

DS-1 Pilot Implementation of Assertions: Summary of Lessons Learned

December 19, 1997
Distribution Version



Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

DS-1 Pilot Implementation of Assertions: Summary of Lessons Learned

Prepared by:

Robyn R. Lutz
Task Lead

Hui-Yin Shaw

Approval:

Tuyet-Lan Tran
Software Assurance Supervisor

John Kelly
ATPO Software Applications Program, PEM

December 19, 1997
Distribution Version



Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

DS-1 Pilot Implementation of Assertions: Summary of Lessons Learned

Robyn R. Lutz and Hui-Yin Shaw
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

January 28, 1998

- 1. INTRODUCTION4
- 2. DESCRIPTION OF DS-1 IMPLEMENTATION AND RESULTS.....4
 - 2.1. SOFTWARE REQUIREMENTS DOCUMENT.....4
 - 2.2. SOFTWARE FAULT TREE ANALYSIS.....5
 - 2.2.1. IPS Throttle Up.....6
 - 2.2.2. SFTA Discussion6
 - 2.2.3. Assertion Checking7
 - 2.2.4. SFTA Study Summary8
 - 2.3. FLIGHT RULES11
 - 2.4. LISP MODELS.....12
- 3. USES OF ASSERTIONS13
 - 3.1. ASSERTIONS AND TEST LOG ANALYSIS13
 - 3.2. ASSERTIONS AND INSPECTIONS14
 - 3.3. ASSERTIONS AND INTEGRATED MODELING TOOLS.....14
- 4. SUMMARY.....15
- 5. REFERENCES15
- APPENDIX A. "Assertions: Instrumenting Safety Critical Code for Maintenance"
- APPENDIX B. "Safety Analysis of Requirements for a Product Family"

ABSTRACT

This report is an end-item deliverable for the research project "Maintaining Software Safety," funded by NASA's Office of Safety and Mission Assurance under UPN 323-08-5H. It describes the pilot implementation of assertions on the Deep Space-1 spacecraft.

The first phase of this RTOP activity culminated in the production of the report, "Assertions: Instrumenting Safety Critical Code for Maintenance". That report made a case, based on previous studies and industrial experience, for more widespread use of assertions.

In the second phase of this activity, reported here, we describe the lessons learned from our experiences identifying safety assertions on the DS-1 spacecraft for use either as checks embedded in the code or as predicates for use during test log analysis or integrated modeling. The key sources investigated for the assertion-identification process were Software Requirements Documents, Software Fault Tree Analysis, Flight Rules, and Lisp code component models.

1. INTRODUCTION

This report builds on the "how-to" report, "Assertions: Instrumenting Safety Critical Code for Maintenance," that was delivered in January, 1997, for informal review by NASA engineers (Appendix A). That report surveyed the literature on assertions, documented the results of previous usage of assertions in industry, and described how assertions can be used to help maintain software safety.

The pilot implementation of assertions described here draws on that report to explore the sources of assertions and the uses of assertions in a particular project, the New Millennium Program's Deep Space 1 spacecraft at JPL. An updated version of the report, incorporating feedback from reviewers, is provided in Appendix A.

Section 2 below describes the lessons learned from our experiences identifying safety assertions on the DS-1 project. The first focus of the DS-1 implementation was on the sources of assertions. This was chosen as a focus because the DS-1 project expressed interest in how to best identify potential candidates for assertions (i.e., sources). Section 2 describes the results of the investigations of the following possible sources of assertions on DS-1: Software Requirements Documents, Software Fault Tree Analysis, Flight Rules, and Lisp models.

A second focus of the implementation was on the uses of assertions. This focus evolved during the RTOP work, as it became clear that many of the same safety properties and many of the same preconditions, postconditions, and invariants appropriate for insertion in the code as assertions might also be appropriate for code analysis (e.g., test log analysis) or model testing. Section 3 of the report describes the uses of assertions, either as checks embedded in the code, as predicates for use during code inspections and test log analysis, or as input to integrated modeling techniques. Section 3 also provides links to related work at JPL, ARC, the IV&V Facility, and elsewhere that use assertions.

Appendix A contains the how-to report, "Assertions: Instrumenting Safety Critical Code for Maintenance." Appendix B is a paper summarizing the results of a joint study with an industrial collaborator (Rockwell Collins Avionics) on software safety techniques for a reusable product family.

2. DESCRIPTION OF DS-1 IMPLEMENTATION AND RESULTS

2.1. SOFTWARE REQUIREMENTS DOCUMENT

Sections of the Software Requirements Document for the DCIU (Digital Control Interface Unit) of the XFS (Xenon Feed System) were investigated as a source of assertions. The DCIU is part of the NSTAR component of DS-1 which provides ion propulsion. NSTAR is the solar electric propulsion module.

It was found that this Software Requirements Document was a good source of assertions, providing preconditions, postconditions, and invariants that contributed to the continued safe operation of the Ion Propulsion System. Some examples of assertions whose source is a Software Requirements Document are:

1. Preconditions

- (a) Reasonableness checks on inputs.
"Ignore output of drifting pressure transducers."
- (b) Requirements on parameters

"This procedure shall always be called with the command to operate the thruster at power level one."

(c) Sequential events

Cathode conditioning command must always precede thruster ignition command.

"DCIU shall ensure that stable xenon flow exists prior to commanding on the Neutralizer keeper power supply.

2. Postconditions

(a) new-value a function of previous-value

Temperature is the average of the two temperatures that are closest to the average of the three input temperatures.

(b) Next-state predicate

"The DCIU shall enter a known and verifiable state upon Power-On-Reset."

(c) Putative claims: next-state a function of previous-value.

"If previous temperature in plenum tank exceeds nominal by delta-P, then Latch Valve 3 has been commanded closed."

3. Invariants

(a) Watchdog timers

"A maximum of 10 seconds of anomalous flow conditions are allowed before fault protection is invoked."

(b) Reachable states

"In the event of this failure, the Power Processor Unit (PPU) controller shall exit the conditioning procedure and set a heater failure flag in the PPU telemetry. The decision to set this flag shall not be made with only zero indicated heater current, an indicated heater voltage in excess of 9.2 V is also required." (Leaving it unclear what state it's in if current=0 AND voltage <= 9.2)

(c) Predicates on endpoints of an interval

"When the thruster set-point is being changed, until the new plenum tank pressure has been reached, the fault protection trigger points shall be changed . . ."

Since the Software Requirement Document was in English, there were also a few errors such as internal inconsistencies, incomplete requirements, and unclear requirements specifications that had to be resolved during the process of extracting the potential assertions from the text.

The results of this analysis were consistent with our expectation that a detailed textual description of the requirements will provide a rich source of useful assertions. The risk in a requirements document as a source of assertions is that, due to the inadequacies and ambiguities of such a text, incorrect assertions may be mistakenly extracted. However, we recommend that future projects use software requirements documents as a baseline source of assertions of required safety properties that (with sufficient review for correctness) can be inserted in the code.

2.2. SOFTWARE FAULT TREE ANALYSIS

Software Fault Tree Analysis (SFTA) is recommended by Leveson as a possible source of assertions (Leveson). Consequently, two studies of SFTAs were performed. The first study developed SFTAs of three hazards on the Cassini spacecraft (Chen-Tsai, et al.). These SFTAs were produced by analysis of the requirements document and the code.

These case studies provided a testbed for the hypothesis that the leaf nodes of Software Fault Tree Analyses are good sources of assertions needed in the code. In these three case studies it was found that SFTA contributed to the verification effort by identifying code locations for additional exception-handling, but was not an efficient way to locate assertions for code insertion.

The second study of SFTAs looked at throttling-related requirements on the IPS (Ion Propulsion Subsystem) component of DS-1. A required safety condition (that proper pressure be achieved before throttling up began) was identified. The negation of this required safety property then became the root node hazard ("failure to obtain proper pressure") of the SFTA.

The contributing causes to this hazard were examined to identify whether assertion-checking could provide some protection against the combination of circumstances producing this hazard. The hazard can be documented as a comment in order that the safety property will not be accidentally changed as the code is updated. The results and discussion of this IPS SFTA are presented below.

Appendix B documents the results of a related demonstration of software safety techniques in a distinct application. The work described there was performed in conjunction with Rockwell Avionics and Iowa State University. The application domain was a new, modular flight instrumentation architecture for cost-saving in certified product line reuse. As with DS-1, SFTA was applied to the software requirements. In addition, SFMEA (Software Failure Modes and Effects Analysis) and a Safety Checklist for use during requirements inspections of safety-critical software were used in an integrated approach. In both the DS-1 and the flight instrumentation cases, the software safety techniques identified some possible additional requirements for enhanced fault tolerance.

2.2.1. IPS Throttle Up

The IPS is the NSTAR (NASA Solar Electric Propulsion Technology Applications Readiness) technology demonstration element on the DS-1 mission. A required safety condition for IPS throttling up was identified for this study:

“The IPS flight software (FSW) shall ensure that the Xenon Feed System (XFS) pressures are at the proper throttle level defined in the look up table prior to adjusting other parameters of the lookup table.” (paragraph 5.22.1.3.1 of ND-302, an equivalent requirement is also found in paragraph 7.2.4.1 of ND-310)

The negation of this required safety property then became the root node hazard (“XFS pressures not at proper throttle level before adjusting other parameters at throttle up”) of the SFTA. The fault tree symbols from MIL-STD-882A as presented in (Leveson and Harvey) were used in this SFTA (see Table 1). Portions of the source code (May 1997, preliminary version) relating to the throttle logic and the XFS pressures were reviewed, and the SFTA diagrams were constructed (examples are shown in Figures 1 and 2).

This study modeled the SFTA examples provided in (Leveson and Harvey) and (Leveson, Cha, and Shimeall). Both of these articles provided failure-mode templates, which were the SFTA structures for programming language statements (e.g., templates were provided for assignment statements, if-then-else statements, case statements, etc.). It was found that the use of these templates supported the precision required for the SFTA task.

The basic procedure for performing SFTA, as stated in (Leveson and Harvey) and (Leveson, Cha, and Shimeall), is to first identify the failure to be analyzed (the root of the fault tree) and then work backwards, expanding each subnode, until no further analysis can be performed or a basic fault event is reached (the leaves). Each subsequent level of the tree subnodes from the root represents the preconditions with the AND or the OR relationship for the subnode a level above.

2.2.2. SFTA Discussion

In a SFTA diagram, a diamond leaf represents a non-primal event which is not developed further for insufficient consequences. A circle leaf represents a basic fault event and requires no further analysis. An oval leaf represents the state (or condition) of the system that permits a fault to occur. A rectangular leaf represents an event to be analyzed further.

In this SFTA study, the non-primal events (the diamond leaves) include those that are outside the scope of this problem statement, or events that are not likely to happen because they are either non-existent in the source codes or not of concern as result of the development verification process (i.e. faulty command content can be caught easily during integration testing).

From the SFTA diagrams, we can identify the possible causes to the failure of not having the XFS pressures at proper throttle level before adjusting other parameters at throttle up. They are the circle, the oval and the rectangular leaves in the SFTA diagrams (portions of the SFTA result are illustrated in Figures 1 and 2):

1. Bad initial value for power_level (e.g. undefined or corrupted)
2. Corrupted power_level command
3. old power_level data corruption
4. Corrupted previous xfs_control.single_plenum_mode
5. Faulty pressure lookup table (faulty pressure set point)
6. Bad plenum solenoid
7. Faulty plenum transducers
8. Faulty plenum stop_solenoid_valve
9. Faulty plenum transducer algorithm
10. Faulty correct_pressure_required
11. Faulty solenoid_valve_control
12. Faulty leak detection
13. Inappropriate plenum command sent

Items 1 through 5 are caused by data corruptions (e.g., due to space radiation or local memory corruption). The single-bit event upset due to radiation in space can be corrected with the onboard software for the single-bit event monitoring and correction. The possibility of local memory corruption can be investigated but is outside the scope of this exercise. These items are considered basic fault events in this study.

Items 6 and 7 are hardware component failures and are the basic fault events. Items 8 through 12 are failures caused by faulty values or algorithms which require further analysis. Item 13 identifies the condition where a fault can be allowed to take place (Inappropriate plenum command sent). An example of such a condition is the wrongful commanding and execution of a NORMAL_PLENUM command when the system should be in SINGLE_MAIN or SINGLE_CATHODE mode.

Another concern which did not come from the result of the SFTA work, but was brought to our attention when reviewing the throttle logic in the code, was the use of power level as synonym to XFS pressures when determining the throttle mechanism (even though, in normal operation, the XFS pressures are expected to correlate to the set power level).

2.2.3. Assertion Checking

To identify whether assertion-checking could provide some protection against circumstances producing the root hazard of this study, two probable causes were studied: inappropriate plenum commanding (a SFTA leaf from the fault tree analysis) and the possibly erroneous throttle mechanism.

In the inappropriate plenum commanding fault event, there is a concern with the system performing inappropriate mode change. The software version being studied does not check for the system condition when processing the plenum mode change command, therefore through the plenum command processing, the system may be placed in other modes when only SINGLE_MAIN or SINGLE_CATHODE mode is appropriate. To prevent this from occurring, possible assertion of required precondition can be investigated for placement in the source code to prevent inappropriate plenum mode change. In the later version of the IPS software, this anomalous situation is taken care of by the IPS fault protection software.

There is a question about the appropriateness in the use of power level in the throttle mechanism determination. In throttle mode, the new power level is compared with the previous power level to determine whether to throttle up or down.

The Throttle Up Requirement (5.22.1.3.1 of ND-302; 7.2.4.1 of ND-310) indicates that plenum pressures should be adjusted up to new set points prior to adjusting other XFS parameters. The Throttle Down

Requirement (5.22.1.3.2 of ND-302; 7.2.4.2 of ND-310) indicates that other XFS parameters should be adjusted prior to adjusting plenum pressures down to new set points. However, in the IPS code, power level is used as a means to determine the throttle procedure.

Consider the case when the plenum pressure is not at the same set point as prescribed for the power_level commanded (e.g., due to faulty plenum transducer), and a new power command is such that power_level > last_power_level. The program will follow the Throttle Up sequence:

1. If the current plenum pressure > new desired plenum pressure, then the software continues with Throttle Up logic by adjusting plenum pressures **down** to new desired pressure points, before adjusting other XFS parameters. This process contradicts the Throttle Up Requirement.
2. If the current plenum pressure < new desired plenum pressure, then the software continues with Throttle Up logic by adjusting plenum pressures **up** to new desired pressure points, before adjusting other XFS parameters. This is consistent with the Throttle Up Requirement.

Similarly, when the plenum pressure is not at the same set point as prescribed for the power_level commanded, and a new power command is such that power_level < last_power_level, the program will follow the Throttle Down sequence:

1. If the current plenum pressure > new desired plenum pressure, then the software continues with Throttle Down logic by adjusting other XFS parameters prior to adjusting the plenum pressures **down** to new desired pressure points. This is consistent with the Throttle Down Requirement.
2. If the current plenum pressure < new desired plenum pressure, then the software continues with Throttle Down logic by adjusting other XFS parameters prior to adjusting the plenum pressures **up** to new desired pressure points. This process contradicts the Throttle Down Requirement.

The throttle processes identified as contrary to the throttle requirements may pose a potential damaging effect on the health of the thruster engine. Therefore, either rewrite the throttle logic to include plenum pressure factor in the throttle process or place exception handling (rather than assertion) in the code to prevent harm to the thruster engine. An example may be:

```
case THROTTLE_UP_XFS:
{
    IF ( ( xfs_control.main_pa_measure < xfs_control.main_pa_desired ) AND
        ( xfs_control.cathode_pa_measured < xfs_control.cathode_pa_desired ) )
    THEN ... continue with the throttle up process
    ELSE ... error handling or proceed with throttle down process
}
```

2.2.4. SFTA Study Summary

In this SFTA study, the possible contributing causes to the root node hazard include data corruption, hardware faults, algorithmic faults, and a condition in which a failure may occur. One fault event was identified during the review of the source code logic, but not as the product of the SFTA.

The contributing causes to the root hazard were examined to identify whether assertion-checking could provide some protection against circumstances producing this hazard. This study found that, with SFTA leaf nodes, an analyst can easily identify locations in the source code to insert protections. This protection may most likely be in the form of exception handling, which is more appropriate for spacecraft operations, rather than assertions. This is the same conclusion as found in the Cassini SFTA studies (Chen-Tsai, et al.).

SFTA can be a helpful technique for conducting safety-critical code review methodologically (it forces a reviewer to consider every possible fault). The SFTA templates ensure detailed analysis of the root hazard.

The SFTA process that we performed in this study shared the similar analysis technique with Hart’s Logical analysis technique (Hart) in which we analyze the code by:

1. focusing on small code segments that are likely to be the root cause of the hazard, rather than trying to understand the entire program ¹,
2. applying analysis to code slices that affect limited number of variables of interests, and
3. focusing on conditional statements (SFTA templates)

We also found that by including assertions as comments in the code (e.g., adding requirement assertions as comments in code) can facilitate maintenance effort, preserve the safety of the software, and enhance the safety requirements’ traceability to code.

Table 1. SFTA Symbols

Symbol	Usage
	Indicates an event to be analyzed further.
	Indicates non-primal events which are not developed further for lack of information or insufficient consequence.
	Indicates a condition. It defines the state of the system that permits a fault sequence to occur.
	Indicates a basic fault event or primary failure of a component. It requires no further development.
	OR gate: indicates that one or more of the input events are required to produce the gated event
	AND gate: indicates that all input events are required in order to cause the output event

¹ Although it is not required to review the whole program source code, but a complete, thorough software fault tree analysis requires accessibility to all related code modules.

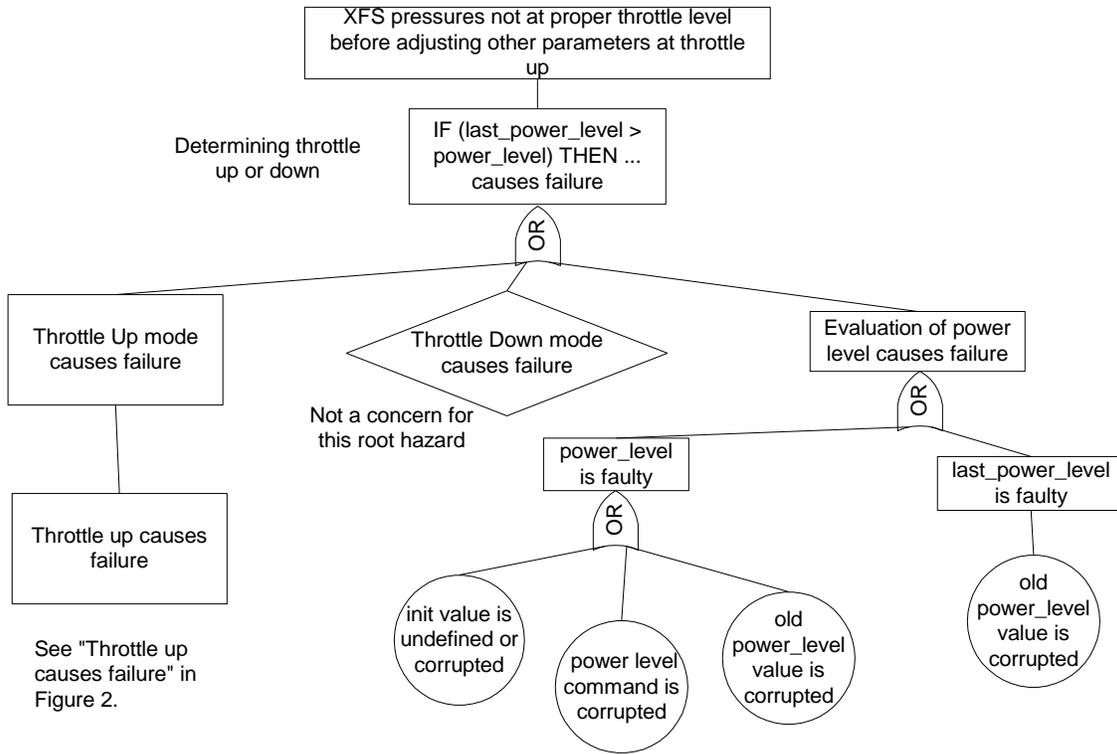


Figure 1. Improper XFS Pressures at Throttle Up

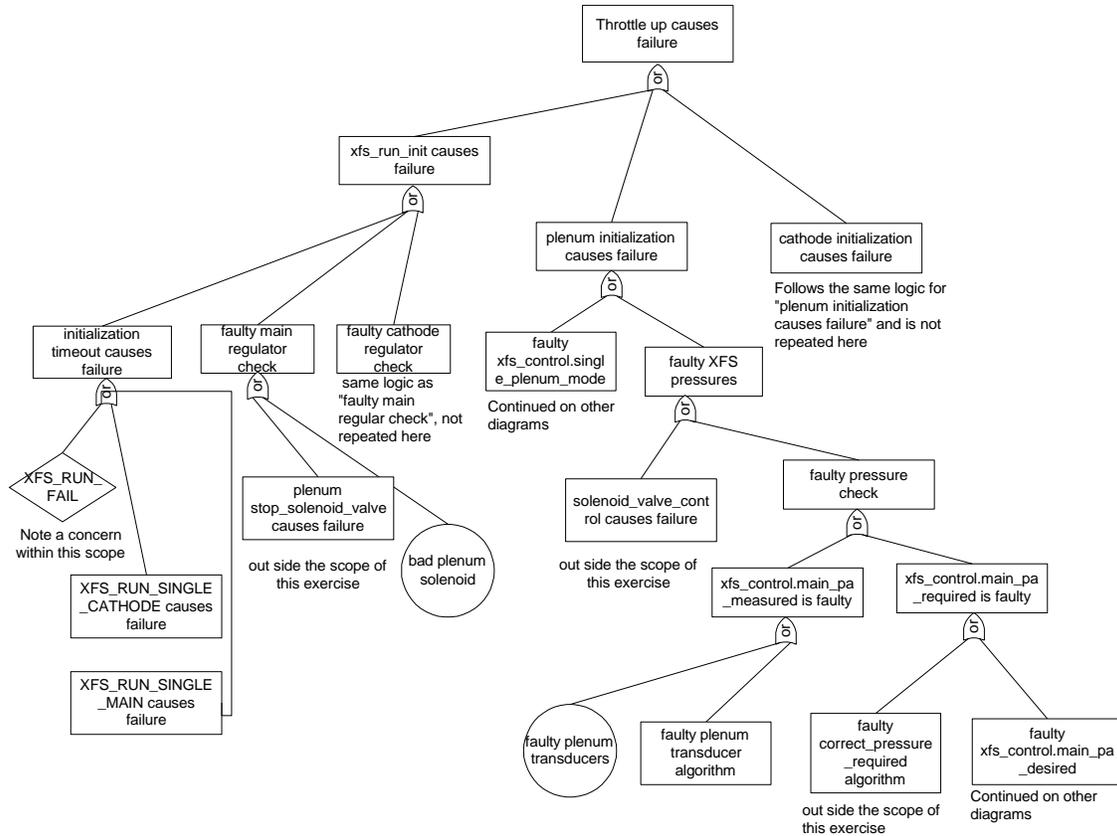


Figure 2. Failure Caused by Throttle Up

2.3. FLIGHT RULES

We had hoped to capture aspects of the relevant flight rules in assertions. However, this effort turned out to be premature and more difficult than anticipated. It was premature since few detailed flight rules had yet been identified for the DCIU/IPS, in part because it was still under development external to JPL. Apart from this, however, we encountered another difficulty in capturing flight rules in assertions. Our interest in assertions is in their potential to assure the safety of the system and to maintain that safety across system updates. To limit risk, we would like to avoid assertions that need to check variables that the software they are embedded in does not already access (information hiding). Flight rules, however, usually encapsulate environmental, system, or historical information that a particular software module does not or cannot know. Thus, flight rules often monitor information that is not accessible to the code in which the proposed assertions would be placed.

For example, one placeholder flight rule that demonstrates both the preliminary nature of the flight rules at that time in our study and the system knowledge needed by any assertion that checked it is "IPS_BURN_POINTING." It states, "The following constraints restrict the direction of the IPS thrusting: 1. Geometric constraints (radiation and optics), such as the MICAS IR radiator; 2. Constraints on gimbal lock coupling (=kinematic amplification factor); 3. Thermal constraints (e.g., PPU's can get hot) (Starbird). To check these constraints in an assertion would require significant reworking of the existing encapsulation of information to allow the assertion's code to access data about the current system state. Martin Feather and Deepak Kulkarni have pointed out that these tradeoffs between the benefits of performing the assertion-checking and the risks involved in the additional access to non-local data and consequent software coupling are also key concerns in current testing research.

To summarize, it appears that some limited aspects of some flight rules (e.g., preconditions to be met before certain actions are taken autonomously) may be appropriate for capture in assertions, but that many flight rules are not appropriate sources of assertions in that planning for assertion-checking would drive the information flow within the code and, hence, require code-restructuring. However, as will be seen below, flight rules appear to be a fruitful source of assertions used to analyze software test logs.

2.4. LISP MODELS

The Lisp code for a preliminary version (R3) of the IPS (Ion Propulsion System) MIR (Mode Identification and Recovery) model was examined as a source of safety-related assertions. Checkik and Gannon, in their COMPASS 96 paper, "Verification of Consistency Between Concurrent Program Designs and Their Requirements," identify three types of safety properties (Checkik and Gannon). These three types of safety properties were used as our guideline for the process of searching for useful safety assertions in the Lisp code.

The first type of safety property is a "reach" property, i.e., that the system can reach a state in which some property holds. This type of property is used to ensure that invariant properties are not satisfied vacuously. The second type of safety property is an "invariant" property that holds in every state. The third type of safety property is "strict cause", meaning that the next transition from each state in which property f1 holds is to a state in which f2 holds. Checkik and Gannon point out that these three types of properties frequently appear in requirements documents, providing possible bridges from the requirements to the testing.

The IPS state changes occur, for the most part, in response to inputs from the DCIU (Digital Control Interface Unit, the microprocessor that controls the Xenon Feed System), informing the IPS of a change in the DCIU's state. The power status of the DCIU (on or off) and the power status of the IPS PPU-LV and PPU-HV (the valves are on or off, i.e., powered or not powered) are also factors in the IPS state changes.

The IPS has eight states:

- Off-or-boot
- Booted
- Standby
- Startup
- Steady-state
- Shutdown
- Beam-off
- Hung.

The DCIU has six states that are reported to the IPS:

- boot
- standby
- startup
- steady
- shutdown
- no-command.

The results of the examination of the IPS Lisp code (Nayak) can be summarized as follows:

1. Properties of the first type (reachability) seem to be less appropriate for capture in code assertions and better checked via other methods (e.g., search of the finite state machine). Examples of properties of the first type would be to verify that each of the eight possible states listed above is reachable.
2. A property of the second type ("property holds invariantly") is that the IPS must be in one and only one of the eight states.

Other invariants can be assembled from the models of the states in the Lisp code to check that the behavior matches the models. For example, "If the power is off, then no command is received" captures the

requirement that commands cannot be received if the power is off. In the Lisp code this invariant property forms part of the model for the Off-or-boot state.

3. There are many properties of the third type ("strict cause, i.e., the next transition from each state in which f1 holds is to a state in which f2 holds") that can be derived from the code, since the Lisp code provides a model describing each state. An example of a case in which Property f1 holds in more than one state, but property f2 holds in only one state, for example, is derived from the requirement that only Startup and Steady-state can transition to Shutdown. One fairly straightforward way to extract these properties from the code would be to make a SCR-type table of conditions and mark which conditions hold in which states. Otherwise, it is laborious to cross-check among the different state's conditions.

The effort to define strict cause may also be useful in investigating exactly what the rationale is for transitions from some states to others. For example, Startup is the next-state only from Steady-state or Standby. In the Startup state a certain property, call it f2, must hold, according to the model. However, in trying to define what the property f1 should be, it turns out that there really is no commonality in the Steady-state and Standby states: it's a one-way street from Standby to Startup, but a two-way street from Startup to Steady-state. Thus, in this case, there is no "strict cause" property of interest. It is interesting, however, that the effort to identify the appropriate assertion raised the question of whether a transition from Startup to Standby was missing.

The following additional possible issues were identified during the process of examining the Lisp code for candidate safety assertions and were reported back to the project:

- a) UNCLEAR. The transition from Standby to Booted is called "Safe", but the transition from Off-or-boot to Booted is called "Booting."
- b) CURRENTLY INCOMPLETE: In this preliminary version, the transitions from Beam Off and from Hung are not yet defined (unless "persist" is not a self-loop, but is itself an additional state).
- c) INCONSISTENT: The documentation says, "If power is cut, the only transition is from standby to off-or-boot." However, dciu-power-loss in the Booted, Startup, Steady, or Shutdown state also causes a transition to Off-or-boot.

To summarize, the Lisp models in the IPS code facilitate the extraction of invariant properties and strict-cause properties (next-state properties). However, a certain amount of "reverse engineering" had to occur to cross-correlate the relevant properties among the various states. A more structured approach to the correlation (e.g., using tables such as SCR or AND/OR) would help reduce the effort needed.

3. USES OF ASSERTIONS

This section describes the uses of assertions, other than as checks in the code, including links to the work of others at JPL, Ames, and the IV&V Facility.

3.1. ASSERTIONS AND TEST LOG ANALYSIS

One unexpected result of our work was the realization that the identification of useful safety-related assertions and the identification of useful predicates to analyze the test logs produced by the execution of the code are often very similar. At JPL, Reinholtz and Dvorak have produced an autonomy test tool (Reinholtz and Dvorak) that expresses constraints upon the state of the system as invariants. The test tool will confirm that the Boolean expression is true for all observed values of the state during the logged test. Callahan et al. at the NASA IV&V Facility have done related work in model checking (Callahan, Easterbrook, and Schneider). Their tool examines the communications between subsystems at their interfaces to support testing activities. At ARC, Mike Lowry has performed model checking of the core of DS-1's Executive using predicates drawn from the Flight Rules to verify its correctness, with his results prompting some change to the Executive.

When Kirk Reinholtz looked at ten sample assertions that we had extracted from the NSTAR requirements document, he found that five of the ten could easily be represented as predicates for their test tool. The

other five of the ten assertions could either probably be represented if enough additional details were provided (e.g., what is meant by a "known and verifiable" state) or could be partially represented (e.g., the tool can represent that output of the pressure transducers is drifting, but perhaps not the requirement that such output be ignored). Thus, at least some of the assertions can be checked either in the code or in the test log analysis.

The testing community, both tool suppliers and researchers, are actively pursuing techniques to incorporate assertions into testing activities. For example, SunTest, a new division of Sun Microsystems Laboratories, lists among its tools for testing Java applications, JavaSpec, which is an assertion based API testing tool. Among researchers, the concept of "perpetual testing," i.e., that software to be maintained properly must be continually tested, can include the merging of assertions into code (see, e.g., <http://www.ito.darpa.mil/Summaries96/E097--UMass.html>).

To summarize, safety assertions to be inserted in the code and predicates to be used in analyzing testing logs are very similar. This suggests that there is some flexibility in where the assertions are placed (code or test tool); the important thing is to use them. Future projects may make a decision to allocate some of the identified assertions to code placement and others to the test tool, depending on the performance overhead (assertion checking can slow execution), impact on data handling (assertion checking can require additional access to non-local data), and security concerns (access to the code may be more restrictive than to the test logs due to reliability concerns).

3.2. ASSERTIONS AND INSPECTIONS

An additional use for assertions that shows promise is for code inspection. Bonnier and Heyer (Bonnier and Heyer), in trying to account for why the use of assertions in industry has lagged when they are so clearly one of the keys to safe and correct software, recommend that informal predicates (with formal syntax but an informal semantics) be used for code inspection. This tends to provide an immediate payback in terms of enhanced code correctness, which in turn can lead to semi-formal or formal specification of those same or additional assertions. In addition, assertions can be formulated in terms of higher-level abstract data types rather than, or in addition to, subsequent low-level implementation invariants. Informal justifications of why each predicate holds can also be given, providing a valuable knowledge-base for future maintenance of the software.

The advantage of initiating the use of assertions for code inspection is that it is easy to put into practice while still allowing for the subsequent automatic derivation of verification conditions from code annotations. Bonnier and Heyer point out that full formality is not a requirement for the use of assertions. Matching the method chosen and the level of formality to the project needs appears to be essential in expanding the use of assertions.

3.3. ASSERTIONS AND INTEGRATED MODELING TOOLS

Blake Marietta and Kulkarni at Ames Research Lab have performed collaborative work with us on maintaining software safety. Among their results in linked representations to preserve software safety during maintenance is the extension of a tool to perform feature extraction on C++ code. One of its capabilities is to automatically extract assertions from commented C++ code. It can document the assertions in html pages with URL links to relevant documents such as software requirements specifications, FTAs (Fault Tree Analyses), FMEAs (Failure Modes and Effects Analyses), or PHAs (Preliminary Hazard Analyses). The user-friendly formats lend themselves to the use of assertions for code reviews (or inspections, see above) and for supporting the maintenance or revision of safety-related code. The process of identifying assertions described in Section 2 above can serve as a frontend to this automated software development tool. See http://grail.arc.nasa.gov/rtops/linked_representations.html for the paper describing their work.

4. SUMMARY

In the first phase of this RTOP activity, which culminated in the production of the report, "Assertions: Instrumenting Safety Critical Code for Maintenance", we made a case, based on previous studies and experience, for a more widespread use of assertions. In the second phase of this RTOP activity, reported here, we describe our experiences identifying safety assertions on the DS-1 spacecraft for use either as checks embedded in the code or as predicates for use during test log analysis. The key sources investigated for the assertion-identification process were Software Requirements Documents, Software Fault Tree Analysis, Flight Rules, and Lisp code component models. Besides assertions' well-accepted role as checks in the code, assertions have been used for analyzing test logs, enhancing code inspections, and supporting maintenance by means of integrated modeling tools.

Acknowledgments: Thanks to Martin Feather, John Kelly, Sun Matsumoto, Kirk Reinholtz, Doug Bernard, Guy Man, and Dan Dvorak at JPL; Deepak Kulkarni, Ann Patterson-Hine, Roberta Blake Marietta, and Mike Lowry at ARC; and Jack Callahan at the IV&V Facility for their shared insights, feedback, and technical assistance with the DS-1 material.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial or noncommercial product, process, or service by name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the Jet Propulsion Laboratory, or the California Institute of Technology.

5. REFERENCES

- Bonnier, Staaffan and Tim Heyer, "COMPASS: A Comprehensible Assertion Method, TAPSOFT '07: Theory and Practice of Software Development, Lecture Notes in Computer Science, pp. 803-817. Springer-Verlag, 1997.
- Callahan, John, Steven Easterbrook, and Frank Schneider, "Automated Software Testing Using Model Checking," Workshop on Living with Inconsistency, International Conference on Software Engineering (ICSE97), Boston, MA, May, 1997.
- Chechik, Marsha and John Gannon, "Verification of Consistency Between Concurrent Program Designs and Their Requirements," Proceedings of the 11th Annual Conference on Computer Assurance, June 17-21, 1996, NIST, Gaithersburg, MD, pp. 103-116.
- Chen-Tsai, Ching; Lee, Susan; Shaw, Hui-Yin; Tran, Tuyet-Lan; and Wang, Monica; "Three Cassini Software Fault Tree Analyses Case Studies"; Jet Propulsion Laboratory, January 1997.
- Hamley, John A., NASA Solar Electric Propulsion Technology Applications Readiness (NSTAR) Thruster Element Technical Requirements Document (TRD), ND-310, JPL D-13638, April 21, 1997.
- Hart, Johnson M., "Experience with Logical Code Analysis in Software Reuse and Re-engineering", A Collection of Technical Papers: AIAA Computing in Aerospace and Astronautics, March 28-30, 1995, pp. 549-558.
- Leveson, Nancy G., *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- Leveson, Nancy G. and Peter R. Harvey, "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983, pp. 569-579.
- Leveson, Nancy G., Stephen S. Cha, and Timothy J. Shimeall, "Safety Verification of Ada Programs Using Software Fault Trees", *IEEE Software*, July 1991, pp. 48-59.

Matsumoto, Sun Kang, NASA Solar Electric Propulsion Technology Applications Readiness (NSTAR) Flight System Software/Fault Protection Requirements, ND-302, JPL, draft version, April 30, 1997.

Nayak, Pandu, MIR Livingstone models.

Reinholtz, Kirk and Dan Dvorak, "The TSTAR Autonomy Test Tool, " submitted for publication, July, 1997.

Starbird, Tom, Flight and Mission Rules, DS-1 internal web page.

APPENDIX A. "Assertions: Instrumenting Safety Critical Code for Maintenance"

(<http://eis.jpl.nasa.gov/quality/qadc/software.htm>).

APPENDIX B. "Safety Analysis of Requirements for a Product Family"

(<http://www.cs.iastate.edu/~rlutz/publications/icre98.ps>)