Verification of Autonomous Systems Using Embedded Behavior Auditors

Daniel Dvorak and Eric Tailor

Jet Propulsion Laboratory, California Institute of Technology, Mail Stop 301-270, 4800 Oak Grove Drive, Pasadena, CA 91109-8099
Daniel.Dvorak@jpl.nasa.gov and Eric.Tailor@jpl.nasa.gov

Abstract. The prospect of highly autonomous spacecraft and rovers is exciting for what they *can* do with onboard decision making, but also troubling for what they *might* do [improperly] without human-in-the-loop oversight. The single biggest obstacle to acceptance of highly autonomous software control systems is doubt about their trustworthiness as a replacement for human analysis and decision-making. Such doubts *can* be addressed with a comprehensive system verification effort, but techniques suitable for conventional sequencer-based systems are inadequate for reactive systems. This paper highlights some of the key features that distinguish autonomous systems from their predecessors and then focuses on one approach to aid in their verification using a "lightweight" formal method. Specifically, we present a little language that enables system engineers and designers to specify expected behavior in the form of invariants, state machines, episodes, and resource constraints, and a way of compiling such specifications and linking them into the operational code as embedded behavior auditors. Such auditors become part of the overall fault-detection design, checking system behavior in real-time, not only in the test-bed but also in flight.

formal methods
temporal logic

INTRODUCTION

Autonomous spacecraft operation is becoming a necessity in space missions. The long round-trip light-time delays of deep space missions preclude earth-in-the-loop control for many mission objectives. Even a 10–40 minute round-trip delay in controlling a robot on the surface of Mars—a relatively nearby planet—puts a severe limit on the amount of science data that can be obtained with earth-based control. At the orbit of Saturn the round-trip delay is on the order of 2.5 hours. Also, without increased autonomy, NASA's objective of running many concurrent missions would strain not only the capacity of the deep space communication network but also the budgets needed to support traditional ground control. Finally, autonomy is an enabler for new mission concepts such as formation flying for interferometers and self-sufficiency for 100-year interstellar explorers.

Traditional space missions without autonomy are already inherently risky. In an examination of notable accidents involving complex systems, Perrow identifies two risk dimensions for high-risk technologies: interactions and coupling (Perrow, 1984). Linear interactions are those in expected and familiar production, and are quite visible even if unplanned, while *complex interactions* are those of unfamiliar or unplanned or unexpected sequences, and are either not visible or not immediately comprehensible. Loosely coupled systems can incorporate shocks and failures without destabilization, while *tightly coupled* systems have more time-dependent processes that cannot be delayed or extended. Perrow's chart of interactions versus coupling, shown in Figure 1, places space missions in the quadrant of the diagram depicting the riskiest technologies—those having complex interactions and tight coupling.

For space missions, "autonomy" implies giving a spacecraft or rover authority to make decisions without human-in-the-loop supervision. Compared to conventional spacecraft controlled by open-loop timed sequencers, autonomous spacecraft are controlled through many closed-loop goal-achieving modules. Autonomous systems are not only *reactive* systems—largely event-driven, continuously reacting to external stimuli and internal events (Harel, 1987)—but also *goal-driven* systems exhibiting purposeful behavior. Although these systems are still deterministic, like their sequencer-based predecessors, they are less predictable from a ground controller's perspective because they make decisions based on dynamically changing spacecraft state. This loss of predictability and automation of decision-making heighten the need for thorough verification and validation.

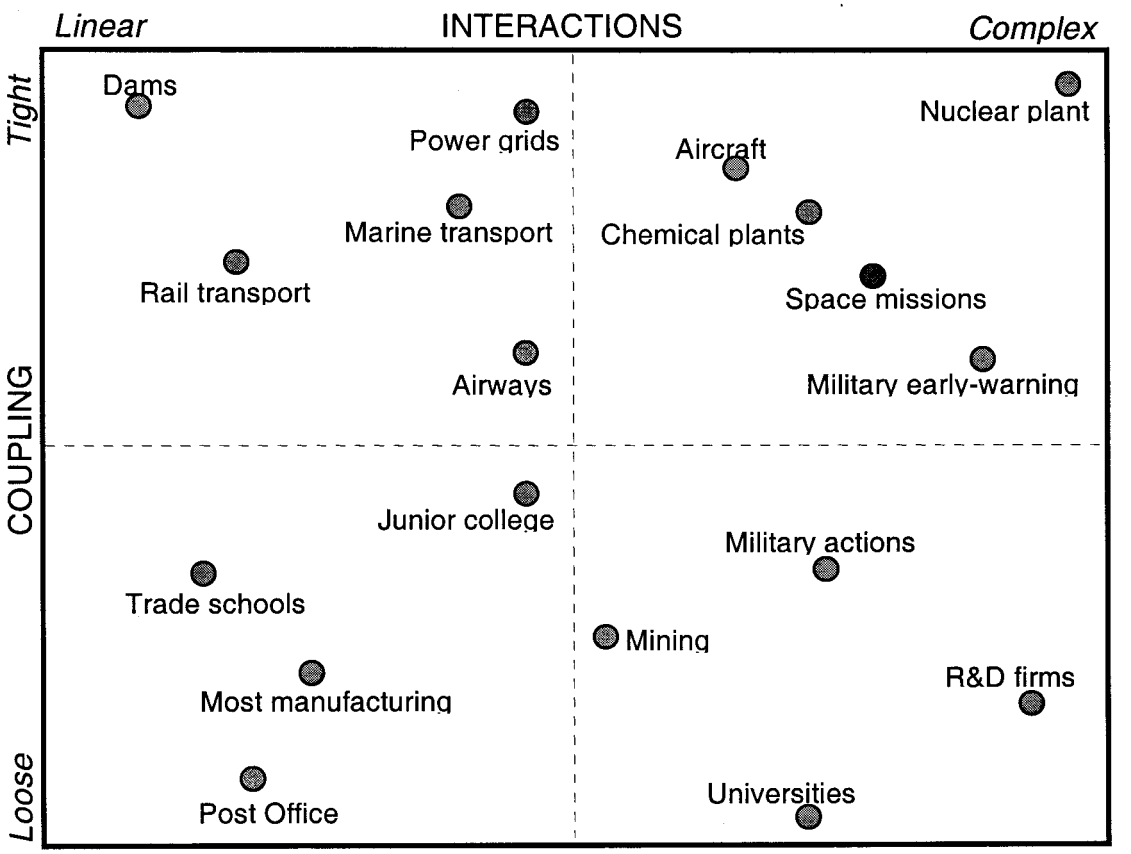

FIGURE 1. High-risk technologies have complex interactions and tight coupling.

Although the objective in verifying autonomous systems is much the same as for conventional systems—verify expected behavior and acceptable performance over a wide range of situations—the methods must change to acknowledge fundamental differences in the control architectures. Transaction-oriented testing methods applied to uplink/downlink communication on sequencer-based spacecraft don't fit well on autonomous spacecraft because so much of the interesting behavior doesn't appear in the downlink at all. Instead, in autonomous spacecraft the behavior of onboard closed-loop control systems becomes a focus of verification, so methods are needed to instrument these control systems and verify that their behaviors stay within specified bounds. It is particularly important to automate the analysis of these behaviors because manual analysis of multiple concurrent control loops would be very tedious and error-prone.

The remainder of this paper describes one approach to automating part of the job of verifying autonomous goal-driven control systems, particularly discrete-event controllers. The approach is based on three elements: a small language for specifying expected behavior, a compiler and class library for generating embedded runtime behavior auditors from specifications, and instrumentation of the application code for access by the auditors. The tool described here automates one part of a verification process; it does not address equally important problems such as test-case generation or analytic verification via model checking (Lowry, 1997).

RUN-TIME VERIFICATION

In current practice, requirements are levied on software by a variety of people including mission designers and system engineers. Such requirements are usually expressed in natural language and are not directly usable for testing. To address that problem we have developed Tspec, a behavior specification language for non-programmers. Tspec uses a notation that spacecraft system engineers find more intuitive than linear temporal logic. These forms are compiled into a conventional programming language as embeddable behavior auditors. These behavior auditors are designed to report not only the occurrence of *unexpected* events & conditions but also the absence of *expected* events & conditions. Auditing is triggered by discrete events such as the updating of a state variable or expiration of a timer. The kinds of behavior violations detectable with these auditors include conditions like 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 is aimed 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* with mission software. This gives them access to potentially all software state variables and 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 due to hardware or software failure. This changes the concept of system testing from "checking the log files" to one of 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*". The concept of embedded auditors is not a new idea; the call-processing software in AT&T's highly reliable (and autonomous) #4 Electronic Switching System of the 1960s contained embedded auditors that alerted engineers to software misbehavior.

Tspec Language

The auditors are generated from specifications of expected/acceptable behavior written in a little language named Tspec. Tspec provides a few simple constructs for expressing common forms of discrete-event behavior expectations. These specifications are written in a notation that is more intuitive to spacecraft system engineers and software designers/developers than linear temporal logic. The Tspec language currently offers four intuitive constructs—invariants, state machines, episodes, and resource constraints—described below.

- *Invariant.* An invariant is a logical condition that should always be true. Invariants, as well as pre-conditions and post-conditions, are particularly useful in verification because they are generally *much* easier to specify than the application logic that is supposed to enforce them. The invariant below specifies a camera safety requirement: never allow the lens cover to be open when the cone angle between the camera boresight and the sun vector is less than 0.1 radian. The associated auditor evaluates the invariant condition every time any of its variables (lensCoverOpen and sunConeAngle) is updated; if the invariant is violated the auditor generates a notification. Unlike inline tests such as C 'assert' macros, invariant specifications are not position-sensitive in the application code due to the way that variables are instrumented.

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

- *State machine.* State machines specify valid behavior for a potentially infinite sequence of state transition activity. The state machine specification below specifies constraints on a traffic light controller's behavior in terms of legal transitions, state durations, and expected transition rate. The associated auditor checks these constraints every time the state variable (color) is updated. The auditor reports violations such as a green-to-red

transition, or a red state duration less than 15 seconds, or a transition rate of more than 10 transitions in any 300-second interval.

```
statemachine TrafficLight ( color ) {
  transitions {
    red    -> green,
    green  -> yellow,
    yellow -> red
  }
  limits {
    duration( red,    15, 60 )
    duration( green,  11, 58 )
    duration( yellow,  2,  4 )
    rate( 4, 10, 300 )
  }
}
```

- *Episode.* An episode specifies a fragment of behavior having a beginning event and ending event, with any number of intermediate events. The episode specification below specifies requirements on a science measurement activity in terms of an expected sequence of steps, required conditions, and timing constraints. The associated auditor checks the progress of the episode every time one of its variables is updated. The auditor reports violations such as an out-of-sequence update (e.g. `SAVED` precedes `FULL`), a required condition that became false (e.g. `power != ON`), or a duration between the first and last steps that exceeds the maximum duration (e.g., more than 150 seconds). Episodes that remain unfinished at the end of a test run are also reported.

```
episode MeasureField (inst_state, SSR_state, power ) {
  steps {
    update( inst_state, READING )
    update( inst_state, FULL )
    update( SSR_state,  SAVED )
  }
  require {
    power == ON
  }
  limits {
    duration( 90, 150 )
    end_begin_delay( 60, infinity )
  }
}
```

- *Resource.* Resource specifications state the type and amount of resources available and the conditions under which they are consumed. A violation occurs if a resource limit is exceeded. Resources are divided into "depletable" (e.g., propellant) and "renewable" (e.g., power from a solar panel). The example below specifies the total amount of electrical power (150 watts) and two conditions under which that power is consumed (e.g., the camera draws 6 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 would report if the aggregate state of the system ever implies greater than 150 watts of power consumption.

```
resource power {
  options {
    Renewable = yes;
    Quantity = 150;
  }

  when (xsspa_pwr == ON)  then consume(power, 45);
  when (camera_pwr == ON) then consume(power, 6);
}
```

Instrumenting the Code

Tspec specifications are compiled into a conventional programming language (e.g. C++) which is then compiled and linked in with the operational software. Obviously, the operational software must be instrumented so that the behavior auditing code has access to the required variables. In an object-oriented design this can be accomplished in an unobtrusive way through the *Observer* design pattern (Gamma, 1994). This mechanism provides loose coupling between operational code and auditor code, with no change in operational code as auditor code is inserted or removed. In fact, the Tspec language provides for the definition of observable variables that are compiled into C++ classes. The operational code may then use these class objects—through *set* and *get* methods—in place of defining the classes by hand in C++.

Currently, Tspec specifications are being compiled into C++ for embedding in software for the X-33 Avionics Flight Experiment at JPL. In this experiment the auditors will be checking behavior visible in a telemetry stream.

Software Development Practices

Software development practice should change in two important ways. First, the concept of "software delivery" should be expanded to include not only the operational code but also the associated verifiable behavior specifications. When a developer is given initial requirements for a software subsystem, he/she should begin by expressing those requirements as verifiable behavior specifications. In effect, we encourage designer-developers toward a "left-brain/right-brain" split where the left brain thinks about how to specify behavior of a subsystem while the right brain thinks about how to implement the desired behavior.

Also, test engineers should conduct inspections of 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 detect misbehavior; early-automated detection makes the debugging job vastly easier.

LIMITATIONS

This paper has focused on run-time verification of an implemented system, which stands in contrast to analytic verification where design-time models are logically/exhaustively checked for undesired behaviors. On the positive side, run-time verification is applied to the implemented system rather than to a design model. This enables run-time checking to detect implementation errors, and it enables checking of behavior at a much more detailed level (since tractable models for model checking are typically an abstraction of the design). On the negative side, behavior is checked during system execution and is therefore limited to the relatively few traces that get exhibited during scenario-based system testing and actual operation. Thus, run-time verification should be viewed as a partner to design-time model checking, not as an alternative.

RELATED WORK

Tspec's auditors are termed *oracles* in the larger field of specification-based testing (Richardson, 1992). As Richardson et al emphasize, oracles for reactive systems must consider not only functional but also timing and safety requirements, and different computational paradigms must often be used to specify such systems. Tspec's constructs are just four of potentially many kinds of behavioral specifications, and their format is intended more for user understandability/simplicity than for expressive generality.

Temporal Rover (Time-Rover, 1998) is a product intended for formal testing of reactive systems based on temporal logic specifications. Using temporal logic's *future* operators such as 'Next', 'Always', 'Sometime', and 'Until'—written as (), [], <>, and U, respectively—safety and liveness properties can be expressed as stylized comments in source files. The source files are then preprocessed to convert the comments into source language statements, and then the resulting source files are compiled in the ordinary way. Whenever execution passes through a commented area, the inserted code is executed to check for violation of specified properties. Tspec differs from Temporal Rover

in two primary ways. First, the Tspec language deliberately does not emphasize temporal logic operators because it was felt that such constructs would be less intuitive to the intended audience of spacecraft engineers than constructs such as invariants, state machines, and episodes. Second, Tspec relies on an Observer design pattern (Gamma, 1994) to ensure that a specification gets reevaluated every time any of its variables changes value. This allows behavioral specifications to be cleanly separated from implementation and eliminates the need to duplicate a specification every place where any of its variables get updated.

SUMMARY

The architecture of executive control software for spacecraft is changing from that of timed sequencers to goal-driven multi-threaded closed-loop reactive control systems. With numerous concurrent, interacting closed-loop control systems operating at different time scales, such systems exhibit very complex behaviors that are impractical to verify using transaction-oriented testing on uplink/downlink channels and very difficult to verify through manual inspection of execution logs. It's more helpful to recognize that much of the run-time verification of control systems can be automated—and even performed in real-time—by embedded behavior auditors that track observed behavior against models of expected behavior. Such models can vary widely in type (e.g., discrete-event, continuous dynamics, stochastic) and level of detail, but in all cases they express requirements in a directly verifiable form. In practice, these embedded behavior auditors become part of a spacecraft's overall fault-detection design, deliberately blurring the distinction between pre-launch testing and in-flight monitoring.

This paper has illustrated the concept of embedded behavior auditors through a small language and library, named Tspec, that allows system engineers and designers to specify expected and acceptable behavior in the form of invariants, state machines, episodes, and resource constraints. Just as pre-conditions and post-conditions in Eiffel check an object's run-time behavior against a design contract (Meyer, 1997), so also do Tspec's testable specifications define and check for contract violations. Importantly, the very existence of such contractual behavior specifications in a project encourages a form of code inspection where only the comparatively brief behavior specifications are inspected for accuracy and coverage rather than a comparatively large body of operational code.

ACKNOWLEDGEMENTS

This paper describes work carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Kirk Reinholtz contributed many formative ideas that shaped this work.

REFERENCES

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* **8**, 231–274 (1987).

Lowry, M., Havelund, K., and Penix, J., "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.

Meyer, B., *Object-Oriented Software Construction, Second Edition,* Prentice Hall, 1997.

Perrow, C., *Normal Accidents: Living with High-Risk Technologies*, New York, Basic Books, 1984, pp. 72–100.

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

Rosenblum, D. S., "A Practical Approach to Programming with Assertions," *IEEE Transactions on Software Engineering*, **21**(1), 19–31 (1995).

Time-Rover Corp., "The Temporal Rover," http://www.time-rover.com (1998).