Formalizing Space Shuttle Software Requirements: Four Case Studies

JUDITH CROW
SRI International
and
BEN DI VITO
VíGYAN, Inc.

This article describes four case studies in which requirements for new flight software subsystems on NASA's Space Shuttle were analyzed using mechanically supported formal methods. Three of the studies used standard formal specification and verification techniques, and the fourth used state exploration. These applications illustrate two theses: (1) formal methods complement conventional requirements analysis processes effectively and (2) formal methods confer benefits even when only selectively adopted and applied. The studies also illustrate the interplay of application maturity level and formal methods strategy, especially in areas such as technology transfer, legacy applications, and rapid formalization, and they raise interesting issues in problem domain modeling and in tailoring formal techniques to applications.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; *tools*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*logics of programs*; *mechanical verification*; *specification techniques*

General Terms: Verification

Additional Key Words and Phrases: Flight software, formal methods, requirements analysis, Space Shuttle, state exploration, theorem proving

1. INTRODUCTION

Although Space Shuttle flight software is generally considered exemplary among NASA software development projects, requirements analysis and quality assurance in early life cycle phases still use processes and tools

This work was supported in part by the National Aeronautics and Space Administration under Contract No. NAS1-19341 (VíGYAN) and NAS1-20334 (SRI).
Authors' addresses: J. Crow, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025; email: crow@csl.sri.com; B. Di Vito, VíGYAN, Inc., 30 Research Drive, Hampton, VA 23666-1325; email: b.l.divito@larc.nasa.gov.

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, and software upgrades are continually introduced. Upgrades accommodate new missions such as the recent MIR docking; provide new capabilities such as Global Positioning System (GPS) navigation; enhance existing capabilities such as the crew displays for Heading Alignment Cylinder (HAC) initiation; and improve 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). These 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 [NRCC 1993, p. 91].[1]

The work described in this article had three main goals: first, to explore and document the feasibility and utility 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; and third, to identify and assess key factors in the transfer of this technology to the aerospace community.

The Shuttle subsystems selected for the project directly reflect the first two goals. GPS, JS, HAC, and 3E/O 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 and HAC requirements are relatively new. GPS, JS, and HAC 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 analyzed using standard theorem-proving techniques. By contrast, 3E/O represents a class of mode-sequencing software that can be quite naturally modeled as a finite-state system and effectively analyzed 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 GPS [Di Vito 1996; Di Vito and Roberts 1996], JS [NASA 1993, Appendix B; Rushby 1996], HAC [Roberts and Beims 1996], and 3E/O [Crow 1995] projects. The technical approach used for GPS is essentially the same as that for JS, except for the relatively minor differences noted in Section 3.1. HAC is a smaller example of the functional approach that examines control logic expressed in tabular form. Although the PVS (Prototype Verification System) and Murϕ tools (cf. Section 2.3) 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

[1]National Research Council Committee for Review of Oversight Mechanisms for Space Shuttle Flight Software Processes

requirements analysis process. The GPS, JS, HAC, and 3E/O projects each uncovered anomalies ranging from minor to substantive, most of which had not been detected by existing requirements analysis processes. A further result is a comparative demonstration of the role of formal methods across a range of maturity levels, underscoring the precept that formal methods confer benefits regardless of when or how extensively they are applied.

This article describes the four case studies and explores the issues and challenges they raise in the areas of technology transfer, legacy applications, rapid formalization, problem domain modeling, and technique tailoring. The implications of maturity of the application for formal methods strategy is a recurring theme throughout the discussion. In addition to the introductory and concluding remarks, the article consists of three main sections: Section 2 provides an overview of the project and its background, Section 3 presents the case studies, and Section 4 discusses their implications.

2. PROJECT BACKGROUND

We first introduce the context in which the case studies were performed, and the tools used to mechanize the analysis. The discussion includes an overview of project participants, a sketch of Shuttle software development, and a brief description of PVS and Murϕ.

2.1 Aerospace Applications

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

This article focuses exclusively on LaRC project activity,[2] which was performed in the context of a broader program of formal methods work [Butler et al. 1995]. The effort consisted of formalizing selected Shuttle software (sub)system modifications and analyzing key system properties using either SRI's Prototype Verification System (PVS) specification language and interactive proofchecker [Owre et al. 1995] or Stanford's Murϕ (pronounced "Murphy") finite-state verification system [Dill 1996].

The formalization and analysis described in the paper were carried out by the authors, with the exception of the JS study, which represents joint work with John Rushby (SRI), who authored the specification. This team was well versed in formal methods and in PVS and had significant

[2]Descriptions of some of the JPL, JSC, and Loral/Lockheed Martin activities undertaken for this project may be found in Hamilton et al. [1995a; 1995b], Lutz and Ampo [1994], and Roberts and Beims [1996]. The overall project is not large, roughly the equivalent of one full-time position at each of the three NASA centers per year, including subcontractor sites.

experience applying formal methods to industrial problems. On the other hand, the authors had only minimal exposure to Murϕ prior to the 3E/O work.

2.2 Shuttle Software

NASA's prime contractor for the Space Shuttle is the Space Systems Division of Boeing North American, Inc. Lockheed Martin Space Information Systems is its software subcontractor. Draper Laboratory serves the Space Systems Division, 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 pseudocode, 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 are issued incorporating one or more CRs. Shuttle CRs are written as modifications, replacements, or additions to existing FSSRs. Lockheed Martin 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 life cycle.

2.3 Summary of PVS and Murϕ

PVS is an environment for specification and verification developed at SRI International's Computer Science Laboratory [Owre et al. 1995]. 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. A summary of the PVS language features used in this article appears in Appendix A.

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," that is, to check that all reachable states satisfy specified properties [Dill 1996]. Murϕ can explore millions of states in a matter of minutes. A brief overview of Murϕ appears in Appendix B.

3. FORMALIZATION STRATEGIES

We discuss two general strategies, one having several variants, and illustrate their application to the GPS, JS, HAC, and 3E/O requirements.

3.1 Concrete Functional Specification

Formalization of both GPS and JS used the functional specification strategy outlined in Section 3.1.2. However, while GPS and JS share the same technical approach, they differ in interesting ways. Since the CR was still under development when the GPS formalization began, traceability between specification and CR was particularly important and dictated a concrete specification style. Additionally, the need to continually refine the specification to accommodate a series of revisions in the CR led to a focus on specification of GPS rather than on proof of GPS behavioral properties. The situation with JS was somewhat different. The JS CR was reasonably stable, and the concerted effort required to understand and in many cases reengineer the JS application yielded a fairly abstract, high-level specification that facilitated the formalization and proof of key JS properties.

We begin with a description of the GPS application and its formalization using a concrete style.

3.1.1 *Overview of GPS*. The GPS retrofit was planned in anticipation of the U.S. Department of Defense phaseout of the existing ground-based radio-navigation network. Shuttle vehicles are to be outfitted with GPS receivers, and additional navigation software is to 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 adapts 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 modify GPS state vectors for consumption by the existing entry navigation software.

3.1.2 *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

```
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> #)
```

Fig. 1. PVS model of a Shuttle *principal function*.

values. This method provides a simple computational model similar to popular state-based methods such as the A-7 model [Heninger 1980; van Schouwen 1990].

One transition of the state-machine model corresponds to an execution of the principal function, for example, one cycle at 6.25Hz or other applicable rate. The state-machine transition function M 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.

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

This function M is expressed in PVS and forms the central part of the formal specification.

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), 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 output and next-state expressions in the general form above describe the effects of invoking the subfunctions belonging to the principal function. In the CR, GPS Receiver State Processing is decomposed into six subfunctions, and GPS Reference State Processing into four. Figure 2 illustrates one of the subfunctions from GPS Receiver State Processing. 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.

Formalization of the two principal functions in PVS was completed and revised three times to keep up with requirements changes ("Mods"). Because of the breadth of this CR, convergence was slow. Requirements changes were frequent and extensive as the CR was worked through the review process. Our initial formal specification was based on a preliminary version of the CR, before a two-phase implementation plan (single-receiver

```
ref_state_anncd_reset_out: TYPE = [#
     G_ref_anncd_reset:        GPS_accelerations,
     GPS_anncd_reset:          GPS_predicate,
     GPS_anncd_reset_avail:    GPS_predicate,
     R_ref_anncd_reset:        GPS_positions,
     T_anncd_reset:            GPS_times,
     T_ref_anncd_reset:        GPS_times,
     V_IMU_ref_anncd_reset:    GPS_velocities,
     V_ref_anncd_reset:        GPS_velocities
     #]

ref_state_announced_reset ( DT_anncd_reset:   delta_time,
                            G_two:            GPS_accelerations,
                            GPS_DG_SF:        GPS_predicate,
                            GPS_SW_cap:       num_GPS,
                            R_GPS:            GPS_positions,
                            T_anncd_reset:    GPS_times,
                            T_current_filt:   mission_time,
                            T_GPS:            GPS_times,
                            V_current_GPS:    GPS_velocities,
                            V_GPS:            GPS_velocities
                            ) : ref_state_anncd_reset_out =

(# G_ref_anncd_reset     :=
     LAMBDA I: IF GPS_DG_SF(I) THEN G_two(I) ELSE null_acceleration ENDIF,
   GPS_anncd_reset       :=
     LAMBDA I: GPS_DG_SF(I) AND
               (T_current_filt - T_anncd_reset(I) > DT_anncd_reset),
   GPS_anncd_reset_avail := GPS_DG_SF,
   R_ref_anncd_reset     :=
     LAMBDA I: IF GPS_DG_SF(I) THEN R_GPS(I) ELSE null_position ENDIF,
   T_anncd_reset         :=
     LAMBDA I: IF GPS_DG_SF(I) AND
                    (T_current_filt - T_anncd_reset(I) > DT_anncd_reset)
                 THEN T_current_filt ELSE null_mission_time ENDIF,
   T_ref_anncd_reset     :=
     LAMBDA I: IF GPS_DG_SF(I) THEN T_GPS(I) ELSE null_mission_time ENDIF,
   V_IMU_ref_anncd_reset :=
     LAMBDA I: IF GPS_DG_SF(I)
                 THEN V_current_GPS(I) ELSE null_velocity ENDIF,
   V_ref_anncd_reset     :=
     LAMBDA I: IF GPS_DG_SF(I) THEN V_GPS(I) ELSE null_velocity ENDIF
 #)
```

Fig. 2. Sample subfunction of GPS receiver state processing specified in PVS.

followed by three-receiver) was adopted. Subsequent versions were written to model the single-string GPS CR and its modifications.

PVS versions were written for Mod B, Mod D/E, and Mod F/G of the CR. Each modification was followed by a requirements inspection, accompanied by various issues written against the corresponding version of the CR. Some, but not all, of the responses to these issues were incorporated into the following modification. Significant requirements changes were still being inserted at each of these modification stages.

The formalization was straightforward, with only minor technical difficulties, the most notable of which are the following:

—State variables are not explicitly identified in Shuttle requirements; the existence of state variables had to be inferred from algorithmic details and contextual clues.

—Read-only and write-only variables are commonly used as flags and appear as principal-function inputs and outputs (respectively), but are not identified as such in Shuttle requirements because these variables are typically not accessed by the external environment. It was necessary to introduce pseudo-inputs and pseudo-outputs to reflect the existence and role of these variables.

—Shuttle requirements are written in a procedural style; variable values are often specified conditionally or left "undefined." Since PVS requires total functions, simulated constants had to be introduced for undefined values (for example, `null_velocity` in Figure 2).

The subset of the GPS requirements used in this study consists of approximately 110 pages of prose, pseudocode, and tables. The two GPS principal functions were formalized in about 3300 lines of PVS specifications (including comments and blank lines), packaged as 11 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.

3.1.3 *GPS Results.* The formal modeling step demonstrated that it is not difficult to bring the precision of formalization to bear on the type of requirements we examined. Expressing the requirements in the language of an off-the-shelf verification methodology was straightforward. The most surprising result was the sheer number of problems in the requirements exposed simply as a consequence of carrying out the formalization. All of the errors identified resulted from the process of specification and from carrying the analysis only to the point of typechecking. It was our intention to conduct some deductive analysis (theorem proving) as well, but this could not be accomplished within the limited scope of the project.

3.1.3.1 *Types of Requirements Issues.* Feedback from requirements analysts indicated that our approach was helpful in detecting three classes of errors normally tracked by the Shuttle program:

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

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

—Type 9: interfaces inconsistent.

An example of a Type 4 error encountered in the CR is omission due to conditionally updating variables. Suppose that, 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.

3.1.3.2 *Summary Statistics*. All requirements issues detected during the formalization were passed on to Loral (now Lockheed Martin) representatives. Those deemed to be real issues, that is, not caused by the misunderstandings of an outsider, were then officially submitted on behalf of the formal methods analysis as ones to be addressed during the requirements inspections. Severity levels are attached to valid issues during the inspections. This allowed us to get “credit” for identifying problems and led to some rudimentary measurements on the effectiveness of formalization.

Table I summarizes a preliminary accounting of the issues identified during our analysis. The issues are broken out by severity level for the three inspections of the CR that took place. A grand total of 86 issues were submitted for the three inspections. Of these 86 issues, 72 were of Type 9 (interfaces inconsistent). The rest were primarily scattered among Type 4 (requirements do not meet CR author’s intent) and Type 6 (requirements not technically clear, understandable, and maintainable). While Shuttle RAs felt these issues would have been discovered eventually, they also felt that the formalization helped discover them earlier than normal.

The meaning of the severity categories used in Table I is as follows:

(1) *High Major*: Cannot implement requirement.

(2) *Low Major*: Requirement does not correctly reflect CR author’s intent.

(3) *High Minor*: “Support” requirements are incorrect or confusing.

(4) *Low Minor*: Minor documentation changes.

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. Lightweight analysis applied early tends to detect primarily superficial problems and to do so quickly, whereas proof-oriented analysis tends to expose fewer, but more subtle,

Table I. Summary of GPS Issues Detected by Formal Methods

Issue Severity	Mod B	Mod D/E	Mod F/G	Totals
High Major	1	0	0	1
Low Major	7	3	0	10
High Minor	19	40	6	65
Low Minor	8	0	2	10
Totals	35	43	8	86

problems. The GPS study suggests the potential of lightweight methods, while the JS application illustrates the utility of more powerful techniques.

3.2 Abstract Functional Specification

While GPS illustrates the role of specification (only) on a relatively new CR, JS illustrates the role of specification and proof on a mature CR. A fairly abstract, high-level specification was created to meet the proof objective.

3.2.1 *Overview of JS*. JS is the function responsible for selecting which of the Shuttle's Reaction Control System jets should be fired to accomplish a given translational or rotational acceleration. JS has two main operating modes: *primary* and *Vernier/Alt*. The former deals with translation and is determined largely by table lookup. The latter deals with rotation and is determined algorithmically. Formalization of the JS CR focused exclusively on the Vernier/Alt component. The slash (/) in Vernier/Alt reflects the fact that this component can operate in one of two modes: Vernier mode in which as many as three of the six small "vernier" jets are selected and Alt(ernate) mode in which as many as three of the large "primary" jets are selected. The 38 primary jets are arranged in 14 groups of two to four jets each, depending on jet location and firing direction. Only 11 of these groups (31 jets) are used for rotational maneuvers.

In both Vernier and Alt modes, jet selection is performed by an algorithm known as "max-dot-product."[3] The max-dot-product algorithm uses a set of tables that record the rotational velocity vector giving the angular acceleration achieved by firing a given jet (or member of a given group of jets) for a single 80ms cycle. The tables are parameterized with respect to where the Shuttle's robotic arm is positioned and the mass of the payload (if any) attached to the arm. The algorithm first selects the vernier jet or the group of primary jets whose acceleration has the largest scalar (dot) product with the desired rotational acceleration vector. If second and third jets are required, they are similarly selected on the basis of the second and third largest scalar products. The second and third jets must also satisfy certain threshold constraints. If three jets satisfying the given thresholds cannot be found, the algorithm considers pairs, or, as a last resort, single jets. The

[3]Formalization of JS was undertaken partly in anticipation of a new jet selection algorithm that was to be introduced in a subsequent CR.

algorithm selects vernier jets singly and primary jets in terms of groups. The primary jets within each group are ranked in a priority order specified by the crew and ground control to minimize wear on the jets and their manifolds.

In Alt mode, the algorithm selects as many as three groups and then chooses the available jet of highest priority from each (selected) group. Availability is determined by failure status, crew (de)selection, and a set of additional constraints implemented as *modes*. *Low +z* mode excludes three groups of primary jets (one on the nose, two on the tail) whose upward-directed plumes may contaminate the cargo area or payload. *Nom* (nominal, that is, both nose and tail), *tail-only*, and *nose-only* modes restrict available primary jets as indicated in order to balance propellant consumption across various tanks. Only *nom* and *tail-only* are available in Alt mode.

3.2.2 *Technical Approach*. The technical approach used to model JS is essentially the same as that used for GPS (Section 3.1.2). Although the abstract state-machine model developed to model JS was refined somewhat to accommodate the significantly larger GPS application, the differences between the two approaches are minor and consist largely of the somewhat less stylized format and more aggressive abstraction used in JS.

The primary objective in applying formal methods to JS was to specify the desired functionality as clearly as possible, providing a readily understood, high-level characterization of the JS algorithm and its behavioral properties. Since the JS documentation consisted largely of low-level diagrams with virtually no explanatory material, the understanding necessary for a concise, accurate characterization required considerable reverse engineering and a process of refining and distilling the specification over several iterations of specification and proof. The specification of the RCS jets in Figure 3 provides a good illustration of the level of abstraction at which JS was modeled. The exact numbers and locations of jets are not relevant at this level of description: it is sufficient to state that there are certain things called "jets," some of which are "verniers" and others of which are "primary." Furthermore, these two kinds of jets are disjoint and exhaustive (that is, there are no other kinds of jets). Additionally, some of the primary jets have a "+z plume," some of the verniers are "downfiring," and so on. These distinctions are stated formally in PVS using uninterpreted types, subtypes, and predicates. Constraints on the types are stated axiomatically. To ensure mathematical soundness, it is also necessary to specify (and prove) that there is at least one jet of each kind and that the number of jets is finite.

JS is part of a control loop; it receives a set of inputs once every cycle and produces a corresponding set of outputs on the same cycle. The JS algorithm described in Section 3.2.1 computes the set of jets to fire in the current cycle as a (pure) function of the inputs to that cycle. However, three additional considerations make it necessary to introduce and retain a limited amount of state data from one cycle to the next. First, one of the JS outputs is a boolean flag (`vfail`) indicating whether any of the four

```
finite_sequences[n: posnat, t: TYPE]: THEORY
BEGIN

  finite_seq: TYPE = [below[n] -> t]

END finite_sequences

requirements: THEORY
BEGIN
IMPORTING finite_sequences
.. .

  jet: TYPE
  rot_jet: TYPE FROM jet

  jet_bound: posint
  jet_count: [rot_jet -> below[jet_bound]]
  jet_count_ax: AXIOM injective?(jet_count)

  finite_number_of_jets: LEMMA .. .

  vernier?: pred[rot_jet]
  vernier_jet: TYPE = (vernier?)
  there_is_a_vernier: AXIOM (EXISTS j: vernier?(j))

  primary_rot?(j): bool = NOT vernier?(j)
  primary_rot_jet: TYPE = (primary_rot?)
  there_is_a_primary: AXIOM
    (EXISTS j: primary_rot?(j))

  downfiring?: pred[vernier_jet]

.. .
END requirements
```

Fig. 3. PVS specification of primary jets used for rotation.

downfiring vernier jets has become unavailable on the current cycle, requiring a comparison with the value of the flag in the previous cycle. Second, the JS CR specifies an optimization in which vernier jets are "reused," i.e., jet selection is not recomputed for a certain number of cycles as long as the commanded direction remains constant and `vfail` remains *false*. Again, this determination requires comparison with values from the previous cycle. Last, the powerful primary jets are pulsed on and off in Alt mode, and state data are necessary for calculating the appropriate on-off cycles. To accommodate this state data while retaining the essentially functional nature of the algorithm, JS was specified as a function that maps a finite sequence of inputs into a corresponding sequence of outputs. Whereas the GPS specification uses an explicit representation of state, the JS specification models state data implicitly in terms of traces or sequences of inputs. Access to state data, that is, to earlier inputs (outputs), is obtained by sequence indexing. Although retaining and accessing the entire state each time state information is required would involve unacceptable diseconomies of storage and computation in an implementation of JS, such issues are not relevant at the level of requirements specification. Formally, `jet_select` is defined as an uninterpreted, higher-order function that takes and returns functions (sequences) as argument and result, respectively. The PVS fragment corresponding to this description appears in Figure 4.

```
   jon_count: TYPE = { n:nat | n=1 OR n=2 OR n=3 }

   direction: TYPE = [# roll: real, pitch: real,
                        yaw: real #]

   major_mode: TYPE = { primary, vernier, alt }
   position_mode: TYPE = { nose, tail, any }
   algorithm: TYPE = {dot_product, .. . }
   priorities: TYPE = [primary_rot_jet -> posint]
   payload_config: TYPE


   input: TYPE = [# dir: direction,
                    avail?: pred[rot_jet],
                    m_mode: major_mode,
                    low_z_mode: bool,
                    pos_mode: position_mode,
                    alg_mode: algorithm,
                    priority: priorities,
                    pc: payload_config,
                    max_jets: jon_count
                 #]

   output: TYPE = [# jon: setof[rot_jet],
                     any_on?: bool,
                     vfail?: bool,
                     ang_incs: direction
                  #]

   max_cycles: posint
   inseq: TYPE = finite_seq[max_cycles, input]
   outseq: TYPE = finite_seq[max_cycles, output]
   in_trace: VAR inseq
   index: TYPE = below[max_cycles]
   i: VAR index

   jet_select: [inseq -> outseq]

.. .
```

Fig. 4. PVS specification of jet select function.

The technical difficulties encountered in formalizing JS derived primarily from two sources: documentation and the formalization of mathematical theories in PVS. The first challenge was to define an appropriate level of abstraction, given that the available documentation captured extremely low level requirements in a notation geared toward implementation rather than specification. The considerable reverse engineering necessary to clarify the JS algorithm relied heavily on the expertise of Shuttle RAs and on the HAL/S code implementation, and resulted in many iterations of simplifying and abstracting the specification. The time required to formalize and analyze JS also reflects the development of PVS theories for the underlying mathematics, including properties of finiteness, injectivity, surjectivity, and left and right inverses. Five of the eight PVS theories for JS develop this mathematical background, which has subsequently become part of the

```
no_failed_jets_v: LEMMA
  m_mode(in_trace(i))=vernier
    AND alg_mode(in_trace(i))=dot_product
      AND member(j, jon(jet_select(in_trace)(i)))
        => member(j, avail?(in_trace(i)))
```

Fig. 5. PVS specification of no failed jets.

standard PVS libraries or of the PVS "prelude" of built-in theories. The five background theories involved the most challenging and time-consuming specification and proof encountered in the JS project, and added significantly to the time and effort required to formalize JS.[4]

The JS requirements explored in this study consist of roughly 70 pages of tables and low-level diagrams. The complete JS specification consists of approximately 500 lines of PVS (including blank lines, but excluding comments), organized as eight PVS theories. As noted above, three of these theories are specific to JS, while the remaining five develop mathematical background. Reverse engineering the JS application and developing the technical approach took approximately one staff month over a three-month period. The specification itself was written and the proofs completed within a matter of weeks.

3.2.3 *JS Results.* The high-level specification of JS as the iterative selection of the three jets or groups of jets that best satisfy the constraints of availability and priority yielded several distinct benefits. Successive rounds of specification and proof yielded a precise formal description of a function whose relative simplicity had been obscured by the low-level diagrams and implementation bias with which it had been documented. For example, the high-level specification captured the fact that the max dot-product algorithm is essentially the same in both Vernier and Alt modes—a fact that was not obvious from the JS CR and might have been overlooked at a lower level of formalization.

The specification was validated by proving a dozen lemmas derived from a list of expected properties provided by our colleagues at IBM (now Lockheed Martin). These high-level properties characterize the capabilities JS should provide and the constraints its outputs should obey. For example, one of the output requirements states that *JS shall never choose a jet that has been designated as failed*. The PVS lemma corresponding to the Vernier mode interpretation of this requirement appears in Figure 5.

In our experience, failed proofs are far more instructive than successful ones; they serve to debug and refine the specification as well as to expose problems in the modeled system or its requirements. Of the 12 specified properties, only the lemma in Figure 5 could not be proved after refinements to the specification. In this case, the failed proof confirmed our

[4]In the period following the completion of JS, there has been a concerted effort in the PVS community to develop general-purpose libraries, although as noted in Dutertre and Stavridou [1997], further library development is essential if formal verification is to become less labor intensive.

ADI Error Scale Deflection — Entry/Landing			
Major Mode Switch Position	Roll Error Full Scale Value	Pitch Error Full Scale Value	Yaw Error Full Scale Value
Major Modes 301–303	Deg	Deg	Deg
High	10	10	10
Med	5	5	5
Low	1	1	1
Major Mode 304 or 602 when IPHASE = 4 or 6	Deg	Deg	Deg
High	25	5	2.5
Med	25	2	2.5
Low	10	1	2.5
Major Mode 602 when IPHASE = 5	Deg	G	Deg
High	25	1.25	2.5
Med	25	1.25	2.5
Low	10	0.5	2.5
Major Mode 305 or 603 when WOWLON = 0	Deg	G	Deg
High	25	1.25	2.5
Med	25	1.25	2.5
Low	10	0.5	2.5
WOWLON = 1	Deg	Deg/sec	Deg
High	20	10	2.5
Med	5	5	2.5
Low	1	1	2.5

Fig. 6. Sample HAC requirements table.

suspicion that the Vernier mode optimization (see Section 3.2.2) potentially allows selection of failed nondownfiring (that is, sideways) verniers. This issue represents a genuine problem in mature requirements and was the most significant of six provocative issues raised by the JS formalization. These issues were found in the iterative process of specification and proof, rather than isolated as errors, for example, in the typechecker.

3.3 Tabular Specification

This section describes a somewhat smaller Shuttle CR that was modeled in a functional style using a tabular format.

3.3.1 *Overview of HAC.* During the final phase of Shuttle flight, the orbiter must enter a "heading alignment cylinder" (HAC) to be properly positioned for its landing approach. Previous flight history revealed that the initial bank onto the HAC is the most critical "terminal area energy management" maneuver, and a low-energy problem is not obvious to the crew until it becomes severe. The HAC CR was introduced to provide HAC initiation prediction and position error displays to give the crew more insight into vehicle energy conditions, thereby increasing situational awareness and improving safety.

A subset of the display enhancements was examined and chosen for modeling during the formal methods study. The attitude direction indicator (`ADI`) and horizontal situation display were modified to incorporate time to HAC intercept, crosstrack error, and altitude error. Key portions of the control logic from this requirements subset are expressed in the CR in a tabular form. Parameters to be displayed and conditions under which various parameter values should be displayed are called out in two requirements tables. Figure 6 shows one of these tables; the other is similar but three times as large.

3.3.2 *Technical Approach.* The method used to model the HAC requirements exploits the PVS TABLE feature, and uses PVS type correctness conditions (TCCs), which include coverage and disjointness proof obligations (TCCs) for PVS TABLE constructs, to check the consistency and completeness of the enabling conditions. Each requirements table is specified in PVS in three steps: (1) introduce type declarations unique to the `ADI` tables, (2) introduce constant declarations for I-load values referenced by the tables, and (3) formulate the table as a PVS function with its defining expression written in TABLE form. Two function definitions were constructed in this manner: `ADI_error_scale_deflection` and `ADI_rate_scale_deflection`. The first of these, which models the requirements of Figure 6, is shown in Figure 7. The tabular PVS specification is an almost verbatim encoding of the original requirements table, suggesting the utility of certain styles of formal specification for requirements capture.

The HAC specification makes extensive use of dependent types in PVS (see Appendix A). Dependencies among the `ADI` variables referenced within the tables were captured using this mechanism. Note that the PVS TABLE constructs were designed for concise tables where each row fits on one line. The HAC tables turned out to be somewhat larger than assumed for the most convenient use of this feature. This caused some rows within the TABLE syntax to be stretched over several lines, but the resulting tables are still quite readable.

Although the HAC formalization encountered few technical difficulties, in large part because of its modest scope, it does raise the issue of scalability. The use of domain knowledge in the HAC study is manageable, but the approach may not be viable for analysis of a large system. For example, if the relationships among variables are complex, the theorem-proving effort necessary to establish disjointness and coverage TCCs could grow accordingly. Table size is also a concern, given that disjointness TCCs (cf. Section 3.3.3) grow quadratically. Practical limitations of the approach were not explored during the study. The five minutes of computation time needed to prove the HAC TCCs suggest that the limits are not too far off, although techniques for analyzing the completeness and consistency of large state-based specifications using tabular representations have been developed [Heimdahl and Leveson 1996].

The HAC requirements consist of roughly 20 pages of prose, pseudocode, and tables. The PVS specifications for the two `ADI` tables were expressed in

```
switch_position: TYPE = {low, medium, high}
major_mode:      TYPE = {mm301, mm302, mm303, mm304, mm305, mm602, mm603}
iphase:          TYPE = {n: nat | n <= 6} CONTAINING 0

ADI_error_inputs: TYPE =
    [# mode: major_mode,
       switch_position: switch_position,
       iphase: {p: iphase | (mode = mm602 => p >= 4) AND
                             ((mode = mm305 OR mode = mm603) => p <= 3)},
       wowlon: {b: bool | b => (mode = mm305 OR mode = mm603)} #]

ADI_error_scale_deflection(A: ADI_error_inputs) : [real, real, real] =

LET mode = mode(A), switch_position = switch_position(A),
    iphase = iphase(A), wowlon = wowlon(A) IN
  TABLE  %  Result is of form: [roll error, pitch error, yaw error]
                         ,                 switch_position
                        %-----------------------------------------------------%
                        |[     high        |      medium       |       low       ]|
    %-----------------------------------------------------------------------------%
    | mode = mm301 OR
      mode = mm302 OR
      mode = mm303      | (10, 10, 10)     | (5, 5, 5)        | (1, 1, 1)        ||
    %-----------------------------------------------------------------------------%
    | mode = mm304 OR
      (mode = mm602 AND
       (iphase = 4 OR
        iphase = 6))    | (25, 5, 5/2)     | (25, 2, 5/2)     | (10, 1, 5/2)     ||
    %-----------------------------------------------------------------------------%
    | mode = mm602 AND
      iphase = 5        | (25, 5/4, 5/2)   | (25, 5/4, 5/2)   | (10, 1/2, 5/2)   ||
    %-----------------------------------------------------------------------------%
    | (mode = mm305 OR
       mode = mm603) AND
       NOT wowlon       | (25, 5/4, 5/2)   | (25, 5/4, 5/2)   | (10, 1/2, 5/2)   ||
    %-----------------------------------------------------------------------------%
    | wowlon            | (20, 10, 5/2)    | (5, 5, 5/2)      | (1, 1, 5/2)      ||
    %-----------------------------------------------------------------------------%
  ENDTABLE
```

Fig. 7. Sample HAC requirements table expressed in PVS.

about 200 lines, requiring roughly two weeks to formulate, iterate, and revise with the cooperation of cognizant Shuttle RAs.

3.3.3 *HAC Results.* Our goal was to analyze the two tables for consistent and complete conditions. Examining the table in Figure 6, it is easy to see that the conditions could not be exhaustive based on the information in the table alone. What is needed is a set of additional conditions modeling the relationships among variables. Expressing these problem-domain constraints in a suitable manner allows the proper analysis to be conducted without raising any false alarms.

Based on the semantic knowledge provided by the Shuttle RAs, several unstated assumptions were identified and added to the model as constraints:

—`major_mode = 602 IMPLIES iphase` $\geq$ `4.`

—`major_mode = 305 OR 603 IMPLIES iphase` $\leq$ `3.`

—`tg_end IMPLIES major_mode = 305 OR 603.`

—`wowlon IMPLIES tg_end.`

The dependent type feature of PVS was used to represent these constraints, affording some convenience when analyzing the coverage and disjointness conditions as TCCs. After two iterations to arrive at a stable formulation of the constraints, an effective analysis procedure emerged. Because of the TCC-based nature of the analysis, it was possible to obtain results by using a PVS command that typechecks a theory and automatically proves its TCCs, relying on the built-in strategies of the theorem prover to establish the table properties.

After incorporating all the constraints identified by the Shuttle RAs, we found that one of the tables was still inconsistent, i.e., several conditions overlapped. Manual inspection of the table `ADI_rate_scale_deflection` (not shown) and the failed TCC proof attempt revealed that several rows of the table had underspecified conditions, allowing them to overlap when the variable `tg_end` was true. By adding the conjunct "`NOT tg_end`" to several rows, a version of the table was constructed that was found to satisfy the disjointness criterion. Shuttle RAs were informed of this apparent flaw and the other analysis results, and this information was fed back to the original requirements authors.

Requirements analysis of the `ADI` error deflection and rate scale deflection tables from the HAC CR showed that the well-formedness analysis of certain tabular requirements is readily carried out. By selectively making explicit domain knowledge previously hidden in the form of unstated assumptions, it is possible to develop a formal specification that can be mechanically analyzed using a verification system such as PVS. The availability of a general-purpose theorem prover enabled direct expression of the domain constraints and their use in deductive analysis. In addition to the analysis, RAs found the identification of relevant domain knowledge valuable in its own right.

3.4 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.

3.4.1 *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 second; 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.

3.4.2 *Technical Approach.* 3E/O is typical of most fault-handling logic in the following respects: it consists largely of mode switching and exception handling, exhibits virtually no algorithmic complexity, and its input and state spaces lack a regular and easily characterizable structure. Primarily because of these three characteristics, conventional specification and verification strategies that use typechecking 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 crucial properties of the 3E/O sequencing algorithm as invariants that must hold in all reachable states. Key elements of this approach are elaborated below.

The challenge in modeling Shuttle flight software typically derives from the number of input variables[5] 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 (for example, current dynamic pressure) and a local environment consisting of variables representing the Shuttle's internal status.[6] To provide a reasonably tractable and accurate model of the input space, we used the following simplifying assumptions to develop a suitable abstraction.

[5]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.

[6]This strategy is reminiscent of the standard A-7 approach [Heninger 1980; van Schouwen 1990], but differs in the way the environment is modeled.

(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.[7]

(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, that is, `v_horiz_dnrng < 0 AND delta_r > del_r_usp`? The value of the operation in each conjunct is either true or false. 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 certain properties hold for all possible values of the two (original) expressions. We use this strategy for all inputs that represent the Shuttle's physical environment.

Figure 8 shows a template for the Murϕ rules used to generate input values for external physical parameters, and Figure 9 illustrates its use in

[7]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;
```

Fig. 8. Murϕ abstract syntax used to model Shuttle's physical environment.

```
Ruleset vel: velocity Do
Ruleset apogee_alt_LT_alt_ref: boolean Do
Ruleset q_bar: dynamic_pressure Do
Ruleset h_dot_LT_hdot_reg2: boolean Do
Ruleset alpha_n_GTR_alpha_reg2: boolean Do
Ruleset v_horiz_dnrng_LT_0: boolean Do
Ruleset delta_r_GTR_del_r_usp: boolean Do

Rule "Select Region"
  !cont_3EO_start
==>
Begin
  r:= reg_sel(vel,q_bar,delta_r_GTR_del_r_usp,
              v_horiz_dnrng_LT_0,
              alpha_n_GTR_alpha_reg2,
              h_dot_LT_hdot_reg2,
              apogee_alt_LT_alt_ref);
  region_selected := true;
  If meco_confirmed & r != reg0
     Then cont_3EO_start := true Endif;
Endrule;

Endruleset;  -- delta_r
Endruleset;  -- v_horiz
Endruleset;  -- alpha
Endruleset;  -- h_dot
Endruleset;  -- q_bar
Endruleset;  -- apogee
Endruleset;  -- vel
```

Fig. 9. Use of Murϕ rulesets to define 3E/O region selection.

the definition of a top-level rule for region selection. The *Ruleset* construct is syntactic sugar that generates a copy of the rules within its scope for every value of the bound variable. For example, the variable `vel` in Figure 9 ranges over the enumerated type `velocity`, and therefore can take on the following three values: `GTR_vi_3eo_max`, `GTR_vi_3eo_min`, and `LEQ_vi_3eo_min`. There is a single rule, `Select Region`, within the scope of this ruleset, so the ruleset will generate three rules identical in all respects except for the value of the variable `vel`. The deeply nested rulesets are used to generate all possible combinations of values for the variables that model the Shuttle's ascent environment. As noted previously, this approach yields a complete, if overly simplistic, model of the input space.

```
cont_3EO_start ->
   ((m_mode=mm102 | meco_confirmed) & !Isundefined(r) & r!=reg0);
```

Fig. 10. 3E/O sequencing invariant.

```
(r = regE) ->
!((m_mode = mm103 &
    (vel = GTR_vi_3eo_max |
       (vel = GTR_vi_3eo_min & apogee_alt_LT_alt_ref))) |
  (m_mode = mm601 & v_horiz_dnrng_LT_0 & delta_r_GTR_del_r_usp))
```

Fig. 11. 3E/O region zero (debugging) invariant.

Inputs that represent the Shuttle's internal state, for example, 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) internal state 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, that is, expressions that must be true in all reachable states. For example, the Murϕ invariant shown in Figure 10 is used to check that a contingency abort is initiated *only* after an abort maneuver region has been selected and main engine cutoff has been confirmed.[8] In Murϕ, "!" denotes negation; "|" denotes disjunction; "&" denotes conjunction; and "--" begins a comment.

Invariants were also used to debug the specification and to explore implications of the requirements. For example, we used an invariant similar to the one reproduced in Figure 11 to generate a counterexample demonstrating that the 3E/O requirements allowed a region 0 assignment (indicating a nominal ascent) to persist from one cycle to the next, resulting in failure to detect changes that would ordinarily trigger an abort maneuver.[9] The negated expression on the right-hand side of the implication conjoins all the conditions under which region 0 should be assigned.

The most difficult part of the 3E/O study was defining an effective model of the physical environment and developing a strategy for representing the model in Murϕ. The size and inherent complexity of the physical domain, and the lack of precedent in applying model checking to this type of application, led to many iterations of experimentation and refinement. Initial attempts at representing the suitably abstracted model failed, because the resulting Murϕ program was too large and could not be compiled. Eliminating system functionality and environmental variables not germane to sequencing behavior, using qualitative ranges and boolean

[8]For reasons beyond the scope of this article, the invariant does not hold in `major_mode 102`.
[9]This anomaly was ultimately attributed to an unstated assumption underlying the requirements statement, rather than to a potentially serious oversight.

Table II. Summary of 3E/O Issues Detected by Formal Methods

Error Type	Number
Undocumented Assumption	3
Inconsistent or Anomalous Terminology	10
Redundant Calculation	2
Missing Initialization	1
Interface Anomaly	2
Logical Error	1

values to model the physical environment, and constraining the scope of environment variables through judicious use of Murϕ rulesets ultimately yielded a viable Murϕ verifier.

The 3E/O requirements consist of roughly 70 pages of prose, pseudocode, flowcharts, and tables. The two main 3E/O functions were specified in approximately 1200 lines of Murϕ code including comments, using approximately a dozen invariants. Familiarization with the CR and with Murϕ took approximately two months full-time equivalent, and the specification and analysis occupied an additional one and a half months.

3.4.3 *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 listed in Table II and reported to the 3E/O RA, roughly one-third will appear in an upcoming Documentation (Errata) CR. The logical error listed in Table II represents a significant error in the requirements that was also discovered by the existing requirements analysis process. To our knowledge, the other issues had not been previously discovered.

3.5 Summary of the Case Studies

The four case studies are summarized in Table III. For each application, the table gives requirement size in pages, type of analysis employed, automated tool used, size in lines of code, and level of effort. Level of effort is estimated in terms of full-time equivalent (FTE), and distinguishes the process of acquiring a working knowledge of the application and its domain (preparation) from the process of formalization and analysis (FM). Requirement size is included as a rough measure of application size.

4. INSIGHTS AND CHALLENGES

Several insights and future challenges emerge from the four case studies. This discussion focuses primarily on the interplay of maturity level and formal methods strategy in the areas of technology transfer, legacy applications, and rapid formalization, and on issues involved in problem domain modeling and in tailoring techniques to applications.

Table III. Case Study Summary

Case Study	CR Size (Pages)	Analysis Technique	Tool Used	Specification Size (Lines)	Level of Effort (FTE Months)	
					Prep.	FM
GPS	110	typechecking	PVS	3,300	2	2
JS	70	typechecking; proof of system properties	PVS	500	1	0.5
HAC	20	typechecking; proof of coverage, disjointness	PVS	200	0.25	0.5
3E/O	70	state exploration; verification of sequencing properties	Murϕ	1,200	2	1.5

4.1 Technology Transfer

Formal methods were not in the critical path in any of the four studies, nor were they officially recognized as a component of the extant requirements analysis process. In the GPS and HAC studies, results of the formalization were actively solicited and served as input to the series of tightly scheduled inspections used to produce a final version of the requirements. In the 3E/O and JS studies, the formalization was not synchronized with ongoing inspections; rather, results were given to individual RAs who determined whether or not to include them in errata reports. Although Shuttle RAs have subsequently participated in training courses where they composed specifications and proofs for problems drawn from aerospace applications, they performed only a background role in formalizing the GPS, JS, 3E/O, and HAC requirements. However, it is worth noting that the GPS RAs were able to read and understand the PVS specifications without becoming PVS practitioners, corroborating a somewhat surprising facility with formal documentation similar to that reported by the AAMP5 project [Srivas and Miller 1995; 1996].

Shuttle flight software requirements analysis is fertile ground for formal methods because of issues such as those listed below, which represent acknowledged and longstanding deficiencies in the existing requirements analysis process. Although other methods could address these deficiencies to varying degrees, formal methods arguably provide the most versatile and powerful response through the use of mechanized tools that not only assure syntactic and semantic correctness, but also offer proof of the presence (or absence) of essential (or undesirable) system properties. The nature of the formal methods response is suggested in the remarks following each of the cited deficiencies.

—*There is no methodology to guide the analysis*: Formal methods offer rigorous modeling and analysis techniques that bring increased precision and error detection to the realm of requirements.

—*There are no completion criteria*: Writing formal specifications and conducting proofs are deliberate acts which are readily correlated with meaningful completion criteria.

—*There is no structured way for RAs to document the results of their analysis*: Formal specifications are tangible products that can be maintained and consulted as analysis and development proceed. When provided as outputs of the analysis process, formalized requirements can be used as evidence of thoroughness and coverage, as definitive explanations of how requirements and subsequent Change Requests achieve their objectives, and as permanent artifacts useful for answering future questions and addressing future changes.

The technology transfer reflected in the four studies varies, largely reflecting differences in project buy-in. Predictably, factors influencing the productivity of formal methods in the Shuttle requirements analysis environment are cultural as well as technical. Culturally, the pivotal factor was the presence of at least one NASA participant willing to champion the use of formal methods on the project. Technically, the most important factor was the ability of the formal methods practitioner(s) to provide meaningful results in a timely fashion. In this regard, it is important to recognize that the nature of results obtainable with a given formal methods approach typically differs as a function of the maturity of the requirements. Analysis of requirements for stable and legacy software generally yields fewer, but more substantive and subtle, issues than that for new designs or for proposed modifications and extensions, as illustrated by the results of the JS versus the GPS studies.

Furthermore, the notion of "meaningful result" depends on the context of the project. In the 3E/O study, several of the issues we found most compelling were not of interest to the RA. For example, we discovered a suboptimal 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 computation cycles, the anomaly was not considered important.

Clearly, the challenge of technology transfer is to develop effective insertion strategies that will ultimately produce in-house expertise. In the interim, an effective strategy is to team experienced formal methods practitioners with motivated application-area experts and to tackle feasible portions of real applications. In hardware engineering, a niche field of specialized formal methods has generated phenomenal interest through effective use of model checking and similar techniques. A challenge to the software engineering community is to develop tools similarly well suited to accelerate technology uptake, especially front-end tools supporting use of formal methods from the inception of requirement specification through the analysis of proposed modifications and extensions.

4.2 Legacy Applications

Space Shuttle flight software constitutes a prime example of a critical legacy system. Formalization of existing subsystems, such as JS, and proposed modifications and extensions, such as 3E/O and GPS, reveal useful, but previously undocumented, artifacts of existing requirements and requirements methodology. For example, the FSSR-style requirements used in the Shuttle program lack an explicit notion of state variable; the existence of state variables must be inferred strictly from context, typically by noting when local variables appear to be persistent. Explicitly modeling the state variables yielded a clearer and more precise statement of GPS requirements. As noted earlier, formalization of the JS algorithm was similarly revealing, exposing an algorithmic uniformity across Vernier and Alt modes that was previously undocumented, but which contributed to a considerably simpler and more readily understood statement of the basic JS function. In this regard, the utility of formal methods for legacy systems lies in providing an explicit, concise, and unambiguous statement of the system's requirements, thereby serving as a basis for communicating, maintaining, and modifying the underlying functionality of the system.

Legacy systems typically exhibit varying stages of maturity, as subsystems are replaced, modified, or extended. With a growing population of legacy systems requiring reengineering, there is a correspondingly large need for effective modeling and analysis tools. One of the more interesting results of the case studies described here is the ability of a single formal methods technique to contribute significantly to the existing requirements analysis process at various life cycle phases, as illustrated by the results of formalization at different depths of analysis in the JS, GPS, and HAC studies. These studies also illustrate the utility of focusing on algorithms and essential properties of systems, rather than the process by which such systems are developed or the details of their implementation. The emphasis on artifact rather than on process differs from that found in many software engineering techniques and offers an effective approach to the analysis of subsystems and system enhancements, and to bridging the gap between successive versions of long-lived systems. The challenge remains to define particularly effective formal methods strategies for legacy systems.

4.3 Rapid Formalization

Although it is generally agreed that formal methods are most productive early in the life cycle, there is some question about the ability of formal methods to keep up with software development, especially when theorem proving or proof checking is involved. The GPS study provides largely unanticipated, but useful, data on this question. GPS was undertaken with the understanding that the application of formal methods would be productive only if it could keep pace with the existing requirements analysis process. As a result, the GPS study opted for a rapid formalization strategy in which analysis was postponed to allow the formalization to proceed in lockstep with successive modifications.

Initially, rapid formalization was viewed with skepticism and adopted only out of necessity. However, as the GPS study progressed, it became clear that the strategy was not only viable, but very effective. Errors were detected quickly and fed back to the requirements authors by the appropriate inspection dates without herculean effort. The formalization was particularly useful for debugging interfaces. There are several advantages to this approach, not the least of which are the implications for technology transfer. The GPS study yielded significant results simply through formalization and a very modest level of analysis that ensured the syntactic and semantic (type) correctness of the specification. This approach would be even more effective in an environment in which requirements are developed and documented in formal notation.

Rapid formalization also poses challenges, including the need to provide formal notations suited to fast-paced, lightweight analysis. There are currently many lightweight tools emerging for specialized formal methods applications, including tools for analyzing tabular formats and model checkers for analyzing finite-state properties. The approach taken in this study was to use one tool (PVS) across a range of maturity levels, for both rapid formalization involving modest levels of analysis and in-depth studies exploiting deductive theorem proving. The higher-order logic features and mechanical theorem-proving capability of PVS were not exploited extensively in the GPS application, and a less expressive language, such as one based on first-order logic, could have been used with only a small loss of elegance. The versatility of formal methods tools that offer a range of analysis strategies including model checking and fully general theorem proving arguably provides a more productive environment even for rapid formalization applications. Nevertheless, the relative merits of generalized versus specialized systems for rapid formalization remains an open question.

4.4 Problem Domain Modeling

The system context for a module or subsystem often imposes implicit constraints on the variables and objects of interest, thereby circumscribing the space of possible variable assignments. Formalizing these relationships as data invariants is potentially very useful. In the HAC study, for example, variables are constrained as a function of the current flight mode. Once identified, these constraints were exploited to yield a fully automatic analysis of the tabular requirements. In the theorem-proving environment in which the HAC requirements were analyzed, automation hinged on formalizing data invariants that were sufficiently rich to make the analysis no more conservative than necessary, i.e., "false alarms" did not arise.

In the moderately sized 3E/O study, the complexity of the problem domain, the exclusive focus on sequencing properties, and the decision to use a fully automatic state exploration tool led to a modeling strategy that exploited variables over qualitative ranges and boolean values, and explicitly ignored the inherently complex relationships among the variables that represented the Shuttle's physical environment (as discussed in Section 3). Although this simplified domain model is adequate for analyzing sequencing properties, it clearly has limited utility for domain-sensitive properties of the 3E/O algorithm.

The HAC and 3E/O studies illustrate very different aspects of problem domain modeling. Nevertheless, both studies suggest the utility of extensive domain models. Given the criticality of Shuttle flight software, as well as its longevity, development of reasonably accurate logical models of the Shuttle's physical environment could be a valuable and cost-effective tool for Shuttle requirements analysis. Roughly analogous to the continuous-domain models currently used for Shuttle simulation, these logical models could provide a symbolic execution environment for analyzing requirement specifications, and for running prototype test suites against the requirements relatively early in the development cycle. Models of the data space would provide a long-term asset useful for rapid formalization and lightweight analysis, as well as for in-depth analysis of existing or proposed algorithms or systems. Data shared across subsystems must be well understood to be used correctly. Formalizing the nature of the shared data space is an effective way to capture that understanding.

4.5 Tailoring Techniques to Applications

The diversity of the four case studies and their various styles of formalization illustrate the principle that formal methods gain considerable effectiveness when they are judiciously tailored to the application at hand. Judicious tailoring goes beyond mere selective application, encompassing the underlying formalization strategy and requirements process. Although careful customization would undoubtedly benefit most, if not all, software engineering methods, judicious tailoring is arguably even more productive in the case of formal methods, because of the significant gap between the abstractness and theoretical power of mathematical logic on the one hand, and the concrete world of real systems on the other. As a result, tailoring can sharpen the focus and reduce the scope of formal models.

In the Shuttle case studies described in this article, formal methods techniques and tools were tailored with respect to several factors, including problem domain, maturity of the (sub)system, life cycle phase, interfaces to other subsystems, and concerns of the requirements analysts. Aside from characteristics of the problem domain, system maturity was probably the most influential factor. This is most forcefully illustrated in the two variants of the functional approach described here: the more abstract approach taken in the JS application versus the more concrete strategy used in the GPS study.

Many research teams produce languages and tools under the assumption that practitioners will use them as they do—an unwise assumption at best; tailoring is both necessary and desirable. Merely providing theoretical power is not the answer. Nor is assuming that every formal methods user has the wherewithal to extensively tune a formal methods technique or system before using it. The challenge for the research community is to develop languages, methods, and tools that acknowledge the need to customize, but to do so quickly, conveniently, and productively.

5. CONCLUSION

We briefly survey related work, summarize the main points and their implications, and review the contribution of the four case studies.

5.1 Related Work

In the approximately five years since the initial LaRC case studies, there has been a significant increase in the number of published accounts of formal methods applications, including several surveys (for example, Craigen et al. [1993a; 1993b] and Clarke et al. [1996b]) and collections (for example, Hinchey and Bowen [1995; 1997] and Kropf [1997]). The broad outlines of our experiences in formalization and analysis of critical systems are similar to those recounted in the majority of published case studies, that is, the process of formalization—whether manual (for example, Ladkin [1995] and Parnas [1995]) or mechanized (for example, Jacky [1995] and Brookes et al. [1996])—clarifies requirements issues and identifies anomalies and lacunae. Proof checking and model checking expose further anomalies and confer a significantly greater level of confidence in the established properties and behaviors [Brock et al. 1996; Clarke et al. 1996a; Curzon 1994; Heimdahl and Leveson 1996; Moore et al. 1996]. The GSP, JS, and HAC studies, viewed as a three-part study, differ from other published accounts in their focus on exploring the use of a single, general-purpose system (PVS) on requirements representing a range of maturity levels.

The case of 3E/O is somewhat different. 3E/O was one of the first published accounts of finite-state verification of a realistic example outside the hardware domain (typically communication and cache coherence protocols). Although there were no directly comparable studies when the 3E/O work was done, there have been subsequent studies that involve finite-state verification of software requirements for aerospace systems. Many of these studies describe the use of finite-state verification to check the completeness and consistency of tabular specifications [Heimdahl and Leveson 1996], and others exploit previous work in communication protocols to analyze requirements for spacecraft controllers [Schneider et al. 1998]. Relatively few use finite-state verification in applications comparable to 3E/O. A notable exception, Sreemani and Atlee [1996] focus on mechanizing the translation of tabular (SCR) specifications into a representation acceptable by a finite-state verifier (SMV) and exploring properties of the mode-switching behavior specified in the A-7E (aircraft) software requirements.

5.2 Summary of Observations

The Shuttle flight software studies described in this article yield several observations of potential interest to the software engineering community.

The observations fall roughly into two categories: factors affecting the utility and uptake of formal methods (*) and generalizations on the results of the four case studies (§).

(*) Motivated application experts are crucial to the successful uptake of formal methods. Such individuals readily learn to read formal specifications. It remains to be seen whether or not this facility will extend to the use of theorem provers and proof checkers.

(*) Applying formal methods "right out of the box" is difficult. Tailoring the methods to the application at hand is both necessary and desirable. Devising a customized approach and doing early prototyping enhance project robustness and are a wise prelude to full-scale analysis.

(*) A corollary of the need to customize is the need to integrate. Although the case studies described here each use a single formal-methods technique, judicious combinations of techniques (for example, model checking and proof checking [Havelund and Shankar 1996] and formal specification and execution [Agerholm and Larsen 1997][10]) offer increased insight and productivity.

(*) Depth of analysis versus timely results is an important trade-off that must be addressed, especially in the context of technology transfer. The utility of rapid formalization in a real project setting should not be overlooked.

(*) Formal methods may be used in a variety of roles, for example, to analyze logical and behavioral properties, to clarify and document requirements, to facilitate communication, and to satisfy standards and provide assurance or certification data. These roles are not mutually exclusive, but reflect a difference in focus that should be factored into the specification and analysis.

(*) Problem domain modeling is valuable for sharpening the analyses performed and reducing the incidence of false alarms.

(*) The development of general-purpose libraries for mathematical and application-specific domains is essential to the productive use and successful uptake of formal methods.

(§) Formal methods have been successfully used to analyze requirements of moderate-to-high criticality, resulting in the detection of issues of varying severity, including potentially serious, but previously undetected, errors in both mature and immature requirements.

(§) Formal methods can confer benefits regardless of how extensively they are adopted and applied; the process of formalization exposes significant anomalies, even without subsequent proof activity.

[10]Available via http://atb-www.larc.nasa.gov/Lfm97/proceedings/

(§) Formal methods are effective throughout the life cycle. Although there are obvious advantages to getting in on the ground floor, formal methods also confer benefits on legacy systems.

(§) Formal methods provide an effective complement to conventional requirements analysis, providing earlier exposure of potential issues, and dovetailing nicely with existing inspection-, simulation-, and test-based processes.

5.3 Concluding Remarks

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 four case studies described here has been the development of reusable strategies, and a clarification of the utility of formal methods techniques across a broad spectrum of maturity levels.

APPENDIX

A. SUMMARY OF PVS LANGUAGE FEATURES

PVS language features used in the article are summarized below. More detailed information on PVS may be found in Owre et al. [1995] and in the PVS manuals [Owre et al. 1993a; 1993b; 1993c].

PVS specifications are organized around the concept of *theories*. Each theory is composed of a sequence of declarations or definitions of various kinds. Definitions from other theories are not visible unless explicitly imported. Theories may be parameterized, allowing reuse. For example, the JS theory `finite_sequences` in Figure 3 is parameterized with respect to a positive natural number and a type. The PVS "prelude" provides a large collection of "built-in" theories.

PVS allows the usual range of scalar types to model various quantities. Numeric types include natural numbers (`nat`), integers (`int`), rationals (`rat`), and reals (`real`). Nonnumeric types include booleans (`bool`) and enumerations (`{C1, C2,...}`). Subranges and subtyping mechanisms allow derivative types to be introduced. Types in PVS may be modeled as sets, with subtypes having the usual set-theoretic meaning. For example, the subtypes `below` and `upto` are defined in the PVS prelude as

```
below(m): TYPE = {s: nat | s < m}
upto(m):  TYPE = {s: nat | s <= m}
```

for some natural number, `m`.

Uninterpreted types are those for which only minimal declarations are provided. The only assumption made on an uninterpreted type `T` is that it is nonempty and disjoint from all other types (except for subtypes of `T`).

Axioms may be used to provide further constraints on an uninterpreted type.

Structured data types or record types are introduced via declarations of the form

```
record_type: TYPE = [# v1: type_1, v2: type_2,... #]
```

The first component of record `R` may be accessed using the notation `v1(R)`. A record value constructed from individual component values may be synthesized as follows:

```
(# v1 := <expr 1>, v2 := <expr 2>,... #)
```

Tuples are similar to records. Tuple types are introduced by writing the form `[type_1, type_2,...]`. Tuple values may be constructed analogously using the form `(<expr 1>,<expr 2>,...)`.

Function types constitute an important class of types in PVS. A declaration of the form

```
fun_type: TYPE = [type_1 → type_2]
```

defines a (higher-order) type whose values are functions from `type_1` to `type_2`. The form `pred[type]` is shorthand for the predicate (that is, a boolean-valued function) `[type → bool]`. Function values may be constructed using lambda expressions:

```
(LAMBDA x, y: <expression of x, y>)
```

A named function is defined quite simply by the notation

```
fn (arg_1, arg_2,...): result_type = <expression>
```

Each of the variables `arg_i` must have been declared of some type previously or given a local type declaration.

Function, tuple, and record types may be "dependent," i.e., some of the component *types* may be constrained by the *values* of components found earlier in the type declaration. For example, in the dependent type declaration for the record type `date`,

```
date: TYPE = [# month: {m: posint | m <= 12},
                day:   {d: posint | d <= max_day(month)}
                #]
```

the type of component `day` is restricted according to the value of the corresponding month.

A LET expression allows the introduction of bound variable names to refer to subexpressions.

```
LET v1 = <expr 1>, v2 = <expr 2>,...
IN <expression involving v1, v2,...>
```

Each of the variables serves as a shorthand notation used in the final expression. The meaning is the same as if each of the subexpressions were substituted for its corresponding variable.

Finally, PVS provides a tabular notation for expressing conditional expressions in a naturally readable form. For example, an algebraic sign function could be defined as follows:

```
sign(x): signs =
  TABLE %--------------------------%
        |[ x < 0 | x = 0 | x > 0 ] |
        %--------------------------%
        |   -1   |   0   |   1     |
        %--------------------------%
  ENDTABLE
```

The “%” character begins a comment in PVS.

B. A BRIEF OVERVIEW OF MURϕ

The Murϕ description language[11] is briefly summarized below, followed by an equally brief overview of the Murϕ finite-state verifier. The discussion is limited to features relevant to the 3E/O example used in the body of the article. Further details on the language and the verifier may be found in Dill et al. [1992], Ip and Dill [1993], and Dill [1996].

The Murϕ language was inspired by Misra and Chandy’s Unity formalism [Dill 1996, p. 390] and provides high-level descriptions for both asynchronous and synchronous finite-state concurrent systems. High-level features such as those found in Pascal, Modula, and C include user-defined data types, procedures, and parameterized descriptions. A Murϕ program description consists of a collection of declarations, transition rules, invariants, and one or more start rules.

The numeric types provided in Murϕ are limited to (finite) integer subranges. Nonnumeric types include booleans and enumerations. Booleans are defined as predefined enumerations using the (predefined) constants “true” and “false.” For example, the following Murϕ fragment illustrates the declaration of an integer subrange and an enumeration type.

```
Type
  timer: 0..10;
  velocity: enum{GTR_vi_3eo_max, GTR_vi_3eo_min,
            LEQ_vi_3eo_min};
```

Compound types, including arrays and records (both) of simple or compound types, as well as procedures and functions (declared at the top level of the program), are also supported.

A transition rule is a guarded command consisting of a condition and a set of actions, that is, a boolean expression characterizing the states in which the corresponding actions (statements) may be executed. If no condition is specified, the condition is assumed to be “true,” i.e., the rule is

[11]Murϕ may be viewed as an “exhaustive tester”—hence the name, which derives from one of Murphy’s Laws: “The bug is always in the case you didn’t test.”

always enabled. Conditions may not have side-effects; only the statements in the body of a rule may modify the values of variables. Although the body may be arbitrarily complex (for example, containing loops and conditionals) it is always executed atomically; no other rule may modify variables or otherwise interfere while a rule is executing. Rules may declare local variables, types, and constants, but these local entities are not part of the state. The syntax of a simple rule is defined as follows, where all components are optional (as indicated by braces):

```
rule [<rulename>]
     [<condition> ==>]
     [{<decls>} begin]
     [<body>]
endrule
```

Start rules (referred to as "start states") are special rules executed only at the beginning of an execution run. A program must have at least one start state, and a start state must initialize (or declare "undefined") all global variables.

Murϕ invariants are also rules, although invariants may be expressed most efficiently in an abbreviated format. The form

```
invariant [<invariantname>]<expr>
```

is syntactic sugar for

```
rule [<invariantname>]
    !<expr> ==> Error
                 "Invariant violated: <invariantname>"
endrule
```

The `Error` statement generates a runtime error. Although the two forms of the invariant are equivalent, the "invariant form" is usually more efficient because the Murϕ compiler can exploit the restricted properties of an expression explicitly identified as an invariant. Expressions appearing in an invariant may not have side-effects.

A Murϕ state consists of the current values of all of the global variables of the description. Execution of a Murϕ program consists of any sequence of states that can be generated from a start rule by repeated selection and execution of the rules. Murϕ is nondeterministic; there are usually many executions, each defined by a unique sequence of selected rules. Executing a rule typically changes the state, since rules typically modify global variables. Various conditions, including those specified by invariants and assert statements, are evaluated in each newly generated state. If the result of this evaluation is "false," verification halts, and a trace of the execution from the start state to the "error" state is printed.

Murϕ is an explict state verifier. Although BDD-based verifiers can be very efficient, it has been claimed that explicit state verifiers are more readily exploited by new users, and provide more predictability and consistency because performance is more directly related to the number of states [Dill 1996, p. 391].

ACKNOWLEDGMENTS

The authors are grateful for the assistance of the requirements analysts and other staff at Lockheed Martin Space Information Systems: Larry Roberts, Mike Beims, Ron Avery, and, in earlier phases of this work, David Hamilton and Dan Bowman. They all patiently answered questions and served up voluminous amounts of documentation. We would also like to thank John Rushby (SRI), Natarajan Shankar (SRI), Rick Butler (LaRC), Michael Holloway (LaRC), John Kelly (JPL), and the anonymous reviewers who provided detailed and constructive comments on the article.

REFERENCES

AGERHOLM, S. AND LARSEN, P. G. 1997. Modeling and validating SAFER in VDM-SL. In *Proceedings of the 4th NASA Langley Formal Methods Workshop (LFM '97)* (Hampton, VA, Sept.), C. M. Holloway and K. J. Hayhurst, Eds. NASA Langley Research Center, Hampton, VA, 51–64. Available via http://atb-www.larc.nasa.gov/lfm97/proceedings.

ALUR, R. AND HENZINGER, T. A., EDS. 1996. *Computer-Aided Verification (CAV '96)* (New Brunswick, NJ, July–Aug.). Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, New York, NY.

BROCK, B., KAUFMANN, M., AND MOORE, J. S. 1996. ACL2 theorems about commercial microprocessors. In *Formal Methods in Computer-Aided Design (FMCAD '96)* (Palo Alto, CA, Nov.), M. Srivas and A. Camilleri, Eds. Lecture Notes in Computer Science, vol. 1166. Springer-Verlag, New York, NY, 275–293.

BROOKES, T., FITZGERALD, J., AND LARSEN, P. 1996. Formal and informal specification of a secure system component: Final results in a comparative study. In *Formal Methods Europe (FME '96)* (Oxford, England, Mar.). Lecture Notes in Computer Science, vol. 1051. Springer-Verlag, New York, NY, 214–227.

BUTLER, R. W., CALDWELL, J. L., CARREÑO, V. A., HOLLOWAY, C. M., MINER, P. S., AND DI VITO, B. L. 1995. NASA Langley's research and technology transfer program in formal methods. In *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS '95)* (Gaithersburg, MD, June 1995), 135–149.

CLARKE, E. M., GERMAN, S. M., AND ZHAO, X. 1996a. Verifying the SRT division algorithm using theorem proving techniques. In *Computer-Aided Verification (CAV '96)* (New Brunswick, NJ, July/Aug.), R. Alur and T. A. Henzinger, Eds. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, New York, NY, 111–122.

CLARKE, E., WING, J. M., ALUR, R., CLEAVELAND, R., DILL, D., EMERSON, A., GARLAND, S., GERMAN, S., GUTTAG, J., HALL, A., HENZINGER, T., HOLZMANN, G., JONES, C., KURSHAN, R., LEVESON, N., MCMILLAN, K., MOORE, J., PELED, D., PNUELI, A., RUSHBY, J., SHANKAR, N., SIFAKIS, J., SISTLA, P., STEFFEN, B., WOLPER, P., WOODCOCK, J., AND ZAVE, P. 1996b. Formal methods: State of the art and future directions. *ACM Comput. Surv. 28*, 4 (Dec.), 626–643.

CRAIGEN, D., GERHART, S., AND RALSTON, T. 1993a. An international survey of industrial applications of formal methods. Vol. 1, Purpose, approach, analysis and conclusions. Tech. Rep. NIST GCR 93/626. National Institute of Standards and Technology, Gaithersburg, MD.

CRAIGEN, D., GERHART, S., AND RALSTON, T. 1993b. An international survey of industrial applications of formal methods. Vol. 2, Case studies. Tech. Rep. NIST GCR 93/626. National Institute of Standards and Technology, Gaithersburg, MD.

CROW, J. 1995. Finite-state analysis of space shuttle contingency guidance requirements. Tech. Rep. SRI-CSL-95-17, Computer Science Laboratory, SRI International, Menlo Park, CA. Also available as Contractor Rep. 4741, NASA Langley Research Center, Hampton, VA, May 1996.

CURZON, P. 1994. The formal verification of the Fairisle ATM switching element: An overview. Tech. Rep. 328, Univ. of Cambridge Computer Laboratory, Cambridge, UK. Available via http://www.cl.cam.ac.uk/users/pc/el1tr94.html

DI VITO, B. L. 1996. Formalizing new navigation requirements for NASA's space shuttle. In *Formal Methods Europe (FME '96)* (Oxford, England, Mar.). Lecture Notes in Computer Science, vol. 1051. Springer-Verlag, New York, NY, 160–178.

DI VITO, B. L. AND ROBERTS, L. W. 1996. Using formal methods to assist in the requirements analysis of the space shuttle GPS change request. Contractor Rep. 4752, NASA Langley Research Center, Hampton, VA.

DILL, D. L. 1996. The Murϕ verification system. In *Computer-Aided Verification (CAV '96)* (New Brunswick, NJ, July/Aug.), R. Alur and T. A. Henzinger, Eds. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, New York, NY, 390–393.

DILL, D. L., DREXLER, A. J., HU, A. J., AND YANG, C. H. 1992. Protocol verification as a hardware design aid. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors* (Cambridge, MA, Oct. 11–14). IEEE Computer Society, Washington, DC, 522–525.

DUTERTRE, B. AND STAVRIDOU, V. 1997. Formal requirements analysis of an avionics control system. *IEEE Trans. Softw. Eng. 23*, 5 (May), 267–278.

HAMILTON, D., COVINGTON, R., AND KELLY, J. 1995a. Experiences in applying formal methods to the analysis of software and system requirements. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT '95)* (Boca Raton, FL). IEEE Computer Society, Washington, DC, 30–43.

HAMILTON, D., COVINGTON, R., AND LEE, A. 1995b. Experience report on requirements reliability engineering using formal methods. In *Proceedings of the International Conference on Software Reliability Engineering (ISSRE '95)* (Toulouse, France). IEEE Computer Society, Washington, DC.

HAVELUND, K. AND SHANKAR, N. 1996. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe (FME '96)* (Oxford, England, Mar.). Lecture Notes in Computer Science, vol. 1051. Springer-Verlag, New York, NY, 662–681.

HEIMDAHL, M. P. E. AND LEVESON, N. G. 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. Softw. Eng. 22*, 6 (June), 363–377.

HENINGER, K. L. 1980. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng. SE-6*, 1 (Jan.), 2–13.

HINCHEY, M. G. AND BOWEN, J. P., EDS. 1995. *Applications of Formal Methods*. Prentice-Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., Hertfordshire, UK.

HINCHEY, M. G. AND BOWEN, J. P., EDS. 1997. *Industrial Strength Formal Methods*. Academic Press, Inc., Orlando, FL.

IP, C. N. AND DILL, D. L. 1993. Better verification through symmetry. In *Proceedings of the 11th Conference on Computer Hardware Description Languages and Their Applications (CHDL '93)*. International Federation for Information Processing, Laxenburg, Austria, 87–100.

JACKY, J. 1995. Specifying a safety-critical control system in Z. *IEEE Trans. Softw. Eng. 21*, 2 (Feb.), 99–106.

KROPF, T., ED. 1997. *Formal Hardware Verification: Methods and Systems in Comparison*. Springer Lecture Notes in Computer Science, vol. 1287. Springer-Verlag, New York, NY.

LADKIN, P. B. 1995. Analysis of a technical description of the airbus A320 braking system. *High Integrity Syst. 1*, 4, 331–349.

LUTZ, R. R. AND AMPO, Y. 1994. Experience report: Using formal methods for requirements analysis of critical spacecraft software. In *Proceedings of the 19th Annual Software Engineering Workshop*. NASA GSFC, Greenbelt, MD, 231–248.

MOORE, J. S., LYNCH, T., AND KAUFMANN, M. 1996. A mechanically checked proof of the correctness of the AMD5_k86 floating-point division algorithm. Tech. Rep. Available via http://devil.ece.utexas.edu:80/~lynch/divide/divide_paper.ps

NASA. 1993. Formal methods demonstration project for space applications—Phase I case study: Space Shuttle orbit DAP jet select. NASA Code Q Final Rep (Unnumbered), National Aeronautics and Space Administration, Washington, DC. Jet Select requirements specification appears as Appendix B.

NRCC. 1993. *An Assessment of Space Shuttle Flight Software Development Practices*. National Academy Press, Washington, DC.

OWRE, S., RUSHBY, J., SHANKAR, N., AND VON HENKE, F. 1995. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. Softw. Eng. 21*, 2 (Feb.), 107–125.

OWRE, S., SHANKAR, N., AND RUSHBY, J. M. 1993a. *User Guide for the PVS Specification and Verification System*. Vol. 1, *Language Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA.

OWRE, S., SHANKAR, N., AND RUSHBY, J. M. 1993b. *User Guide for the PVS Specification and Verification System*. Vol. 2, *System Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA.

OWRE, S., SHANKAR, N., AND RUSHBY, J. M. 1993c. *User Guide for the PVS Specification and Verification System*. Vol. 3, *Prover Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA.

PARNAS, D. L. 1995. Using mathematical models in the inspection of critical software. In *Applications of Formal Methods*, M. G. Hinchey and J. P. Bowen, Eds. Prentice-Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 17–31.

ROBERTS, L. W. AND BEIMS, M. 1996. Using formal methods to assist in the requirements analysis of the space shuttle HAC change request (CR90960E). JSC Tech. Rep. Loral Space Information Systems, Houston, TX.

RUSHBY, J. 1996. Automated deduction and formal methods. In *Computer-Aided Verification (CAV '96)* (New Brunswick, NJ, July/Aug.), R. Alur and T. A. Henzinger, Eds. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, New York, NY, 169–183.

SCHNEIDER, F., EASTERBROOK, S., CALLAHAN, J., AND HOLZMANN, G. 1998. Validating requirements for fault tolerant systems using model checking. In *Proceedings of the 3rd International Conference on Requirements Engineering (ICRE '98)*. IEEE Computer Society, Washington, DC.

SREEMANI, T. AND ATLEE, J. M. 1996. Feasibility of model checking software requirements. In *Proceedings of the 11th Annual Conference on Computer Assurance* (Gaithersburg, MD, June). IEEE Computer Society, Washington, DC, 77–88.

SRIVAS, M. K. AND MILLER, S. P. 1995. Formal verification of the AAMP5 microprocessor. In *Applications of Formal Methods*, M. G. Hinchey and J. P. Bowen, Eds. Prentice-Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 125–180.

SRIVAS, M. K. AND MILLER, S. P. 1996. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods Syst. Des. 8*, 2, 153–188.

VAN SCHOUWEN, A. J. 1990. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Tech. Rep. 90-276, Dept. of Computing and Information Science. Queen's University, Kingston, Ontario, Canada.

Received: June 1997; revised: October 1997; accepted: November 1997