# Interferometer Software Development at JPL: Using Software Engineering to Reduce Integration Headaches

Michael D. Deck[a], Braden E. Hines[b]

[a]Cleanroom Software Engineering, Inc., 7526 Spring Dr. Boulder, CO 80303

[b]Jet Propulsion Laboratory, 4800 Oak Grove Dr. Pasadena, CA 91109
[b]California Institute of Technology

## ABSTRACT

This paper describes some of the software engineering practices that are being used by the Realtime Interferometer Control Systems Testbed (RICST) project at JPL to address integration and integratability issues. New documentation and review techniques based on formal methods permit early identification of potential interface problems. An incremental life cycle improves the manageability of the software development process. A "Cleanroom mindset" reduces the number of defects that have to be removed during integration and test. And team ownership of work products permits the project to grow while providing a variety of opportunities to team members. This paper presents data, including software metrics and analysis, from the first several incremental deliveries developed by the RICST project.

**Keywords:** interferometry, integration, software, software engineering, instrumentation, Cleanroom

## 1 INTRODUCTION

Software is arguably the most critical system in any modern scientific instrument. Software provides the glue that links other systems to the observer. Without attention to software engineering as a critical spacecraft technology, system integration problems can lead to mission delays and even failures.

The Realtime Interferometer Control Systems Testbed (RICST) project is part of the Interferometry Technology Program at JPL. RICST is developing an integrated hardware/software testbed that will be used by several projects including SIM, DS3, GSU CHARA, and the Keck interferometer. A significant part of the RICST effort is to provide embedded software in a multiprocessor architecture for active control of optical devices. Because it is supporting so many different missions, the RICST design objectives give great weight to flexibility and extensibility. At the same time, the search for cost-effective solutions places a premium on reusability and reliability.

An interferometer requires tight coupling between various subsystems. Their optics, electronics, computer systems, and software must be architected and designed as one system. The complexity of the instrument is such that system integration is a major issue; this is where the engineering success of an instrument is made or lost. Opto-mechanical and electronic complexity make it desirable to place as much of the complexity of the instrument as possible inside the computer system, in the software, thus allowing potential software solutions to difficult optical or electronic integration issues. This software includes computer-controlled servos and controllers as well as traditional sequencing, data processing, and decision logic. For example, NASA's ground-based Palomar Testbed Interferometer, PTI[1,2] was developed using a similar technology base, as were earlier systems[3]. PTI is a complex automated system with five major opto-mechanical subsystems, including ~65 sensors, ~18 actuators, high-bandwidth (up to 2000 Hz) computer servo control, nine 680X0 processors, 5 VME cardcages, and ~100K lines of C/C++ code. The PTI development was very successful and resulted in numerous software architecture innovations[4].

RICST's goals include the development of generic interferometer functionality that meets the needs of all its many customers' needs, including full automated observatory and control functionality for a variety of missions and ground instruments. In the process, RICST will allow JPL's experience with ground-based interferometry to be applied to flight and other ground projects, training new staff who can then disseminate this experience to many projects.

A second RICST goal is to perform end-to-end testing and integration of prototype flight hardware with the instrument software, providing feedback to the flight hardware development effort on component performance in an integrated environment.

---

M.D.D. email: deckm@cleansoft.com; www.cleansoft.com/cleansoft; (303) 494-3152.
B.E.H. email: Braden.E.Hines@jpl.nasa.gov; huey.jpl.nasa.gov; (838) 354-2465

The RICST effort also offers an opportunity to prototype processes that can eventually be used on other projects of similar complexity.

This paper describes RICST use of certain software engineering practices to reduce integration problems. The results presented reflect the first few incremental deliveries and span a time frame of approximately 18 months. These results suggest that advanced software engineering practices can improve the effectiveness of flight software projects by helping to mitigate integration risk. A preliminary report of results, detailing the early phases of technology transfer, was published in February 1997[5].

## 2 "CLEANROOM" SOFTWARE ENGINEERING SUMMARY

In the early 1980's, Dr. Harlan Mills of IBM's Federal Systems Division[6,7] observed the quality gap between the hardware and software components of a system and proposed that software developers adopt the successful practices of precision manufacturing. Specifically, he suggested a "Cleanroom" approach to software engineering based on two guiding principles:

- **Design Principle**. Programming teams *can* and *should* strive to produce systems that are nearly error-free upon entry to testing.
- **Testing Principle**. The primary purpose of testing is to *measure* the quality of the developed software product, and not to "test quality in."[8]

Most of the software engineering practices adopted by RICST are guided by the Cleanroom software engineering approach. These practices have been further tailored by the RICST team assisted by Cleanroom experts using experience with numerous Cleanroom projects[9,10,11,12].The practices adopted by the RICST team include the following:

- An incremental life-cycle model delivers end-to-end testable subsets of user functionality over time.
- Hierarchical specification and design using the "box structures" approach is combined with object-orientation using the Unified Modeling Language[13].
- Team reviews informally apply techniques of functional correctness verification result in designs that are nearly error-free before any testing begins.

RICST is depending on the Cleanroom process to deliver software for hardware components that will not be available through much of the development process. A process like Cleanroom, which does not rely on exhaustive testing to deliver quality, offers the best chance for success in such an environment.

## 3 METHODOLOGY PHASING

The RICST team used a phased approach to initial adoption of the Cleanroom practices. It was similar but not identical to that of Hausler, et al. [14]. The phased approach introduces the methodology practices over time. After the initial phases, the team was aware enough of the methodology goals, and was confident enough in their ability to perform in the methodology, that they could actively contribute to subsequent tailoring.

The phased introduction had several steps. First, the project conducted a risk assessment in which it identified key risks to project success. Next, it rank-ordered several competing project goals such as meeting schedule commitments, meeting performance requirements, and designing for maintainability. A phase plan was developed out of the risk assessment and goals prioritization. It defined four phases of technology introduction.

1. Introduce the Cleanroom management techniques of incremental development and team review. This initial phase will precede full-scale work on the project development, and is intended to help refine external commitments.
2. Make the team comfortable with the Cleanroom process through the successful completion of one increment using a subset of Cleanroom practices.
3. Introduce any additional formal techniques needed for specification, verification.
4. Complete the project using specialty techniques developed for this project.

## 4 INCREMENTAL DEVELOPMENT

Cleanroom uses an incremental life-cycle model[9] in which multiple deliveries occur during the course of the project. RICST adds another level to this model, breaking deliveries into multiple internal increments. Each increment provides a subset of

the final system functionality that can be tested in an environment resembling the final system environment. Increments accumulate to form deliveries that are subject to full customer assurance testing. The first four deliveries are shown, together with approximate allocation to development phases, in Figure 1 below.

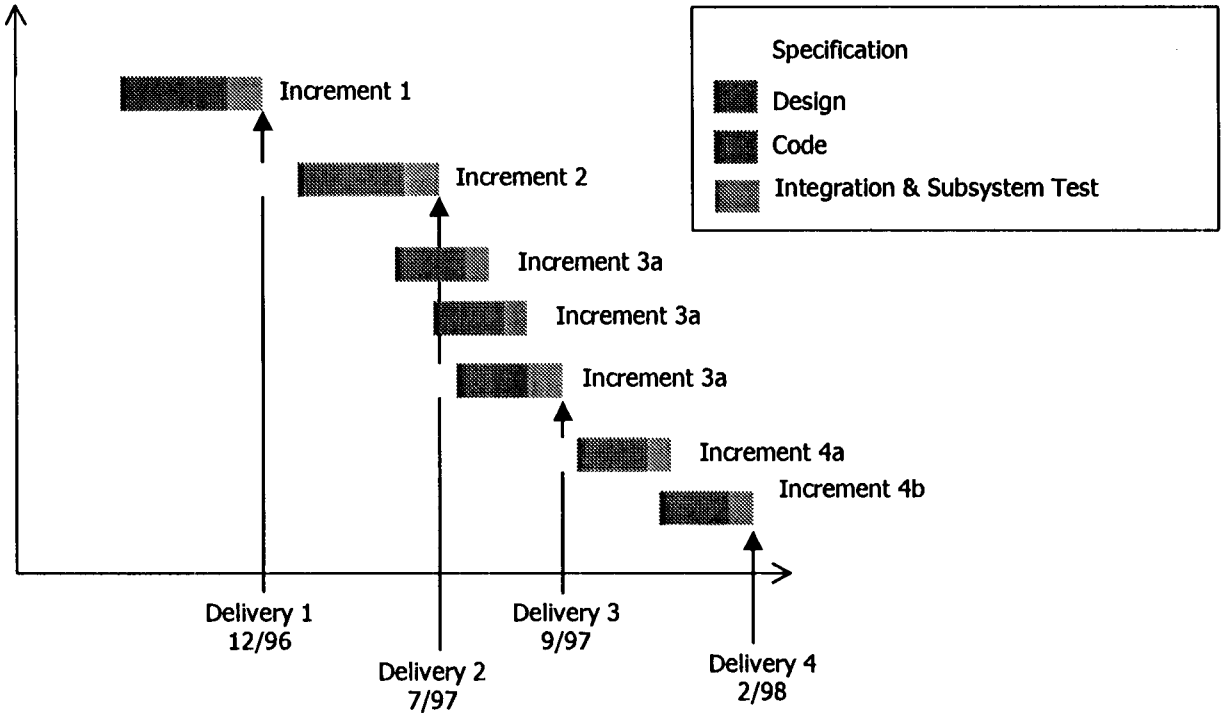


Figure 1. RICST Deliveries and Increments

Traditional top-down incremental development[15] recommends that requirements analysis and specification be complete before incremental design and coding begins. The RICST team, aware of their environment of changing requirements, opted instead to do some requirements and specification work up-front, but to improve and extend the requirements and specification documents significantly within each increment.

### 4.1 Increment Plan

Table 1 summarizes the content of deliveries through February, 1998.

Table 1. Deliveries

| # | Delivered | Lines C++ Code | Focus | Technical Challenges |
|---|---|---|---|---|
| 1 | 12/96 | ~6000 | optical delay line: zero-velocity tracking | realtime periodic task scheduler, multi-CPU coordination; shared memory management; hardware interfaces |
| 2 | 7/97 | ~14000 | full delay line functionality | object-oriented architecture of hardware interfaces; graphical user interface; dither calibration |
| 3 | 9/97 | ~17000 | fringe-tracker hardware integration | hardware-software codevelopment |
| 4 | 2/98 | ~27000 | white-light fringe tracking | realtime coordination between fringe tracker and delay line to form phasing subsystem |

### 4.2 Inter-Increment Activities

Significant effort takes place between increments. First, the results of the previous increment(s) are assessed and process improvements are considered. RICST uses the Defect Prevention Process (DPP)[16] to determine the root cause of key defects and to analyze potential process changes that could have prevented it.

We also consider each new increment to be just like the beginning of a new project: there is a significant body of existing work to build on, an end set of requirements in mind, a target delivery date, and an increment plan to meet it. Re-planning is a frequent activity between increments, as we account for changes to requirements and/or environment that occurred during the previous increment.

The incremental development process meshes well with RICST's customers' needs, as several of the customers are themselves sophisticated interferometer testbeds whose goals are to deliver functionality that increases with time.

### 4.3 Benefits of the Incremental Model

The incremental model has enabled early resolution of several kinds of problems faced by RICST. By getting early experience on the actual hardware, the project team was able to correct several deficiencies in their initial architecture for hardware drivers and for shared memory management. At the same time, the team validated their initial approach to real-time periodic task scheduling. However, the incremental model is not an excuse for the kind of iterative prototyping that produces code which is unsuitable for flight. Instead, each increment focuses on a key project risk area but the code that is developed is done to meet demanding reliability and maintainability standards.

## 5 DEVELOPMENT ACTIVITIES

RICST used most of the specification, design, and review techniques that are considered part of the "typical" Cleanroom canon. The box structures method was used for specification and design, team iterative reviews stressed both design quality and correctness, and a form of functional verification was applied throughout. At the same time, the RICST team developed innovative ways of integrating box structures with object-orientation and the Unified Modeling Language, or UML[13].

### 5.1 Box Structures

Box structures is an approach to system specification developed by Dr. Harlan Mills and his colleagues[17,18]. There is a close link between box structures and object-orientation[19]. In box structures there are three views of any system, component, object, or part. The black box, or "specification," view describes the entity's behavior in terms of events that the system can respond to and the visible behavior that ensues. When the black box behavior depends on the entity's retained data, an abstract model can be used. The black box formalism is supported by functional semantics[20] and model-based abstraction[21].

The state box, or "design," view of an entity describes concrete data implementation structures, but hides processing through the functional abstraction. The "concrete" data structures at this level are themselves instances of lower-level black box data specifications[22]. The state box view closely resembles state machines[21].

A third view, called the clear box or "implementation," describes algorithms or processing. This view defines lower-level process specifications that are themselves functional abstractions with a corresponding black box view. The clear box approach incorporates the theory of structured programming[23].

Each level of this decomposition hierarchy reveals new black boxes. Each of them may have a separate specification, design, and implementation. Because this hierarchy is self-similar at every level down to code, the same techniques are learned once and applied throughout development. However, even though all three views may exist at every level, the team may choose not to document every view individually.

The following sections describe the levels of engineering documentation produced by the RICST developers.

### 5.2 The System Black Box Specification

The first decision in system specification is the choice of a black box boundary: what is "inside" the black box, and what is "outside?" The RICST team chose to place the entire integrated hardware/software system inside the box. Thus the stimuli to the system are user commands and sequences as well as unscheduled events such as laser-beam interruptions (e.g., due to dust) and the reception of science inputs (e.g., starlight). Although the definition of system behavior is extraordinarily difficult with this boundary, it was valuable to take this view as it kept the user of the system in mind, rather than describing operations in terms of the engineer's view.

The team chose to specify the system in terms of an abstract data model that included "hard" elements such as "current optical delay line position" as well as "soft" ones like "science data output buffer." The RICST team found this abstract model to be the most useful both for understanding the system's behavior and for validating that behavior against evolving requirements.

The specification consists primarily of 4-column tables, with one column each for a description of:

- a unique event received by the system, including parameter values if appropriate
- a condition on the state of the abstract model that determines a unique response
- an observable behavior
- a change, if any, to the state of the abstract model

### 5.3 The System Architecture

The system design evolved as a hierarchy of box structures. First, a distinction was made between RICST (which included ground control, telemetry, the space vehicle, and the interferometer) and the RICST Instrument (the interferometer alone). A further distinction was made between the instrument and the real-time computer (RTC). Finally, the RTC was decomposed into functional components like the delay line and the star tracker. Most of the functional components are bound to optomechanical devices but some (like the instrument command and data system) are pure computing artifacts. The system architecture document describes interfaces at the RICST level, the instrument level, and the component level. At present, the team is using inspection and discussion to verify that the architecture meets the black box specification. A future goal is to bring formal methods (see below) to bear on this problem as well.

### 5.4 RTC Component Documents

Each of the RTC components has its own document containing component-specific design requirements, the black box specification of the component, and the design of the component as a collection of C++ classes. These documents contain a combination of graphical models (using the Unified Modeling Language), textual descriptions and motivation, and formal notations.

The need to write detailed specifications for each component, subcomponent, and module forced each team member to carefully examine the behavioral characteristics of each part. As a result, interface designs and requirements assumptions were given much more scrutiny that they would have received in a typical similar project.

### 5.5 Class Specification & Design

By the time that design decomposition reached the level of C++ objects and classes, the specifications moved from documents to comment text embedded within C++ files. The same underlying formal model still applied, but different notations were used.

The team followed a simple rule: every method of every object was required to have a black box specification defined by its author. Every user of a method could then copy that specification inline as the intended behavior of the method invocation. That user could then apply function abstraction techniques (after Linger, et al.[23]) to state the net effect of a sequence of function calls.

The team used a combination of styles in arriving at black box specifications. One style was top-down: the author wrote an intended behavior first, then the design or implementation of that behavior, and then compared the two for correctness. A second approach was to write the design or implementation first, using a mental specification, then derive the net effect and use that as specification.

There were several interesting discoveries that arose out of this design and specification process. The first was the ability to observe and document a huge number of potential failure modes that were not immediately obvious on initial investigation. For example, when a function receives a pointer as an argument, one potential source of failure is that the pointer is invalid (either null or "wild"). By writing formal specifications for each function, the user of that function is presented with a black box behavior such as shown in Figure 2 which reads, "if pDataObj is invalid, the results of this function are undefined; otherwise set the object pointed to by pDataObj to new_value." It is then up to the user to decide whether this is acceptable or not.

```
[ pDataObj is invalid → UNDEFINED
| true → *pDataObj := new_value ]
```

*Figure 2. Black Box Behavior Example*

All together, describing all the undefined and "error" cases accounted for perhaps as much as 70% of the method implementation effort. While this may seem extreme – only spending 30% of the time on "nominal" cases – we find that programmers are naturally pretty good at getting the nominal case right. It's the abnormal or unusual cases that lead to catastrophic system failures.

### 5.6 Applications to Hardware

During the verification process, we discovered that a similar sort of analysis and documentation process can be applied to hardware/software interfaces. For example, in order for driver dt1401 to perform in its nominal case, the DT1401 card must be connected to a source of timing pulses. We could document this assumption as shown in Figure 3.

```
[ DT1401 input "B" not connected to
    output of TimingSource ||
  Timing Source not running → UNDEFINED
| true → ... nominal behavior ... ]
```

Figure 3. Hardware Interface Black Box

Further, the timing source must have the right synchronization signals. The synchronization signals must be triggered by the GPS receiver, which in turn must be on and operating within specified parameters. All of these connections can be documented using the same functional approach. The result of this chain of analysis leads to a statement, documented in the system specification under the heading "Hardware Setup" as follows:

- DT1401 input "B" connected to output of TimingSource
- TimingSource running
- Synchronizer output connected to input of TimingSource
- Synchronizer running
- Synchronizer trigger input connected to GPS receiver
- GPS receiver "on" and one of trigger outputs connected to trigger input of Synchronizer

In this way the hardware setup can be "verified" in a fashion similar to the way functional program correctness is verified.

Using functional techniques, a set of precise hardware interface specifications is being developed that will limit risks resulting from the simultaneous engineering of hardware and software.

### 5.7 Tools and "CleanSpec"

One of the most useful outcomes of having a formal specification for every method or procedure is that a user of the procedure can quickly see what it does without worrying about how it does it. In fact, if one makes a copy of the procedure's specification on top of every use of that procedure in the code, it is possible to summarize large amounts of code in a few pages.

```
void procedure_a (int x) {

  /* copy of specification of procedure_b */
  procedure_b (x,y);

  /* copy of specification of procedure_c */
  procedure_c (y,z);

}
```

Figure 4. Use of Inline Specification

Unfortunately, the team very quickly discovered that copying and maintaining these inline specifications was extremely time-consuming and burdensome. So a tool was developed that would automatically expand and update specifications as needed. The developer of a procedure now writes the specification, using special keywords and tags, within a comment in the C++ header file. The user of the procedure places a special comment on top of any call to the procedure, describing parameters to be substituted and selecting formatting options. The tool expands the specification and produces a listing that is used in the iterative review process.

### 5.8 Team Iterative Review

Each artifact of development is subjected to an iterative review process by the team. The system black box specification and system architecture are reviewed by a "lead" team consisting of project architects and senior engineers. Each component document is reviewed by the team of 3-5 engineers that owns the component, plus engineers representing the components

that use it. Component implementations are reviewed using verification-based inspection[24] by 2-3 people other than the developer*. Code was compiled by the developer before the final implementation review.

# 6 TESTING & RESULTS

## 6.1 Testing Practices

Cleanroom emphasizes the use of statistical (sometimes called "stochastic") black box testing based on operational profiles[25]. The emphasis of this testing is the measurement of reliability, rather than the coverage of program structures. This is essentially Monte-Carlo testing of the software, with the added twist that we use our knowledge of how the system will be used to make better choices of test cases by selecting input sets (sequences of stimuli) that are statistically representative of actual usage. It offers a clean separation between the development and testing activities and reduces the number of common-mode failures that might arise.

To date, the RICST team has chosen (primarily due to resource limitations) to use traditional forms of testing including functional coverage testing, rather than statistical testing. Private, ad hoc "unit" testing by developers is specifically prohibited on the RICST project. Although it cannot be physically prevented, it is difficult in most cases because of the realtime, interdependent nature of the system. The results to date show that the review process delivers better and more reliable code, more cost-effectively. More importantly, the prohibition against developer "unit" testing enforces a strict adherence to team review techniques.

## 6.2 Results

The data from RICST testing, some of which are shown in Table 2, are somewhat difficult to analyze. When taken in aggregate, the number of defects found in testing is higher than would be expected from the process used. There are several reasons why this may be so. Because of the way we defined the system black box to contain hardware as well as software, many hardware and configuration errors appear in these numbers. We also must account for the high rate of staff growth and the complexity of the project.

The total defect rate also includes "defects" associated with the team's lack of familiarity with the support tools including compilers, operating systems, and makefiles. It also includes several defects in the operating environment itself. For example, the realtime operating system takes 600 µsec to switch contexts when certain communications tasks are running. Diagnosing such problems is invariably costly, though repairs were relatively simple.

Finally, we have observed a strong correlation between the granting of a waiver from rigorous review and the detection of significant defect counts.

Table 2. Embedded System Defects

| # | Delivered | Lines C++ Code in Embedded System | New Defects Discovered | Cumulative Defects | Cumulative Defects per 1000 LOC |
|---|---|---|---|---|---|
| 1 | 12/96 | ~6000 | 34 | 34 | 6 |
| 2 | 7/97 | ~8000 | 69 | 103 | 13 |
| 3 | 9/97 | ~13000 | 46 | 149 | 11 |
| 4 | 2/98 | ~16000 | 73 | 222 | 14 |

Only defect rates for the embedded system code are reported, as those are the only defects that RICST tracks and monitors. The GUI code is considered to have lower reliability requirements and is not subject to defect tracking at present.

Defects are not tracked to the increment of code in which they may have appeared, so it is impossible to say at this point whether a defect discovered during testing of delivery #2 appeared in new code or was undetected in increment 1 testing.

The objective metrics from increment 1 are only part of the story. We also believe that

* Although the team's agreed-upon process mandates certain reviews, it is possible for a waiver to be granted by management.

- The code is smaller, in terms of the number of statements, than would have been expected from non-Cleanroom work, by 20% or more. This is based on comparisons with very similar functionality contained in the Palomar Testbed Interferometer. We hope to do head-to-head comparisons with a similar JPL project in the next increment.
- The team reviews helped all team members understand the architecture and interfaces better than they would have in a typical JPL project. The team also has more flexibility to meet other external demands on members' time because of that distributed knowledge.
- Team reviews have also proved invaluable as a means of disseminating domain-specific knowledge, which is concentrated in a few key individuals, to the rest of the team. This helped RICST meet its goal of developing additional interferometer software/integration experts.
- The likelihood of having few defects in any given class appears to be strongly correlated to the emphasis placed on that module's specification and review.
- The design is simpler, more robust, more modular, and will be easier to adapt to changing requirements, than it normally would be. Frequently, in the process of verification, the additional conceptualization and reasoning performed by the developer or the team revealed simplifications and improvements to what had previously been conceived.
- When the review process was followed, no defects were introduced through the correction of a fault after testing began.
- Hardware/software integration issues are better understood than ever before, and the cost of diagnosing such issues during testing is expected to be significantly less than for similar projects.
- The team is enthusiastic about the process.

We are in the process of improving the process documentation, tightening up process controls, and implementing new metrics in an effort to discover how defect densities may be further reduced and what factors influence defect density.

## 7 LESSONS LEARNED

One of the lessons learned from the first four RICST increments is the importance of a standardized, documented process that clearly describes the specification, design, and verification techniques. The distribution of defects is very suggestive of uneven application: those modules that were subjected to the closest review scrutiny by the most senior team members had far fewer defects.

Experienced and savvy developers must "buy-in" to the idea of a Cleanroom process. While senior developers may be biased against a process that eschews unit testing, their domain experience is crucial. In fact, the desire to do good box structures design often drives the architecture, and a seasoned software architect is needed to develop a framework that will both meet project needs and is amenable to Cleanroom verification.

Careful management control over team activities is important while methodology learning is taking place. Attributes of this control include the frequency and structure of reviews, the balance between perfection and adequacy, and judgments as to functionality that could be deferred to other increments. After a first successful increment, the team members can be relied on to make many of the right decisions, but the first increment is critical. Further, we observed that, as people came closer to what felt like "coding," they showed some inclination to lapse into the bad old habits of ad-hoc testing and debugging rather than formal specification and team reviews. Making this difficult period as short as possible is one important factor in increment scheduling. It is no longer possible to physically prevent ad-hoc unit testing. However, every possible management technique, including peer pressure, must be used to discourage it in favor of the more cost-effective Cleanroom techniques.

In any project where safety is critical or even where failures are very costly, it is probably necessary to conduct some amount of supplemental testing in addition to statistical testing. Thorough statistical testing is very effective at measuring reliability, but it can miss defects that are unlikely to occur. The combination of statistical and later formal coverage testing addresses these concerns.

The RICST project demonstrates the ability of Cleanroom software engineering to meet the challenges of a hard real-time multiprocessing environment where reliability demands are high. Discussion with members of other JPL project teams suggests that RICST quality and productivity are well within norms for similar projects, and may even be somewhat better.

## 8 ACKNOWLEDGEMENTS

The project described in this paper is supported in part by government contract no. NAS 7-1260.

## 9 REFERENCES

1. M. Shao and M.M. Colavita, "Potential of long-baseline infrared interferometry for narrow-angle astrometry," *Astronomy and Astrophysics* **262**, 353-358, 1992.

2. M.M. Colavita, M. Shao, B.E. Hines, J.K. Wallace, Y. Gursel, C.A. Beichman, X.P. Pan, T. Nakajima, and S.R. Kulkarni, "ASEPS-0 Testbed Interferometer," *Proc. SPIE: Amplitude and Intensity Spatial Interferometry II* **2200**, 89-97, 1994.

3. M. Shao, M.M. Colavita, B.E. Hines, D.H. Staelin, D.J. Hutter, K.J. Johnston, D. Mozurkewich, R.S. Simon, J.L. Hershey, J.A. Hughes, and G.H. Kaplan, "The Mark III Stellar Interferometer," *Astronomy andAstrophysics*, **193**, pp. 357-371, 1988.

4. B. Hines, "ASEPS-0 Testbed Interferometer Control System," *Proc. SPIE: Amplitude and Intensity Spatial Interferometry II* **2200**, pp. 98-109, 1994.

5. M. Deck and B. E. Hines, "Cleanroom software engineering for flight systems: a preliminary report," *IEEE Aerospace Conference*, February, 1997.

6. M. Dyer and H.D. Mills, "The Cleanroom Approach to Reliable Software Development," *Proc. Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting: Production of Reliable Flight-Crucial Software*, November, 1981.

7. H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software*, pp. 19-25, September, 1987.

8. M.D. Deck, "Cleanroom Software Engineering: Quality Improvement and Cost Reduction," *Proc. Pacific Northwest Software Quality Conference*, October, 1994.

9. R.C. Linger, "Cleanroom Process Model," *IEEE Software*, pp. 50-58, March, 1994.

10. M.D. Deck, "Cleanroom Software Engineering: Quality Improvement and Cost Reduction," *Proc. Pacific Northwest Software Quality Conference*, October, 1994.

11. S. Wayne Sherer, Ara Kouchakdjian, and Paul G. Arnold, "Experience Using Cleanroom Software Engineering," *IEEE Software*, pp. 69-76, May, 1996.

12. V.I. Basili, M. Zelkowitz, F. McGarry, G. Page, S. Waligora, and R. Pajerski, "SEL's Software Process-Improvement Program," *IEEE Software*, pp. 83-87, November, 1995.

13. H-E. Eriksson and M. Penker, *UML Toolkit*, Wiley, 1998.

14. P.A. Hausler, R.C. Linger, and C.J. Trammell, "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems Journal*, **33(1)**, pp. 89-109, January, 1994.

15. R.C. Linger, and A.R. Hevner, "The Incremental Development Process in Cleanroom Software Engineering," Proc. *Workshop on Information Technologies and Systems* (WITS-93), December, 1993.

16. R.G. Mays, C.L. Jones, and G.J. Holloway, "Experiences with Defect Prevention," *IBM Systems Journal*, **29(1)**, January, 1990.

17. H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, 1985.

18. M.D. Deck, Mark G. Pleszkoch, R.C. Linger, and H.D. Mills, "Extended Semantics for Box Structures," *Proc. 25th Hawaii International Conference on System Sciences*, January, 1992.

19. M.D. Deck, "Cleanroom and Object-Oriented Software Engineering: A Unique Synergy," *Software Technology Conference*, April, 1996.

20. H.D. Mills, "The New Math of Computer Programming," *Communications of the ACM*, **18(1)**, pp. 43-48, January, 1975.

21. K. S. Shankar, "Data Structures, Types, and Abstractions," *IEEE Computer*, pp. 67-77, April, 1980.

22. M.D. Deck, "Data Abstraction in the Box Structures Approach," *Proc. 3rd International Conference on Cleanroom Software Engineering Practices*, October, 1996.

23. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Reading, MA: Addison-Wesley, 1979.

24. M. Dyer, *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.

25. J.D. Musa, "Operational Profiles in Software-Reliability Engineering," *IEEE Software*, pp. 14-32, March, 1993.