracts of the executive code, converted to an Algol-like syntax. The `funcall-with-maintained-property` construct (see Figure 2a) defined in the executive is called with two arguments:

`props`, the properties to be maintained during execution of the task's body, and `closure`, the body itself. `funcall-with-maintained-property` in turn uses the `unwind-protect` construct to provide a wrapper around the body's execution. This ensures that if an abort occurs during the execution (for example, an error is signaled because a locked property is violated, and the error is unrecoverable), the locks are released before the task exits.

The `maintain-properties-daemon` (see Figure 2b) monitors property violations. It invokes automatic recovery if a property violation occurs; if the recovery does not succeed, an abort occurs. If a task's body executes without aborting, the `release-locks` statement also executes afterwards. However, this statement is not itself protected. Thus, if an abort occurs during the execution of `release-locks`, `funcall-with-maintained-property` exits—whether or not the locks have been released. The bold arrow in Figure 2a and the bold code in Figure 2b indicate the interleaving of these two processes that leads to the error trace.

Over the course of several days, a NASA Ames design-verification team (Klaus Havelund, John Penix, and Michael Lowry) using Spin found five concurrency bugs, including the one above. Four of these bugs were deemed important by the executive team, which believes that traditional testing would not have found these errors. In addition, the checker verified that several properties were correct; that is, it found no error traces.[7]

Runtime verification

In contrast to design-time formal verification, runtime formal verification checks the implemented system's behavior during execution, rather than a model's properties at design time. This verification is formal in that it checks system behavior against a specification (a model) of valid behavior using runtime auditors (also called *oracles*). This

approach is an element of the larger field of specification-based testing.[8]

Runtime verification has different benefits and limitations than design-time model checking. On the positive side, it checks the implemented system rather than a design model. So, it can detect implementation errors and check behavior at a much more detailed level. On the negative side, because runtime verification checks behavior during system execution, the checked behavior is limited to the relatively few traces that get exhibited during scenario-based system testing and actual operation. Thus, we view runtime verification as a partner of design-time model checking, not an alternative to it.

In current practice, a variety of people, including mission designers and system engineers, levy requirements on software. Such requirements are usually expressed in natural language and are therefore not directly usable for testing. To address that problem, we've developed Tspec, a behavior-specification language for nonprogrammers. Tspec uses a notation that spacecraft system engineers and software designers and developers find more intuitive than linear temporal logic. They can express behavioral constraints and expectations, using simple Tspec constructs. Such specifications are then compiled into a conventional programming language and linked with a small class library to form *behavior auditors*. When linked with the operational software, these embedded behavior auditors perform runtime verification.

Tspec constructs. Tspec currently offers five types of behavior specifications—*inline tests, invariants, state machines, episodes,* and *resource constraints*—that a user can instantiate to specify the boundaries of acceptable behavior. In all cases, if the observed behavior violates the specified behavior, the violation is reported in a system-specific manner, typically through logging and alerting. With the exception of inline tests, which get evaluated in the direct flow of execution, the behavior auditor associated with each speci-

```
      invariant NoSunInCamera (lensCoverOpen, sunConeAngle){
          never (lensCoverOpen && sunConeAngle < 0.1)
      }
(a)

      statemachine TrafficLight (color){

          states {red, yellow, green}

          transitions{
              red      -> green
              green    -> yellow
              yellow -> red}

          limits {
              duration(red,    15, 60)
              duration(green, 11, 58)
              duration(yellow, 2,   4)
              rate(4, 10, 300)}
      }
(b)

      episodeMeasureField(calibrate, reading, powerOn){
          steps {
              update(calibrate, true)
              update(reading, BEGIN)
              update(reading, END)
          }
          require {powerOn == true
          limits {
              duration(90, 150)
              end_begin_delay(60, infinity)
          }
      }
(c)

      //Electric power: renewable resource, 150 watts.
      resource renewable power (150);

      //show usage and freeing of power
      when (xsspa_pwr, ON)    consume (power, 45);
      when (camera_pwr, ON)   consume (power, 6);
(d)
```

Figure 3. Examples of Tspec constructs: (a) an invariant; (b) a state machine; (c) an episode; (d) a resource.

fication gets evaluated indirectly as an observer of changes to specific variables (described later in "Instrumenting the code").

An inline test—like a C assert macro—specifies a boolean expression that should always evaluate to true when the control flow passes through during execution.

An invariant specifies a logical condition that should always evaluate to true. Unlike an inline test, which is evaluated only when control flow passes through it, an invariant's condition is evaluated every time any of its variables changes value, regardless of which line of code caused the value change. The invariant in Figure 3a specifies a camera safety requirement: never allow the lens cover to be open if the cone angle between the camera boresight and the sun vector is less than 0.1 radian. This example is simple enough that you might consider extending the construct to enforce the invariant rather than merely check it. However, stating an invariant is generally much easier than enforcing it.

A state machine specifies constraints on state-transition activity and is evaluated every time a specified state variable changes value. For example, the state machine in Figure 3b specifies requirements on a traffic-light controller in terms of legal transitions (for example, red to green), state durations (for example, the red-state duration must be between 15 and 60 seconds), and the expected transition rate (four to 10 transitions in any 300-second interval).

An episode specifies a behavior fragment having a beginning event and an ending event, with potentially many intermediate events, where the events are expected to occur in the specified order. A change in any of an episode's event variables triggers evaluation. The episode in Figure 3c specifies requirements on a science measurement activity in terms of expected steps (three update events are expected), required conditions (powerOn must remain true during the episode), and timing constraints (the

episode should take at least 90 seconds but not more than 150 seconds, and at least 60 seconds must separate the end of one episode from the beginning of the next). Episodes that remain unfinished at the end of a test scenario are reported as warnings.

A resource specifies the type and amount of an available resource and the condition under which it is consumed. A violation occurs if a resource limit is exceeded. Resources are either *depletable* (for example, propellant) or *renewable* (for example, power from a solar panel). The resource in Figure 3d specifies the total amount of electrical power (150 watts) and the two conditions under which that power is consumed (for example, the camera draws six watts when it is powered on). With additional when-consume statements that detail the amount of power consumption implied by specific states, the associated auditor will report if the system's aggregate state ever implies greater than 150 watts of power consumption.

Behavior auditors. Auditors report not only the occurrence of unexpected events and conditions, but also the absence of expected events and conditions. Discrete events, such as the updating of a state variable or expiration of a timer, trigger the auditing. The kinds of behavior violations that these auditors detect include conditions such as value out of range, illegal state transition, out-of-order event, resource threshold exceeded, state persisted too long, and activity started but never completed. The focus on discrete-event behavior checks aims at detecting failures in decision-based autonomous control systems. Continuous control systems such as attitude control normally include specialized monitors as part of the spacecraft's fault-protection design that abstract behavior into a few discrete states.

The auditors are not part of some temporary test scaffold; rather, they are embedded in mission software. This gives them access to potentially all software state variables; therefore they can check virtually any flight rule, not just the subset that might be checkable from a log of selected variables and events. In addition, continuous checking during a mission can provide early warning to ground operations when something unexpected is happening—whether because of hardware or software failure. This changes the concept of system testing from "checking the log files" to embedded real-time behavior monitoring, from development through deployment. This

approach truly implements a maxim of flight-software engineering: "Test what you fly and fly what you test."

Instrumenting the code. Obviously, the operational software must be instrumented so that the behavior-auditing code can access the required variables. In an object-oriented design, this can be accomplished unobtrusively through the Observer design pattern.[9] This mechanism provides a loose coupling between the operational code and auditor code, with no change in the operational code as the auditor code is inserted or removed.

Currently, Tspec specifications are being compiled into C++ for embedding in software for JPL's X-33 Avionics Flight Experiment. In this experiment the auditors will be checking behavior visible in a telemetry stream. Later, in JPL's X2000 program, Tspec specifications will be embedded in autonomous control loops.

UR APPROACH TO, AND EXPERI-ences with, design-time model checking and runtime verification suggest several important changes in software-development practices.

Although the design verification of the executive code of DS-1 Remote Agent took less than a week, constructing an abstracted design in the Promela language (used by the Spin model checker) from the Lisp code took about one-and-a-half work-months. Achieving a model that was sufficiently abstracted to be computationally tractable for verification by Spin required significant effort for two main reasons. First, model-checking languages today are impoverished compared to programming or specification languages. Consequently, Lisp is much more expressive than Promela, and straightforward syntactic translations of Lisp into Promela result in code blowup. So, the translation was hand-tailored. Second, much of the code was not relevant to the correctness conditions on which this exercise focused. Understanding the code sufficiently to know how to prune away the irrelevant portions of the design and limit the degrees of freedom of the remaining code without eliminating possible error traces was difficult. For manual modeling, understanding is a prerequisite to performing good abstractions.

To enable automated design verification of autonomy software, researchers are pursuing automated modeling. We believe that with suf-ficiently good facilities for abstraction, programming languages and design-specification languages can be used directly in model checking. This will let developers use model checking directly as part of a debugging package for autonomy-software design. Automated abstraction is a challenging research goal being pursued by a number of collaborating research institutions, including NASA Ames. A near-term alternative is to provide an abstraction workbench, in which autonomy designers can annotate their code with directives for various kinds of abstractions, which would be applied syntactically.

Furthermore, we believe that the concept of "software delivery" should include not only the operational code, but also the associated verifiable behavior specifications. When a developer receives the initial requirements for a software subsystem, he or she should begin by expressing those requirements as verifiable behavior specifications. This has the very positive effect of focusing attention first on what the behavior should be rather than on how to implement it.

Finally, test engineers should inspect the behavior specifications. Behavior specifications are significantly shorter and easier to understand than operational code, so this type of inspection is more approachable than a formal code inspection. Such inspections help ensure that developers have adequately specified the expected behavior and thereby reduce the chance of undetected errors. Developers should be praised when their behavior specifications catch a bug, because early automated detection greatly eases debugging. ◻

Acknowledgments

References

1. C. Perrow, *Normal Accidents: Living with High Risk Technologies,* Basic Books, New York, 1984.

2. O. Grumberg, ed., *Proc. CAV '97: Ninth Int'l Conf. Computer-Aided Verification, Lecture Notes in Computer Science 1254,* Springer-Verlag, Berlin, 1997.

3. J. Crow et al., "A Tutorial Introduction to PVS," presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, 1995; http://www.csl.sri.com/sri-csl-fm.html.

4. M. Kaufmann and J. Moore, "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp," *IEEE Trans. Software Eng.,* Vol. 23, No. 4, Apr. 1997, pp. 203–213.

5. M. Lowry, K. Havelund, and J. Penix, "Verification and Validation of AI Systems That Control Deep-Space Spacecraft," in *Foundations of Intelligent Systems, Proc. Ismis '97: 10th Int'l Symp. Methodologies for Intelligent Systems, Lecture Notes in Artificial Intelligence,* No. 1325, Springer-Verlag, 1997.

6. G.J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.,* Vol 23, No. 5, May 1997.

7. K. Havelund, M. Lowry, and J. Penix, *Formal Analysis of a Space Craft Controller Using Spin,* NASA Ames Tech. Report 1770, NASA Ames Research Center, Moffett Field, Calif., 1998.

8. D. Richardson, S.L. Aha, and T. O'Malley, "Specification-Based Test Oracles for Reactive Systems," *Proc. 14th Int'l Conf. Software Eng.,* IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 105–118.

9. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, Mass., 1994.

Michael Lowry is a senior research scientist in the Computational Sciences Division of NASA Ames Research Center, and is the leader of the Automated Software Engineering group. His research interests have focused on high-assurance program synthesis and verification and validation techniques for complex systems. He serves on the editorial board of Kluwer's *Journal of Automated Software Engineering,* and was the 1997 program cochair of the IEEE Conference on Automated Software Engineering. Previously, he was a research scientist at the Kestrel Institute, where he developed mathematical approaches to program synthesis. He received his BS and MS in electrical engineering and computer science from MIT, and his PhD in computer science from Stanford University. Contact him at NASA Ames Research Center, M/S 269-2, Moffett Field, CA 94035; lowry@ptolemy.arc.nasa.gov.

Daniel Dvorak is a principle member of technical staff in the Information and Computing Technologies section of the Jet Propulsion Laboratory, where his interests have focused on monitoring, fault detection, and system-level testing of autonomous systems. Previously, he worked at Bell Laboratories on the monitoring of telephone switching systems and on the development of R++, a rule-based extension to C++. He received his BS in electrical engineering at the Rose-Hulman Institute of Technology, his MS in computer engineering at Stanford University, and his PhD in computer science at the University of Texas at Austin. Contact him at the Jet Propulsion Laboratory, 4800 Oak Grove Dr., M/S 301-270, Pasadena, CA 91109-8099; daniel.dvorak@jpl.nasa.gov.