Formalizing Space Shuttle Software Requirements

Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP'96), San Diego, CA, January 1996, pp. 40–48.

Judith Crow
Computer Science Laboratory
SRI International
Menlo Park, CA 94025
crow@csl.sri.com

Ben L. Di Vito
VíGYAN, Inc.
30 Research Drive
Hampton, VA 23666-1325
b.l.divito@larc.nasa.gov

Abstract

This paper describes two case studies in which requirements for new flight-software subsystems on NASA's Space Shuttle were analyzed, one using standard formal specification techniques, the other using state exploration. These applications serve to illustrate three main theses: (1) formal methods can complement conventional requirements analysis processes effectively, (2) formal methods confer benefits regardless of how extensively they are adopted and applied, and (3) formal methods are most effective when they are judiciously tailored to the application.

1 Introduction

Although Space Shuttle flight software is generally considered exemplary among NASA software development projects, requirements analysis and quality assurance in early lifecycle phases still use products and tools dating from the late 1970s and early 1980s. As a result, these analysis and assurance activities remain largely manual exercises lacking well-defined methods or techniques. At the same time, Shuttle flight software is life-critical and increasingly complex. Software upgrades to accommodate new missions such as the recent MIR docking, new capabilities such as Global Positioning System (GPS) navigation, and improved algorithms such as the newly automated three-engine-out contingency abort maneuvers (3E/O) and the recent optimization of Reaction Control System Jet Selection (JS) are continually introduced. Such upgrades underscore the need recognized in the NASA community and in a recent assessment of Shuttle flight software development, for "state-of-the-art technology" and "leading-edge methodologies" to meet the demands of software development for increasingly large and complex systems [12, p. 91].

Over the last three years, NASA Langley Research Center (LaRC) and its subcontractors, VíGYAN and SRI, have investigated the use of formal methods (FM) in aerospace applications, as part of a three-center demonstration project involving LaRC, the Jet Propulsion Laboratory (JPL), and the Johnson Space Center (JSC). Loral Space Information Systems (formerly IBM, Houston) participated as a subcontractor for JSC.

This paper focuses exclusively on LaRC project activity[1], which was performed in the context of a broader program of formal methods work [1]. The effort consisted of formalizing selected Shuttle software (sub)system modifications and analyzing key system properties using either SRI's PVS specification language and interactive proof-checker [14], or Stanford's Murϕ (pronounced "Murphy") finite-state verification system [4, 8]. The LaRC work had three main goals. First, to explore and document the feasibility of formalizing critical Shuttle software requirements representing a spectrum of maturity levels. Second, to develop reusable formal methods strategies for representative classes of Shuttle software. Third, to identify and assess key factors in the transfer of this technology to JSC and Loral.

The Shuttle subsystems selected for the LaRC projects directly reflect the first two goals. JS, 3E/O, and GPS are all part of critical on-board flight software. The JS requirements are mature and stable, the 3E/O requirements are somewhat newer, having only recently stabilized after a series of iterations, and the GPS requirements are quite new and still in flux. JS and GPS belong to a class of Shuttle software that is readily formalized using a functional model of computation, basically a control function augmented with state variables, and effectively verified using standard theorem-proving techniques. By contrast, 3E/O represents a class of mode-sequencing software that can be quite naturally modeled with finite-state systems and effectively verified using state exploration. We have developed general strategies for specifying and verifying these two classes of Shuttle software and have demonstrated their utility in the JS [11, Appendix B], 3E/O [2], and GPS [3] projects. The technical approach for JS is essentially the same as that used for GPS, although the strategy was refined and

[1] Descriptions of some of the JPL, JSC, and Loral activities undertaken for this project can be found in [5, 6, 9]. The overall project is not large, roughly the equivalent of one full-time position at each of the three NASA centers per year, including Loral, VíGYAN, and SRI time.

documented as part of the GPS study. Accordingly, we present only GPS to illustrate the approach. Although PVS and Murϕ were used for this work, the strategies are applicable to other systems with equivalent power and functionality.

The key technical results of the project include a clear demonstration of the utility of formal methods as a complement to the conventional Shuttle requirements analysis process. The JS, 3E/O, and GPS projects each uncovered anomalies ranging from minor to substantive, most of which were undetected by existing requirements analysis processes. A further result is a comparative demonstration of the different roles formal methods can play, underscoring the following two precepts: formal methods confer benefits regardless of how extensively they are applied, and formal methods are most effective when they are judiciously tailored to the application.

2 Formalization Strategies

In this section we discuss two general strategies and illustrate their application to the GPS and 3E/O requirements, beginning with a sketch of Shuttle software development and a brief description of PVS and Murϕ.

2.1 Shuttle Software Background

NASA's prime contractor for the Space Shuttle is the Space Systems Division of Rockwell International. Loral Space Information Systems (formerly IBM Federal Systems, Houston) is their software subcontractor. Draper Laboratory also serves Rockwell, providing requirements expertise in guidance, navigation and control.

Much of the Shuttle software is organized into major units called *principal functions*, each of which may be subdivided into *subfunctions*. Since the late 1970s, software requirements have been written using venerable conventions known as Functional Subsystem Software Requirements (FSSRs) — low-level software requirements specifications written in English prose, presented with strong implementation biases, and accompanied by pseudo-code, and diagrams and flowcharts in arcane notations. Interfaces between software units are specified in input-output tables. Inputs can be variables or one of three types of constant data: *I-loads* (fixed for the current mission), *K-loads* (fixed for a series of missions), and physical constants (never changed).

Shuttle software modifications are packaged as Change Requests (CRs), that are typically modest in scope, localized in function, and intended to satisfy specific needs for upcoming missions. Roughly once a year, software releases called Operational Increments (OIs) are issued incorporating one or more CRs. Shuttle CRs are written as modifications, replacements, or additions to existing FSSRs.[2] Loral Requirements Analysts (RAs) conduct thorough reviews of new CRs, analyzing them with respect to correctness, implementability, and testability before turning them over to the development team. Their objective is to identify and correct problems in the requirements analysis phase, avoiding far more costly fixes later in the lifecycle.

2.2 Summary of PVS and Murϕ

PVS (Prototype Verification System) is an environment for specification and verification developed at SRI International's Computer Science Laboratory [14]. The distinguishing characteristic of PVS is a highly expressive specification language coupled with a very effective interactive theorem prover that uses decision procedures to automate most of the low-level proof steps. Murϕ is a fully automatic state exploration tool developed by David Dill and his students at Stanford University that uses efficient encodings, including symmetry-based techniques, and effective hash-table strategies to do "reachability analysis," i.e., to check that all reachable states satisfy specified properties [4, 8]. Murϕ can explore millions of states in a matter of minutes.

2.3 Functional Specification

As one of the larger ongoing Shuttle CRs, the GPS CR provides a significant upgrade to the Shuttle's navigation capability. A portion of this CR has been formalized using the functional specification technique. After a brief overview of GPS, we develop the abstract state machine model used and describe the results of this application.

2.3.1 Overview of GPS

The GPS retrofit was planned in anticipation of the DoD's phaseout of the TACAN navigation system. Shuttle vehicles will be outfitted with GPS receivers and additional navigation software will be incorporated into the Shuttle to process the position and velocity vectors generated by these receivers. Currently, the Shuttle navigation system can accept state vector updates derived from ground-based radar observations. The Shuttle GPS software CR will adapt this feature, providing the capability to update the Shuttle navigation filter states with selected GPS state vector estimates similar to the way state vector updates are currently accepted from the ground.

The GPS trial formalization focused on a few key areas because the CR is very large and complex. After preliminary study of the CR and discussions with the GPS RAs, we decided to concentrate on two new principal functions, emphasizing their interfaces to existing navigation software and excluding crew display functions. The two principal functions, known as GPS Receiver State Processing and GPS Reference State Processing, select and

[2] The JS requirements are roughly 70 pages of tables and low-level diagrams, 3E/O is roughly 70 pages of prose, pseudo-code, flowcharts, and tables, and the GPS subset undertaken here is approximately 110 pages of prose, pseudo-code, and tables.

modify GPS state vectors for consumption by the existing entry navigation software.

2.3.2 Technical Approach

We have devised a strategy to model Shuttle principal functions based on the use of a conventional abstract state machine model. Each principal function is modeled as a state machine that takes inputs and local state values, and produces outputs and new state values. This method provides a simple computational model similar to popular state-based methods such as the A-7 model [7, 17].

One transition of the state machine model corresponds to one scheduled execution of the principal function, e.g., one cycle at rate 6.25 Hz or other applicable rate. All of the inputs to the principal function are bundled together and a similar bundling of the outputs is arranged. The state variable holds values that are (usually) not delivered to other units, but instead are held for use on the next cycle.

The state machine transition function is a mathematically well-defined function that takes a vector of input values and a vector of previous-state values, and maps them into a vector of outputs and a vector of next-state values.

$$M : I \times S \to [O \times S]$$

This function M is expressed in PVS and forms the central part of the formal specification. We construct a tuple composed of the output and state values so only a single top-level function is needed in the formalization. Some values may appear in both the output list and the next-state vector.

While the function M captures the functionality of the software subsystem in question, the state machine framework can also serve to formalize abstract properties about the behavior of the subsystem. The common approach of writing assertions about *traces* or sequences of input and output vectors is easily accommodated. For example, we can introduce sequences $I(n) = < i_1, \ldots, i_n >$ and $O(n) = < o_1, \ldots, o_n >$ to denote the flow of inputs and outputs that would have occurred if the state machine were run for n transitions. A property about the behavior of M can be expressed as a relation P between $I(n)$ and $O(n)$ and formally established, i.e., we prove that the property P does indeed follow from the formal specification M using the PVS proof-checker.

Figure 1 shows the abstract structure of a Shuttle principal function rendered in PVS notation. Key features of this structure are the two kinds of variable data (input values, previous-state values) and three kinds of constant data (I-loads, K-loads, constants) used as arguments, and the result returned containing both output values (produced for other principal functions) and next-state values (used by this principal function on the next cycle).

The PVS definition assumes all input and state values have been collected into the structures `pf_inputs` and `pf_state`. Additionally, all I-load, K-load, and constant inputs used by the principal function are collected into similar structures. The `pf_result` type is a record that contains an output component and a next-state component. Each of these objects is, in turn, a structure containing (possibly many) subcomponents.

```
pf_result: TYPE = [# output: pf_outputs,
                     state:  pf_state #]

principal_function
         (pf_inputs, pf_state,
          pf_I_loads, pf_K_loads,
          pf_constants) : pf_result =

   (# output := <output expression>,
      state  := <next-state expression>
    #)
```

Figure 1: PVS model of a Shuttle *principal function*.

The output and next-state expressions in the general form above describe the effects of invoking the subfunctions belonging to the principal function. In practice, this can be very complicated so a stylized method of organizing this information has been devised, based on the use of a LET expression to introduce variable names corresponding to the intermediate inputs and outputs exchanged among subfunctions.

In the CR, GPS Receiver State Processing is decomposed into six subfunctions, and GPS Reference State Processing into four. In several cases the subfunction requirements were sufficiently complex that it became necessary to introduce intermediate PVS functions to decompose the formalization further. While this is a reasonable strategy, it does cause some loss of traceability to the original requirements. Clarity and readability were judged more important, however, and such decompositions were introduced as needed. A consistent decomposition scheme helped make the use of such intermediates as transparent as possible.

The two GPS principal functions were formalized in about 3300 lines of PVS specifications (including comments and blank lines), packaged as eleven PVS theories. Writing the original version and performing three revisions to track requirements changes took an estimated two staff months of effort over a four-month period. This period followed an earlier round of familiarization with the CR.

2.3.3 GPS Results

Experience with the GPS effort showed that the outlook for formal methods in requirements analysis is promising, but it is still too early to declare victory. The GPS requirements were still converging at the time of this study, and a start during a later revision cycle would have made the FM results more meaningful. Stable requirements were not expected until late in 1995.

PVS can and has been used effectively to formalize this application; the custom specification approach should be easy to duplicate for other areas, yielding good prospects for continuation of these efforts. Although the specification activity was assisted by tools, manual specification is also feasible here, albeit with reduced benefits.

Some Shuttle RAs are optimistic about the potential impact of FM. Their feedback indicated our approach was helpful in detecting three classes of errors normally tracked by the Shuttle program:

1. Type 4 — requirements do not meet CR author's intent.

2. Type 6 — requirements not technically clear, understandable and maintainable.

3. Type 9 — interfaces inconsistent.

An example of Type 4 errors encountered in the CR is omission due to conditionally updating variables. Suppose, for example, one branch of a conditional assigns several variables, leaving them unassigned on the other branch. The requirements author intends for the values to be "don't cares" in the other branch, but occasionally this is faulty because some variables such as flags need to be assigned in both cases. Similar problems can occur with overlapping conditions which cause ambiguity with respect to correct variable assignments.

Examples of Type 9 errors, the most prevalent type encountered, include numerous, minor cases of incomplete and inconsistent interfaces. Missing inputs and outputs from tables, mismatches across tables, inappropriate types, and incorrect names are all typical errors seen in the subfunction and principal function interfaces. Most are problems that could be avoided through greater use of automation in the requirements capture process.

It is worth noting that most errors detected in the CR during the formalization were not exposed by typechecking or other automated analysis activity, but were found during the act of writing the specifications or during the review and preparation leading up to the writing step. Attempting to write a PVS construct and looking around for the declared objects it needs effectively flushed out many problems. On the other hand, there were also cases where additional errors were found during the typechecking phase. For example, interface problems (Type 9) usually surfaced during the act of specification writing, while more serious problems stemming from logic errors or inappropriate operations (Types 4 and 6) often persisted to the typechecking stage, where type mismatches would reveal an error such as a missing subscript.

A preliminary comparison with the conventional analysis process is revealing. By considering errors detected in Reference State Processing during the first pass of the FM-assisted analysis versus the subset of those errors also found by the current process, a bit of anecdotal evidence emerges for what can be missed by a manual review process:

Issue Severity	With FM	Existing
High Major	2	0
Low Major	5	1
High Minor	17	3
Low Minor	6	0
Totals	30	4

While most of these problems were of the minor Type 9 variety, a few were more serious. Since errors escaping detection during this phase will be more costly to correct during development or system test, our results suggest that the added precision of formalization used early in the lifecycle can yield tangible benefits.

Although many of these issues could have been found with lighter-weight techniques, the use of formal specifications can detect errors *and* leave open the option of deductive analysis later on. When we reach the point of modeling higher level properties and carrying out proofs, we expect to see fewer errors. Lightweight forms of analysis applied early detect more problems and detect them quickly, but the errors tend to be superficial. As more powerful analysis methods are introduced, we expect to find fewer, but more subtle problems.

The next step in the application of FM to GPS, which had been deferred as of this writing, is to identify and formalize important behavioral properties of the processing of GPS position and velocity vectors. Proving these properties hold will offer a powerful means of further shaking out the requirements before (and even after) passing them on to development.

2.4 Mode-Sequencing Specification

The primary purpose of the 3E/O CR is to automate and upgrade the 3E/O contingency guidance function. Following a brief overview of 3E/O, we describe the finite-state machine model used to specify the mode-sequencing component of 3E/O and summarize the results of the analysis.

2.4.1 Overview of 3E/O

3E/O is responsible for monitoring ascent parameters and, if three Shuttle main engines fail sequentially or simultaneously, calculating and commanding the appropriate abort maneuver if an abort is necessary. In certain situations, 3E/O is also responsible for automatic contingency maneuvers resulting from the failure of two Shuttle main engines (2E/O). 3E/O is executed repeatedly at specified intervals that range from 1.92 seconds to 0.16 seconds; each execution is part of a *guidance cycle* that remains active during powered flight until either a contingency abort is required or progress along the powered flight trajectory is sufficient to preclude an abort even if three main engines fail.

3E/O consists of two main functions: a region-select function that selects a contingency-maneuver mode based on the values of ascent parameters, and a contingency guidance function that is used strictly for display if the ascent is normal, but is responsible for calculating and commanding initial abort maneuvers determined by the selected region if a contingency arises. The maneuvers calculated and commanded by the two 3E/O functions may differ from one guidance cycle to the next in response to changes in the external environment or in the Shuttle's internal state.

2.4.2 Technical Approach

3E/O is typical of most fault-handling logic in the following respects: it consists largely of mode switching and exception handling, it exhibits virtually no algorithmic complexity, and its input and state spaces lack a regular and easily characterizable structure. Due largely to these three characteristics, conventional specification and verification strategies that use type checking and theorem proving to establish correctness of functional requirements are not effective validation methods for fault-handling applications. The simplest way to validate fault-handling systems is to enumerate the entire input and state spaces by brute force. While this is rarely practical—the state space of most applications of interest is far too great to permit exhaustive enumeration—it is often possible to "downscale" the state space of an application to a reasonably small, finite size, while retaining essential behaviors of the original system. The downscaled or aggressively abstracted system can then be analyzed effectively using state exploration, as we have done with 3E/O. In broad terms, the technical approach used for 3E/O has been to develop an abstract model that would enable us to encode the basic sequencing constraints as transitions in a finite-state machine and to specify key properties of the 3E/O sequencing algorithm as invariants that must hold in all reachable states. We elaborate key elements of this approach below.

The challenge in modeling Shuttle flight software typically derives from the number of input variables[3] and their inherent complexity. The strategy we developed defines a state transition system operating within a two-level context: a global environment consisting of variables representing sampled sensor values for external physical parameters (e.g., current dynamic pressure), and a local environment consisting of variables representing the Shuttle's internal status.[4] To provide a reasonably tractable and accurate model of the input space, we make the following simplifying assumptions.

1. *The largely continuous values representing the Shuttle's physical environment can be modeled using either qualitative ranges or booleans.* For example, to verify a particular sequence that (partially) determines assignment of abort maneuver regions, it is sufficient to check whether the current altitude and altitude rate predict an apogee altitude greater or less than the calculated altitude-velocity curve. Since the exact values or physical laws involved are irrelevant for verifying the crucial sequencing properties, a simple boolean-valued check suffices. Similarly, to verify another sequence that also (partially) determines assignment of abort maneuver regions, it is necessary to check that the inertial velocity falls within certain I-loaded thresholds. Since the exact inertial velocity is not a factor (in this case or any other), there is no need to represent a continuous range of values; the sequencing property can be established with respect to a qualitative range that reflects the set of possibilities defined by the fixed thresholds.

2. *Constraints on the simultaneous values assumed by variables representing physical parameters can be ignored.* For example, we make no attempt to capture the relation between velocity and altitude. Although this is clearly naive, it is also overly general; while we may consider too many cases, we do not overlook any.[5]

3. *The implicit notion of time inherent in an ordered sequence of events is sufficient.* No further or more explicit representation of time is necessary for analyzing 3E/O sequencing properties.

We further reduce the large number of inputs by exploiting the fact that a boolean-valued operation on two inputs is equivalent, as a sequencing constraint, to a simple boolean variable. For example, in 3E/O, the boolean expression used to determine if the Shuttle has sufficient range for a particular type of abort maneuver checks two conditions: is the down-range horizontal earth-relative velocity strictly less than 0, and is the difference between the predicted and actual range capability strictly greater than an I-loaded minimum acceptable range difference, i.e., `v_horiz_dnrng < 0 AND delta_r > del_r_usp`. The value of the operation in each conjunct is either true or false, and indistinguishable from a single boolean-valued variable; as a sequencing constraint, this conjunction is equivalent to the expression: `v_horiz_dnrng_LT_0 AND delta_r_GTR_del_r_usp`, where each conjunct is reduced to a simple boolean variable. By universally quantifying over these variables, we effectively show that for all possible values of the two (original) expressions, certain properties hold. We use

[3] The 3E/O requirements document contains six full, double-spaced pages of inputs, most of which represent I-loaded thresholds used to calculate the order and timing of the maneuver sequences.

[4] This strategy is reminiscent of the standard A-7 approach [7, 17], but differs in the way the environment is modeled.

[5] In the context of finite-state verification, a technique which prides itself on being able to handle very large (but finite) state spaces, it is far better to consider too many possibilities, than too few.

```
Ruleset <external physical parameter 1>: TYPE
  Do
⋮

Ruleset <external physical parameter n>: TYPE
  Do

Rule "<rulename>"

⋮

Rule "<rulename>"

Endruleset;

⋮

Endruleset;
```

Figure 2: Murϕ Abstract Syntax Used to Model Shuttle's Physical Environment.

this strategy for all inputs that represent the Shuttle's physical environment.

Figure 2 shows a template for the Murϕ rules used to generate input values for external physical parameters. The *Ruleset* construct is syntactic sugar that generates a copy of the rules within its scope for every value of the bound variable. Inputs that represent the Shuttle's internal state, e.g., the flag that indicates whether contingency 3E/O has been activated or the variable that encodes whether main engine cutoff has occurred, are modeled as independent inputs using simple nondeterministic rules that generate all possible (combinations of) values in the input space.

Finally, we model basic mode sequencing properties as state transition constraints and then show that if these constraints are satisfied, key properties of the algorithms also hold. The key properties are specified as invariants, i.e., expressions that must be true in all reachable states. For example, we use an invariant to check that all outputs persisting from the previous cycle remain correct with respect to conditions in the current cycle.

Employing the general strategy outlined here, the two main 3E/O functions were specified in approximately 1200 lines of Murϕ code including comments, using approximately a dozen invariants.

2.4.3 3E/O Results

Our experience has been that the process of formalizing and analyzing requirements invariably exposes undocumented assumptions, inconsistent and imprecise terminology, redundant calculations, missing initialization, interface anomalies, and logical errors. Of the issues tabulated below and reported to the 3E/O RA, roughly one-third will appear in an upcoming Documentation (Errata) CR. Interestingly, many of the issues we found most compelling were not of interest to the RA. For example, we discovered a sub-optimal sequence in which an entry maneuver is potentially calculated twice because a test is executed after rather than before the calculation. However, since the Shuttle currently uses only around 50% of the available compute cycles, the anomaly was not considered important. The logical error listed below represents a significant error in the requirements that was also discovered by the existing requirements analysis process. To our knowledge, the other issues listed below had not been previously discovered.

Error Type	Number
undocumented assumption	3
inconsistent or anomalous terminology	10
redundant calculation	2
missing initialization	1
interface anomaly	2
logical error	1

3 Issues and Conclusions

In these concluding remarks, we touch on the issue of technology transfer and return to an earlier theme: the versatility of formal methods, using the 3E/O, JS, and GPS studies to illustrate the various roles and concomitant benefits available through the judicious application of formal techniques.

3.1 Technology Transfer

One aspect of technology transfer, demonstrating feasibility on important applications, has been addressed by the results of the case studies. Other issues pertain to the eventual uptake of formal methods by the aerospace industry. Working directly with industry on these studies has been helpful for all sides and has led to the following observations.

- *Perceived benefits:* Shuttle RAs are generally inclined to believe in the benefits of formalism and need only modest demonstrations such as GPS and 3E/O to convince them. RAs have long felt they lack adequately precise, mechanizable ways to express and analyze complex requirements.

- *Cost effectiveness:* Our case studies, largely carried out by experienced practitioners, showed that limited, tailored use of FM appears to be cost effective. Large projects, however, have a real and understandable need for quantifiable cost predictions. This remains an unmet need and could be an obstacle to convincing managers to adopt FM. The cost effectiveness of FM in the hands of novice users is still an open question.

- *Reading specifications:* PVS provides a formal specification language of considerable power while still preserving the syntactic flavor of modern programming languages. As a result, PVS specifications are quite readable by those with little prior exposure to PVS. The GPS RAs were able to read and understand the PVS specifications without becoming PVS practitioners, corroborating a similar experience in the AAMP5 project jointly undertaken by SRI and Collins Commercial Avionics [10], where the Collins engineers quickly became adept at reading PVS specifications of the AAMP5.

- *Writing specifications:* RAs have had less practice writing formal specifications, and still feel uncomfortable with it. Developing the skill set and experience necessary to write formal specifications will require more attention in a second round of technology transfer. Nevertheless, we are highly encouraged by our experience at a recent training course offered by NASA LaRC, where the participants (Loral RAs and JSC personnel) all specified and proved several problems drawn from aerospace applications.

- *Verification:* Not enough formal verification was performed by RAs in the case studies to draw valid conclusions. We anticipate a significant learning curve in this area, although the training course cited above was encouraging.

Overall, the outlook for industrial adoption of the type of formal methods we explored is promising. Continued insertion projects are needed to develop in-house expertise in selected companies. The best strategy is to team experienced practitioners with motivated application area experts and to tackle a feasible portion of a real application.

3.2 Formal Methods: Roles and Benefits

The role of formal methods on a project can vary along several dimensions. We consider first the variation conferred by the different levels of formal methods analysis available within a single technique or tool, the first level being specification only. Although a specification that has not been validated through proof can be aptly compared to a program that has not been debugged, there are nevertheless real benefits to be gained from modeling and formally specifying requirements, including the following.

- *Clarify Requirements:* A formal specification provides a concise and unambiguous statement of the underlying requirements, thereby exposing fundamental issues that tend to be obscured by lengthy informal statements. Our experience with JS illustrates this point nicely. Decoding the JS requirements documents required the combined efforts of the multi-center team over several months and relied extensively on resident expertise at Loral. However, when LaRC formalized the high-level JS requirements in approximately 500 lines of PVS, it became clear that the JS function is basically very simple and, with the help of the formal specification, could probably be communicated in less than a day to those with no prior JS exposure.

- *Articulate Implicit Assumptions:* Formalisms can help identify and supplement limitations in requirements methodology. The concept of state variables, for example, is not explicitly mentioned in FSSR-style requirements. Their existence must be inferred from context by noting when local variables appear to be persistent. Explicitly modeling these state variables has made the augmented GPS requirements clearer and more precise. Similarly, using Murϕ to explore the cyclic behavior of 3E/O exposed undocumented assumptions about the input space well before we introduced invariants.

- *Expose Flaws:* Although the GPS project has not yet undertaken any proofs, it has exposed a significant number of flaws in the requirements, especially among the subsystem interfaces. Applications like GPS that involve large, complex systems and immature requirements, can expect to realize substantial benefit from formal specification alone. Similarly, state exploration of the 3E/O specification revealed several potential issues even before introducing invariants.

Given a tool like PVS that offers a highly expressive language and strong typechecking that is inherently undecidable, the distinction between *specification only* and *specification with proof of simple properties* is somewhat blurred because specifications can not be considered type-correct until all type-correctness obligations have been discharged in the prover. Nevertheless, challenging a specification by proving properties with a proof-checker and checking invariants through state exploration represent a different level of activity that provides far greater assurances than specification alone, as noted in [16]. At this level, the benefits include:

- *Confirm/Disconfirm Key Properties:* System properties or constraints can be precisely stated and deductively verified. The JS specification was validated by proving a dozen lemmas derived from a list of expected JS properties. The property that JS shall never choose a failed jet proved to be false (i.e., the corresponding lemma was unprovable), exposing a genuine problem in mature requirements.

- *Debug Specifications:* Less significant properties that ought to be true about a specification can be stated as challenges and proved. When such a proof fails, it can be due to a real problem in the requirements or merely to a mistake in expressing the specifications.

A second source of versatility derives from the variety of formal methods techniques. As suggested in the GPS, JS, and 3E/O studies, there are many ways to model systems and calculate their properties, each of which confers different benefits.

- *Finite-State Verification:* These techniques, including state exploration and model checking, are good for specification debugging; counter example generation; automatic, rapid, and reliable exploration of variations; and exhaustive enumeration. Conversely, finite-state verification often demands more concrete specifications, which are typically poor vehicles for communicating and documenting systems and their properties, as we found in the case of 3E/O.
- *Proof Checking:* Deductive techniques support more varied and more abstract models, more expressive specification, more varied properties, and more reusable verifications. Conversely, theorem proving techniques are typically less automatic and invariably require more effort on the part of the analyst.

Furthermore, although JS, GPS, and 3E/O have each used a single formal methods technique, a further variation offering potentially greater productivity involves applying a combination of methods to a given problem by integrating model-checking, proof-checking, and techniques such as simulation.[6]

A formal specification, particularly one that has been verified, is best viewed as a point of departure, providing an effective basis for documenting, calculating, and predicting current system behavior and for analyzing future modifications and extensions. Reuse of formal methods products and strategies provides the best return on investment in formal specification and analysis. The most lasting contribution of the work described here has been the development of two reusable strategies and a clear demonstration of the tradeoffs offered by the versatility of formal methods techniques.

Acknowledgments

The authors are grateful for the assistance of the requirements analysts and other staff at Loral Space Information Systems: Larry Roberts, Mike Beims, Ron Avery, and, in earlier phases of this work, David Hamilton and Dan Bowman. All patiently answered questions and served up voluminous amounts of documentation. We would also like to thank John Kelly (JPL), John Rushby (SRI), Natarajan Shankar (SRI), Rick Butler (LaRC), and three anonymous reviewers. This work was supported in part by the National Aeronautics and Space Administration under Contract No. NAS1-19341 (VíGYAN) and NAS1-20334 (SRI).

[6]Several formal methods systems integrate model-checking and proof-checking, some more intimately than others. See [15] for a general discussion of the PVS approach and a summary of related efforts, and [13] for a detailed discussion of tabular and state-transition specifications and their verification using the integrated facilities of PVS.

References

[1] Ricky W. Butler, James L. Caldwell, Victor A. Carreño, C. Michael Holloway, Paul S. Miner, and Ben L. Di Vito. NASA Langley's Research and Technology Transfer Program in Formal Methods. In *Tenth Annual Conference on Computer Assurance (COMPASS 95)*, pages 135–149, Gaithersburg, MD, June 1995.

[2] Judy Crow. Finite-State Analysis of Space Shuttle Contingency Guidance Requirements. Technical Report SRI-CSL-95-17, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1995. Also forthcoming as a NASA Contractor Report for Task NAS1-20334.

[3] Ben L. Di Vito and Larry Roberts. Using Formal Methods to Assist in the Requirements Analysis of the Space Shuttle GPS Change Request. Contractor report, NASA Langley Research Center, Hampton, VA, 1996. To appear.

[4] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992. Cambridge, MA, October 11-14.

[5] David Hamilton, Rick Covington, and John Kelly. Experiences in Applying Formal Methods to the Analysis of Software and System Requirements. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 30–43, Boca Raton, FL, 1995. IEEE Computer Society.

[6] David Hamilton, Rick Covington, and Alice Lee. Experience Report on Requirements Reliability Engineering Using Formal Methods. In *ISSRE '95: International Conference on Software Reliability Engineering*, Toulouse, France, 1995. IEEE Computer Society.

[7] K. L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[8] C. Norris Ip and David L. Dill. Better Verification through Symmetry. In *CHDL '93: 11th Conference on Computer Hardware Description Languages and*

their Applications, pages 87–100. IFIP, 1993. Ottawa, Canada.

[9] Robyn R. Lutz and Yoko Ampo. Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software. In *19th Annual Software Engineering Workshop*, pages 231–248. NASA GSFC, 1994. Greenbelt, MD.

[10] Steven P. Miller and Mandayam Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[11] Multi-Center NASA Team from Jet Propulsion Laboratory, Johnson Space Center, and Langley Research Center. *Formal Methods Demonstration Project for Space Applications – Phase I Case Study: Space Shuttle Orbit DAP Jet Select*, December 1993. NASA Code Q Final Report (Unnumbered).

[12] National Research Council Committee for Review of Oversight Mechanisms for Space Shuttle Flight Software Processes, National Academy Press, Washington, DC. *An Assessment of Space Shuttle Flight Software Development Practices*, 1993.

[13] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing Tabular and State-Transition Specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1995.

[14] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[15] S. Rajan, N. Shankar, and M.K. Srivas. An Integration of Model-Checking with Automated Proof Checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[16] John Rushby. Formal Methods and the Certification of Critical Systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.

[17] A. John van Schouwen. The A-7 Requirements Model: Re-Examination for Real-Time Systems and an Application to Monitoring Systems. Technical Report 90-276, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, May 1990.