

# The TSTAR Autonomy Test Tool

Kirk Reinholtz and Dan Dvorak  
Jet Propulsion Laboratory  
California Institute of Technology  
4800 Oak Grove Drive  
Pasadena, CA 91109  
first.last@jpl.nasa.gov

June 15, 1997

## Abstract

The new breed of autonomous goal-driven spacecraft contain much more onboard capability than their sequence-driven predecessors, demanding corresponding advances in software verification techniques. Although autonomous systems are deterministic, they are highly sensitive to the environment, such that the response of a system in certain contexts must be explored in detail in order to provide confidence in both the design and implementation. We describe a system verification strategy and tool based upon the automatic generation and execution of a large number of tests that are “near” a given nominal mission scenario, and a novel use of formal methods to analyze the test results. Results from verifying one software system bear out the benefits of using formal behavior specifications.

## 1 INTRODUCTION

NASA is moving into an era of increased spacecraft autonomy[1]- a natural outcome of a desire to reduce the cost of science data combined with the impact of light-time communication delays and the availability of ever more powerful space-capable computers.

---

<sup>1</sup>The work described was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

Autonomy has the potential to decrease the cost of spacecraft operations, improve reliability, and provide increased science product volume and quality. However, before these things can occur, we must provide a compelling argument that we can address reliability concerns over the full product lifecycle.

Traditional spacecraft flight software testing basically demonstrates that each command works correctly, and that combinations of commands that are likely to be used during the mission work properly together. This has been appropriate and effective, because the systems are designed to minimize the influence of environmental factors on the execution of low-level commands.

However, almost by definition, as the degree of autonomy increases, the sensitivity to the environment also increases<sup>1</sup>. Since the system is sensitive to the environment, and the actual mission environment can't be predicted with sufficient accuracy, one must

---

<sup>1</sup>“Autonomy” is often described as “closing more loops on-board”, but it may also be viewed as an online optimization problem (e.g. minimize resource consumption, maximize science return, optimize a schedule) where the environment has substantial influence on the outcome of the optimization. It is the nature of discrete optimization problems that they can be very sensitive to parameters in the sense that a seemingly small change in the input can cause a large and “non-linear” difference in the output. For example, a change in the length or time of an event of a fraction of a percent can cause a planner algorithm to emit a very different plan. We make this distinction in order to stress the importance of state exploration.

explore the behavior of the system over a range of plausible environments in order to gain confidence in the robustness of the system.

The tools and techniques we propose automatically generate a large number of plausible environments, explore the response of the system over the environments, and characterize the robustness of the system. Our approach is based upon dynamic comparison of the state trajectory of the system being tested against mathematically rigorous (formal) descriptions of the expected behavior of the system. We propose to use both black-box (i.e. description in terms of initial state, inputs and outputs only) and white-box (internal details of the software execution are exposed) descriptions, because the closed-loop nature of the algorithms will tend to mask some faults and so render black-box testing incomplete.

We begin with by describing how the differences between conventional and autonomous spacecraft impact the test and verification problem. We then outline our proposed tool suite and show how it addresses the issues. Finally, we describe the results analysis component in some detail, including some sample specifications.

## 2 WHY TEST?

Ultimately, our objective in testing is to improve the expected science return, and so minimize the incremental cost of science data. Premature spacecraft failure is the biggest threat to science return, so a significant part of autonomy software tries to protect the spacecraft against onboard failures and self-destructive commanding<sup>2</sup>. Unfortunately, since an autonomous system by definition has substantial control over its own fate, it follows that the spacecraft is highly vulnerable to mistakes within the autonomy design and implementation, and so should be heavily exercised.

If we stipulate that the spacecraft can't be com-

---

<sup>2</sup>Spacecraft have always had a powerful on-board fault protection capability. Modern autonomy software enables greater fault coverage and responses that are more likely to allow the mission to progress without human involvement in the recovery process.

manded to cause itself permanent harm, then the next biggest threat to science data return is to command the spacecraft to do something of low science value. How to provide a measure of the science value of all possible observations *a priori* is at best an open research question, because of the difficulty of capturing and encoding the full trade-off space.<sup>3</sup> It will thus be a long time before it will be rational to allow an autonomous system make significant decisions as to what science data to acquire<sup>4</sup>. It will therefore remain important that we have a method of confirming that commands to the spacecraft will provide good science return, even if maintenance of spacecraft health is no longer a concern.

## 3 TOOL SUITE

Our tool suite, named TSTAR, covers system testing tasks ranging from automatic generation of test cases to automatic results analysis. The purpose of each component is outlined below. In particular, this paper focuses on results analysis, so TAUDIT is described in greater detail later in this paper.

**Tgen** – Test-case generation. This component will derive a number of plausible operational scenarios, given as input a single nominal scenario and certain perturbation criteria. For example, it may be given as input an encounter with an asteroid, which it will use to compute a large number of plausible ways in which the encounter may actually occur; it will vary timing relationships, faults, and resource consumption. It also derives test-specific pass/fail criteria, which is forwarded to TAUDIT, where it is used to analyze the results of the tests.

---

<sup>3</sup>For example, the Mars Pathfinder scientists (dozens of them) spend much of the night negotiating the next day's observations. It is apparent that the science data is important to them, and seems unlikely that they would entrust these decisions to an algorithm. We also speculate that it would be very difficult to extract their decision processes.

<sup>4</sup>We must decrease the cost of spacecraft, and improve our ability to encode the trade-off space, until it's cheaper (per unit of science return) to build them with built-in science decision making than to command them during the mission.

**Texte** Test execution. This component will provide a uniform interface by which test execution is controlled. The actual execution takes place upcm an existing simulator, testbed, or other vehicle, typically provided by the project that built the system that is being tested.

**Taudit** Results analysis. Test execution results are analyzed by this component. It infers the state trajectory of the system being tested by the contents of various logs and messages generated during the test, and then validates that trajectory via format predicates derived from various sources including flight rules, design rules, and test input.

**Tvis** Visualization. This component will provide visualization features for exploring the results of the results analyzer. In particular, we conjecture that proximity to failure will prove difficult to summarize, and so **Tvis** will provide for interactive exploration of this and other results.

## 4 Taudit

The purpose of **TAUDIT** is to check the state trajectory of the software under test against a mathematical specification of correct behavior, and to report any discrepancies. Although **TAUDIT** is a component of the **Tstar** suite of automated test tools, it may also be used standalone to validate a system against formal specifications.

The ideal is that **TAUDIT** is used with a programming discipline that generates code from specifications, and specifies additional axioms as the code is written and yet others specific to a particular test case. Sufficient state trajectory would be exposed that **TAUDIT** could then be used to confirm that there are no violations of any of the axioms. Of course, this is not always possible, and may not even be cost-effective, depending on the application [2, §2.1]. **TAUDIT** does not enforce this ideal, and may be used over a wide range of rigor: from simple application as a heavy-duty assertion notation, all the way to full formal specification with critical aspects proven and all checked dynamically.

**TAUDIT** supports and encourages an assertion [3] or annotation [4, 5] style of programming, whereby assumptions made by the programmer are captured in the form of pre and post conditions [6] and dynamically checked. Ideally (but rarely, if ever), one proves that the pre and post conditions are complete, consistent, and imply the correct operation of the program. **TAUDIT** provides confidence in the pre and post conditions at much lower cost, because it confirms that a given execution does not violate the conditions, rather than attempting to *prove* that they are never violated.

The specification of the system to be tested is written in the formal language of **TAUDIT**. The mathematical logic of **TAUDIT** is a roughly a first order logic without quantification, with a rich set of relational, arithmetic, logical, and bit operators, and several useful datatypes (boolean, number-theoretic integers, floats, sets, and enumerated types). It also includes operators to gain access to the previous value of an expression and to detect a change in the value of an expression, and the time at which those events occurred, which together make it natural to write an concise axiomatic specification of the system without resorting to temporal logics. Non-recursive functions are provided for expressive convenience, but do not add to the power of the notation. **TAUDIT** includes assignment, but assignment operates “outside” of the mathematical logic and is intended to be used to synthesize variables out of complicated functions on the input, which are then operated upon from within the logic.<sup>6</sup>

Our intent is that a suite of domain-specific languages will be developed in the spirit of e.g. Larch/LCL [7], which are easier to use than **TAUDIT**, and are compiled into **TAUDIT**. We thus get the best of two worlds: users can work in domain notations, and **TAUDIT** can work in a single formal domain<sup>7</sup>. A single improvement to **TAUDIT** leverages into benefits

<sup>5</sup>Or translated into.

<sup>6</sup>Our intent is to avoid customizing the tool to particular applications by giving the users access to significant computational capability that can be used to transform input data before processing by the bulk of **Taudit**.

<sup>7</sup>In fact even **Taudit** has two levels of notation: a syntactically rich infix notation for humans, and a lisp-like prefix notation for automated manipulation.

for multiple user don'ts. Also, TAUDIT provides a mathematical rigor that could be excessive for some applications and can be hidden by the higher degree of abstraction that may be provided by domain-specific user friendly notations. This approach was used in Larch to good effect.

#### 4.1 Examples

Spacecraft are constrained by what are called "Flight Rules". These rules generally express conditions that must always or never occur. Typical flight rules might be "Never point the camera at the sun", "Always keep the antenna within 0.1 radians of the Earth", "The fuel heater must be on for the thirty minutes prior to operating the engine". We can easily encode these and similar rules:

```

#! Camera cone relative to the Sun must
#! always be at least 0.2 rads.
invariant camera1
  camera_sun.cone > 0.2;

#! Antenna angle relative to the earth
#! must always be less than 0.1 rads.
invariant antenna1
  antenna_earth_cone < 0.1;

#! When the engine is turned on, the
#! heater must have been on for at
#! least 30 minutes.
invariant engine3 @T(engine_on)
  -> heater_on
    & (now-tup(heater_on)) >= 30;

#! The clock must advance by at most
#! 1/8 second.
funcdecl diff(_a1) _a1 - prev(_a1);
invariant tick
  0 <= diff(clk)
  & diff(clk) <= 0.125;

```

Constraints upon the state of the system are expressed as invariants, as can be seen in these examples. Each invariant starts with the "invariant" keyword, followed by the name of the invariant (used to

uniquely identify the invariant) and then a boolean expression. TAUDIT will confirm that the boolean expression is true for all observed values of the state.

The function "@T()" is true only in the time instant at which the argument becomes true. Similar functions detect other change conditions. The function "tup" returns the time at which the boolean argument last became true. "tdn" and "tch" can access the time at which an expression became false or changed, respectively. TAUDIT provides the variable "now", which is the current time.

Assignments operate within the context of guarded commands, as can be seen in the following example. The variable "firstsubframe" is provided by the system and is true only for the first iteration through the guarded commands at any time instant. Invariants may be applied to the state trajectory that occurs while the guarded commands are cycling.

```

guardedcmd d1a
  firstsubframe & @C(N) & N > 0:
  go := T;
guardedcmd d1b
  go:reps := N;
guardedcmd d1c
  go:rslt := 1;
guardedcmd d1d
  go:go := F;
guardedcmd d1e
  (reps > 0):
  rslt := rslt * reps,
  reps := reps - 1;

```

We have demonstrated the use of TAUDIT to check the execution of a software emulator for a microcontroller (the Intel 18085.4). Each opcode was axiomized, and then tests that exercise all opcodes were executed. The following is a somewhat more detailed example. Not shown are some simple axioms that show that all arithmetic is modeled in terms of natural numbers, nor are the (many!) declarations shown.

```

#! eight-bit twos-complement addition.
funcdecl add8(_a1, _a2) (_a1+_a2)%256;
#! 1 if _a1, else 0.

```

```

funcdecl bv(_a1) (_a1)?1:0;

funcdecl addcommon(_a1, _a2)
  rA <- add8(rA, _a1)
  & fCY <- bv((rA+_a1)>255)
  & fS <- bv(add8(rA, _a1) > 127)
  & fZ <- bv(add8(rA, _a1) = 0)
  & fP <- bv(even_parity(add8(rA, _a1)))
  & fAC <- bv((rA%16+_a1%16) > 16)
  & nc(_a2, {rA}, {});

#! ADD r          5-6
invariant ADDR   op_nns(#b10000000) ->
  addcommon(rSSS ,1);

#! INR r          5-8
invariant INRR   op_ndn(#b00000100) ->
  rddd <- add8(prev(clk,rddd),1)
  & fS <- bv(add8(prev(clk,rddd), 1) > 127)
  & fZ <- bv(add8(prev(clk,rddd),1) = 0)
  & fP <- bv(even_parity(add8(prev(clk,rddd),1)))
  & fAC <- bv((prev(clk,rddd)%16+1%16) > 16)
  & nc(1, {rddd}, {});

```

The operator “←” is semantically and syntactically equivalent to the equality operator “=”, but (to date, informally) conveys the additional information that the left hand side may have changed during the execution of the opcode, but the right hand side must not have changed. The “nc()” function explicitly states what values are allowed to change: all other state must have remained unchanged.

The careful reader will also note that “prev()” has another argument. The first argument is used to indicate when to latch the previous value: when the first argument changes, the value of the second argument is latched. We found that this makes it much easier to write robust specifications, since the time epochs can be controlled from within the specification, rather than by when the state variable values happen to be emitted from the system under test.

## 4.2 Related work

There are several other systems in the literature that share our goal of closing the gap between a formal

specification and the concrete program it specifies by verifying program execution against the formal specification. In this section we will compare several of them with our tool,

**ACL2[8]** ACL2 is an interesting tool that uses an executable formalized subset (including recursive functions) of CommonLisp[9] as its notation. It has a sophisticated theorem prover, TAUDIT uses a very similar subset of CommonLisp. TAUDIT is restricted to total functions, where ACL2 allows the use of partial functions at the cost of reduced reasoning capability. TAUDIT does not have recursive functions, though they could easily be added: we don’t provide them because we’re trying to simplify automated reasoning. ACL2 does not directly support the dynamic verification goal of TAUDIT, but see no reason why it couldn’t be easily adapted to that use by executing the specification and code on the same inputs and comparing the results of the two computations.

**ADL[5]** ADL is used to specify the post-conditions of functions, and can check that for a particular execution, the post-conditions do indeed hold true. Its primary purpose is to formally specify the semantics of functions written in e.g. C++. The specification is separate from the implementation of the function, as it is for TAUDIT. ADL also has one base language and multiple domain languages, as does TAUDIT. ADL does not appear to have some of our concepts e.g. “@T()”, latching “prev()”, and “tch()”, and so is less expressive in those quasi-temporal areas. It gains full expressiveness via recursive functions, where TAUDIT does so via guarded commands that operate outside of the TAUDIT logic. ADL is generally richer in syntax and modularization than TAUDIT: TAUDIT is rather spartan and concise in comparison<sup>8</sup>. ADL does not seem to have support for automated proof systems. We believe that automated proof support is important to a full lifecycle specification system such as TAUDIT.

<sup>8</sup>Our design includes modularization and scoping, but those features have not been implemented.

DIT, and so decided to limit the expressiveness so as to simplify the construction of an automated prover. ADL supports partial functions, where TAUDIT does not<sup>9</sup>.

**Anna[4]** Anna uses a subset of the target programming language (Ada) with a few additions (including a form of multi-sorted quantification), to formally annotate the program with predicates in such a manner that the predicates can be checked during runtime. The specifications are placed in the code in the form of formal comments, where in TAUDIT they are separate from the code. Anna is purposefully nearly the same language as the program being specified, where TAUDIT is explicitly not the same language. Anna has significant machinery to extract the necessary runtime information to check the annotations, where TAUDIT requires that this is done manually. Automated proofs are not explicitly supported, but are not excluded except perhaps for the complications of typing, partial functions, and quantification.

**Larch[7]** Larch is a specification notation and proof system. Specifications are generally written in a notation specific to the programming language of the system being specified, which are then converted to and analyzed in the Larch back-end language, L<sub>1</sub>. L<sub>1</sub> specifications are not intended for dynamic comparison with program execution, but rather towards the use of proof techniques to support assertions made about the system. L<sub>1</sub> is a first order logic notation.

**SEQ-REVIEW[11]** SEQ-REVIEW is a production-quality and heavily used multi-purpose file browser that can parse a wide variety of files, display projections of the contents in various graphical and textual forms, and perform specialized constraint checking. It has a built-in recursive programming language similar in spirit to AWK, that can perform

---

<sup>9</sup>We recognize that there are good arguments in defense of partial functions[10], but made a decision biased towards simplifying the model of our logic. We will revisit this decision if it becomes a major problem.

arbitrary computations on the input. It would be possible, but generally very inconvenient and inefficient, to use SEQ-REVIEW to do the same checks as performed by TAUDIT. TAUDIT is positioned as a formal specification and dynamic analysis tool based upon mathematical logic and intended to support automated reasoning, where SEQ-REVIEW is positioned as a sophisticated multi-purpose file browser that performs some specialized constraint checks.

**SRLT[12]** SRLT performs white-box testing, using the SPIN model checker[13] to verify execution traces. It computes equivalence partitions of potential execution traces, and generates test cases for partitions that have not been tested. SRLT is primarily intended to test distributed systems for e.g. deadlock, race conditions, etc. SRLT uses the same white-box approach as TAUDIT, in fact the SRLT logging facility could be used to generate data for TAUDIT. SRLT is positioned to test against behavioral specifications of interacting state machines, where TAUDIT works with algebraic specifications.

## 5 PROJECT STATUS

TAUDIT has been implemented and is being used to verify a software implementation of a microcontroller (the Intel 8085.4). We found that writing the specification was about one fourth as much work as was the actual implementation of the emulator. Our original goal was to apply it to a remote agent spacecraft, but schedule differences have so far prevented us from doing so.

The remaining tools of the Tstar project have been discussed or designed to some level of detail, but none have been implemented.

## 6 FUTURE WORK

**Add automated reasoning** TAUDIT is designed to support automated reasoning, primarily because we believe that our intended application will result in specifications so large that we will need

automated tools to gain confidence in the specification itself.

**Fly it as a hypervisor**[14] Although TAUDIT was originally conceived as a system verification tool for pre-flight testing, its small size and efficient event-driven checking make it suitable to serve as an onboard in-flight behavior monitor. Any violations detected during flight would alert the ground that the spacecraft is behaving outside of an acceptable envelope.

**Add state-oriented notation** TAUDIT is able to track observed behavior against a state-transition description, but it requires a tedious group of guarded commands. We plan to add a construct to simplify specification of statecharts.

## 7 SUMMARY

Testing spacecraft systems that have a high degree of autonomy requires new testing techniques, because of the higher sensitivity to the environment that such systems, by definition, exhibit. TSTAR addresses the issues by automatically generating a large number of plausible scenarios "near" a given mission profile, and validates the execution of each against a formal specification of correct behavior. One component of the suite, TAUDIT, has been implemented and successfully applied to production code.

## References

- [1] S. Hedberg. AI Coming of Age: NASA uses AI for Autonomous Space Exploration. *IEEE Expert*, pages 13-15, June 1997.
- [2] J. Rushby. Formal Methods and Digital Systems Validation for Airborne Systems. Technical Report NASA Contractor Report 4551, NASA, 1993.
- [3] D.S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, 21(1):19-31, January 1995.
- [4] D. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*. Springer-Verlag, 1990.
- [5] S. Sankar and R. Hayes. ADL - An Interface Definition Language for Specifying and Testing Software. *SIGPLAN Notices*, 29(8):13-21, 1994.
- [6] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [7] J.V. Guttag and J. H. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [8] M. Kaufmann and J.S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203-213, April 1997.
- [9] Jr. G. Steele. *Common LISP: The Language*. Digital Press, Redford, Mass., 1984.
- [10] D.L. Parnas. Predicate Logic for Software Engineering. *IEEE Transactions on Software Engineering*, 19(9):856-862, September 1993.
- [11] P.F. Mالدague. SEQ-REVIEW: A Tool for Reviewing and Checking Spacecraft Sequences. Technical Report NASA TR 19950011146N, NASA, November 1994.
- [12] J. Callahan, S. Easterbrook, and F. Schneider. Automated Software Testing Using Model Checking. In *Workshop on Living with Inconsistency at the International Conference on Software Engineering (I CSE)*, Boston, Ma., May 1997.
- [13] G. Holzmann. The Model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279-295, May 1997.
- [14] T.C. Bressoud and F.B. Schneider. Hypervisor-Based Fault-Tolerance. *ACM Transactions on Computer Systems*, 14(1):80-107, January 1996.