Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

High-Automation Proofs for Properties of Requirements Models

Ben L. Di Vito*

NASA Langley Research Center, MS 130, Hampton, VA 23681, USA; e-mail: `b.l.divito@larc.nasa.gov`

Received: date / Revised version: date

Abstract. We describe an approach and experimental results in the application of mechanized theorem proving to software requirements analysis. Serving as the test article was the embedded controller for SAFER, a backpack propulsion system used as a rescue device by NASA astronauts. SAFER requirements were previously formalized using PVS during a NASA pilot project in formal methods, details of which appear in a NASA guidebook. This paper focuses on the formulation and proof of properties for the SAFER requirements model.

To test the prospects for deductive requirements analysis, we used the PVS theorem prover to explore the upper limits of proof automation. A set of property classes was identified, with matching proof schemes later devised. After developing several PVS proof strategies (essentially prover macros), we obtained fully automatic proofs of 42 model properties. These results demonstrate how customized prover strategies can be used to automate moderate-complexity theorem proving for state machine models.

1 Introduction

Requirements modeling and analysis are promising areas for applied formal methods and technology transfer. Established requirements-based languages and tools, such as Statecharts/Statemate [6], SCR [7], and RSML [9], have been fruitfully applied in practice and continue to attract interest. New analysis methods are frequently reported, especially ones using lightweight techniques based on finite state exploration.

Less frequently mentioned, but no less important, are modeling and analysis methods based on general purpose logics. Deduction using such logics can validate a requirements model's accuracy or demonstrate the absence of undesirable system behavior. Although extensive proving capability exists for highly expressive languages, deductive analysis using powerful theorem provers is often considered less attractive than the special-purpose methods. In this paper we present experimental results showing that theorem proving can be applied effectively to requirements analysis.

Previous technology insertion efforts [3,20] convinced us that problem domain experts can easily learn to read formal specifications, and with a little more effort, can learn to write them. Gaining theorem proving skills, however, takes a much larger training investment. For prover-based requirements analysis to enter common practice, high levels of proof automation are needed, at least for properties that are mathematically shallow, i.e., those yielding to proofs of low to moderate complexity. Our results suggest that by expressing properties in a stylized manner, then adding semi-custom prover strategies, aggressive proof automation is indeed feasible.

Model checking techniques have been advocated as a more practical alternative to the full deductive power of theorem proving. Users have been willing to trade expressibility and theoretical power to gain a higher level of automation through model checking. Although these techniques are genuinely useful, we should not ignore the untapped potential of modern theorem provers. In fact, the results of this study suggest that provers have been underestimated and reasonable approaches for boosting their automation level in certain domains can be readily found. Moreover, the performance of such automated deduction compares favorably with that of model checking.

The SAFER example in this study has characteristics that would make it suitable for model checking analysis. On the other hand, attempting a similar but more challenging application, for example, one dominated by

* This work was performed while the author was with ViGYAN, Inc., Hampton, VA 23666, USA.

arithmetic relations or inductively defined data objects, would drive us out of the zone of feasibility for model checkers. Instead, our approach would be able to handle this richer class of problems while retaining a good deal of the automation described in this paper.

NASA's Simplified Aid for EVA Rescue (SAFER) [11] is a backpack propulsion system for free-floating astronauts, intended as a self-rescue device on Space Shuttle missions, as well as during Space Station construction and operation. The embedded controller for SAFER served as the test article for our study. SAFER requirements were previously formalized using PVS (Prototype Verification System) during a NASA pilot project in formal methods, details of which appear in the appendix to a NASA guidebook [15].

We used the PVS theorem prover to explore the limits of possible automation. PVS is an environment for formal specification and verification developed at SRI International's Computer Science Laboratory [16]. The system is freely available under license from SRI.

Our work was performed in the context of a broad program of applied formal methods activity at NASA's Langley Research Center (LaRC) [2]. Additional background and overview material on the use of formal methods in aerospace applications can be found in Rushby's formal methods handbooks [18,19], and in the previously mentioned guidebook volumes [14,15].

2 Overview of SAFER

SAFER is a small, self-contained, backpack propulsion system enabling free-flying mobility for a crew member engaged in extravehicular activity (EVA). SAFER is a single-string system designed for contingency use only. The description that follows draws heavily on excerpts from Appendix C of the guidebook [15], which in turn was based on the SAFER Operations Manual [13] and the Prime Item Development Specification [12]. The guidebook version of SAFER is a modified form of the real article, created to suit the pedagogical needs of a guidebook example. Although the SAFER application is modest in size, it nevertheless contains elements typical of control systems for aerospace vehicles.

2.1 Concept of Operation

SAFER fits around the Extravehicular Mobility Unit (EMU) primary life support subsystem backpack (Figure 1). SAFER uses 24 gaseous-nitrogen (GN_2) thrusters to achieve six degree-of-freedom maneuvering control. Operation is by a simple hand controller. Propulsion is available either on demand, i.e., in response to hand controller inputs, or through an automatic attitude hold (AAH) capability. Hand controller inputs can command either translations or rotations, while attitude hold is designed to bring and keep rotation rates close to zero.

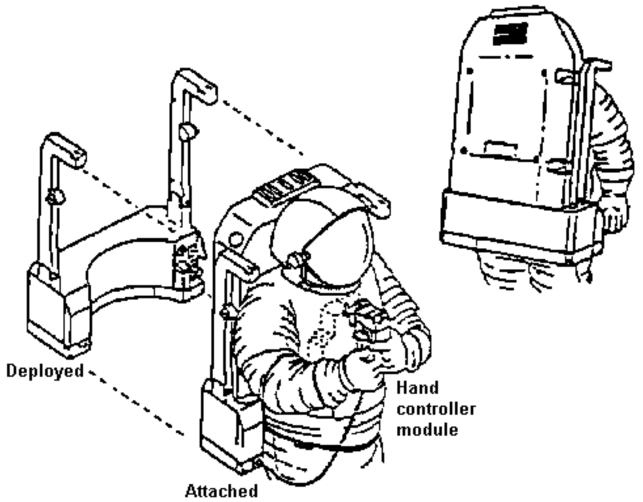

Fig. 1. SAFER use by an EVA crew member.

The SAFER flight unit consists of three assemblies: the backpack propulsion module, the hand controller module, and a replaceable battery pack. The propulsion module is the primary assembly, and its avionics box is the main component of interest for our study. Housed in the avionics box are the control electronics assembly, inertial reference unit, data recorder assembly, and power supply assembly.

The 24 thrusters are physically distributed around the backpack, with four thrusters pointing in each of the ±X, ±Y, and ±Z directions. When a thruster valve is commanded open, pressurized GN_2 is released and expanded through the thruster's conical nozzle to yield a propulsive force. SAFER can fire up to four thrusters at a time, providing motion with six degrees of freedom (±X, ±Y, ±Z, ±roll, ±pitch, and ±yaw).

A SAFER crew member controls the flight unit and monitors its status by means of the hand controller module (HCM) shown in Figure 2. The hand controller grip is mounted on the right side of the HCM with an integral push-button switch for initiating and terminating AAH mode. An acceleration command is generated by moving the grip from the center null position to mechanical hard-stops along the hand controller axes.

The avionics software is responsible for controlling the SAFER unit in response to crew member commands. Digital interfaces to SAFER components enable its microprocessor to achieve control. Several classes of inputs are monitored and accepted by the avionics software: hand controller switches, avionics sensors, and a thruster-on sensor. Similarly, there are several classes of outputs generated by the avionics software: hand controller displays, thruster actuation commands, and data recorder events.

Both rotational and translational accelerations are commanded by software within the maneuvering control subsystem. Rotations resulting from the AAH function

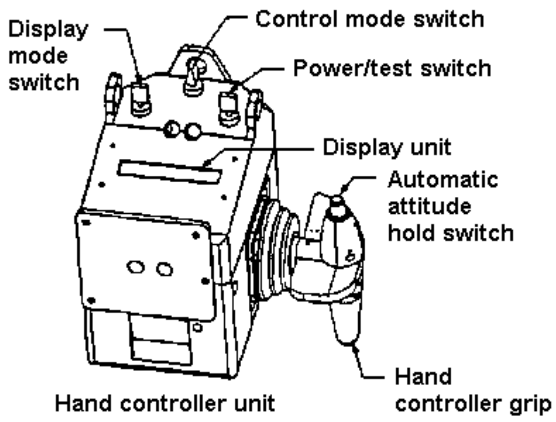

Fig. 2. SAFER hand controller module.

are invoked automatically in response to rotation rates sensed by the inertial reference unit. Special cases result from the interaction of the AAH function and explicitly commanded accelerations.

A SAFER crew member can initiate AAH at any time by depressing or "clicking" the pushbutton on the hand controller grip. When the button is double clicked (pressed twice within 0.5 sec), AAH is disabled for all three rotational axes. If AAH is active, and the crew member issues a rotational acceleration command about any axis, AAH is immediately disabled on that axis. When this occurs, the remaining axes remain in AAH.

Acceleration commands from the hand controller and from the AAH function are combined to create a single acceleration command. Thruster selection logic is provided to choose suitable thruster firings that achieve the commanded acceleration.

2.2 Formalization of SAFER Requirements

SAFER requirements originally were stated in conventional, natural language form, each cast as a "shall" statement. Several examples follow.

> The avionics software shall disable AAH on an axis if a crew member rotation command is issued for that axis while AAH is active.
>
> Any hand controller rotation command present at the time AAH is initiated shall subsequently be ignored until a return to the off condition is detected for that axis or until AAH is disabled.
>
> Hand controller rotation commands shall suppress any translation commands that are present, but AAH-generated rotation commands may coexist with translations.

Based on such requirements and additional descriptive material, a formal model was developed.

The PVS formalization of the SAFER system appears in the guidebook. A subset of the SAFER requirements was chosen for modeling that emphasizes the main functional requirements and omits support functions such as the ground checkout features. Even within the flight operation requirements some functions were represented only in abstract form.

Eleven PVS theories formalize the requirements:

- `avionics_types`
- `hand_controller_module`
- `propulsion_module`
- `inertial_reference_unit`
- `automatic_attitude_hold`
- `thruster_selection`
- `power_supply`
- `data_recorder`
- `self_test`
- `HCM_display`
- `avionics_model`

The formal model uses a state machine representation for the main control function. Figure 3 shows a sketch of the top level PVS representation, including controller interface types and main transition function. The controller is assumed to run continuously, executing its control algorithms once per frame, whose duration is set at 5 msec. In each frame, sensors, switches and the hand grip controller are sampled to provide the inputs to the control functions for that frame. Based on these inputs and the controller's state variables, actuator commands and crew display outputs are generated, as well as the controller's state for the next frame.

Figure 3 shows several data types, the components of which appear in the formulas presented in Section 5.1. PVS uses the notation `[# v1: type1, ... #]` to define a record structure, and `(# v1 := expr1, ... #)` to construct a record value. Components of such a record are accessed using functional notation: `v1(R)`.[1] For example, the value of state component `clock` after SAFER undergoes a state transition is given by the PVS expression `clock(state(SAFER_control(in, st)))`.

Figure 4 depicts a state diagram of the AAH engagement protocol, which indicates how SAFER reacts to the position of the AAH pushbutton and several other conditions. Figure 5 contains the PVS incarnation of this diagram. The AAH feature plays a prominent role in the formulation of model properties.

3 Requirements Analysis

We now turn our attention to how deductively based requirements analysis can be conducted. This type of analysis is not meant to be complete. Other forms of scrutiny have been and will continue to be used in examining

[1] PVS has added the notation `` R`v1 `` as a synonym for `v1(R)`.

```
avionics_inputs: TYPE = [#
        HCM_switches:  HCM_switch_positions,
        grip_command:  hand_grip_position,
          . . .  #]

avionics_outputs: TYPE = [#
        HCM_displays:   HCM_display_set,
        prop_actuators: actuator_commands,
        data_recorder:  data_recorder_packet #]

avionics_state:  TYPE = [#
        msg_display:  HCM_display_state,
        AAH_state:    AAH_state,
        clock:        nat,
        self_test:    self_test_state #]

avionics_result: TYPE = [#
        output: avionics_outputs,
        state:  avionics_state #]

SAFER_control (avionics_inputs: avionics_inputs,
               avionics_state:  avionics_state)
              : avionics_result =

      (# output := (# HCM_displays   :=  . . . ,
                      prop_actuators :=  . . . ,
                      data_recorder  :=  . . . #),
         state  := (# msg_display :=  . . . ,
                      AAH_state   :=  . . . ,
                      clock       :=  . . . ,
                      self_test   :=  . . . #)
       #)
```

Fig. 3. Top level state machine rendered in PVS.

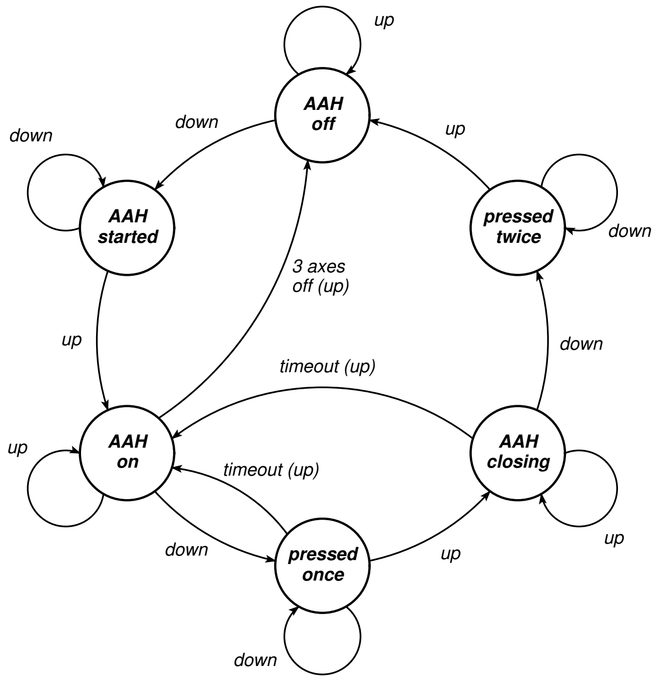

Fig. 4. Automatic attitude hold state diagram.

```
button_transition(state:   AAH_engage_state,
                  button:  AAH_control_button,
                  active:  rot_predicate,
                  clock:   nat,
                  timeout: nat): AAH_engage_state =
 TABLE
       state       ,             button
                   %----------------------------%
                   |[ button_down   | button_up ]|
   %--------------------------------------------%
    | AAH_off       |  AAH_started   | AAH_off    ||
    | AAH_started   |  AAH_started   | AAH_on     ||
    | AAH_on        |  pressed_once  | state_A    ||
    | pressed_once  |  pressed_once  | state_B    ||
    | AAH_closing   |  pressed_twice | state_B    ||
    | pressed_twice |  pressed_twice | AAH_off    ||
   %--------------------------------------------%
 ENDTABLE
   WHERE state_A =
           IF all_axes_off(active)
              THEN AAH_off ELSE AAH_on ENDIF,
         state_B =
           IF clock >= timeout
              THEN AAH_on ELSE AAH_closing ENDIF
```

Fig. 5. AAH state diagram expressed in PVS.

software requirements. It is incumbent on requirements analysts to determine how and to what extent these deductive techniques can be integrated into their existing processes.

3.1 Requirements Model Properties

Given a formal requirements model, several types of rigorous analysis are possible. Precisely stated models have consequences that can themselves be precisely stated. By formulating traits of expected system behavior, it is possible to analyze requirements, in a limited way, for well-formedness and compliance with desired characteristics. Once expressed in this manner, it is further possible to formally prove that the properties follow from the definitions given in the requirements model.

There are many possible aspects of SAFER behavior one might wish to investigate or verify. Several examples are suggested below.

- When AAH is inactive and no hand grip commands are present, there should be no thruster firings.
- SAFER should never fire more than four thrusters simultaneously.
- No two selected thrusters should oppose each other, i.e., have cancelling thrust with respect to the center of mass.
- Once AAH is turned off for a rotational axis it remains off until a new AAH cycle is initiated.

Properties such as these identify behavior that designers and analysts insist a system must have if it is to satisfy their expectations. These properties must logically follow as consequences of the definitions contained in the system model. Formal statements of such properties are sometimes called challenge theorems or putative theorems. If a mistake is made in deriving the requirements or formalizing them, attempts to express and prove these conjectures will help detect the error. This approach constitutes a rigorous method of analyzing requirements. It becomes possible to definitively answer questions about system behavior, reducing the chances of error from miscalculation, interpretation, or engineering judgement.

3.2 Formulating Properties

By leaving the nature of properties open-ended, there is little we can do to enhance proof automation. On the other hand, by identifying common classes of properties and restricting ourselves to instances of these classes, we gain opportunities to develop reusable proof strategies. Moreover, such reuse can extend beyond immediate project needs. It is reasonable to expect, for example, that other applications relying on state machine models could use the property classes developed in this study.

General experience with state models suggests two broad types of properties: those applying to a single step or state transition, and those applying to sequences of states or transitions. In this study, we limit ourselves to finite (but unbounded) sequences or traces. This means the resulting formulas can express safety properties but not liveness properties. Extension of the approach to cover liveness is left as a future refinement.

This two-class taxonomy is not fine-grained enough, however. It fails to capture a notion of mode-based behavior, a key attribute of systems such as SAFER. The AAH mode depicted in Figure 4 is a feature that induces long-term cycles in system behavior, which play a central role by naturally dividing system traces into distinct segments.

To deal with modal behavior we introduce specialized property classes having a semantics similar to that of the *until* operator found in many temporal logics. In such logics the formula $\phi U \psi$ is true if ϕ holds in all future states until ψ becomes true. Our formulation is based on the "weak" until operator; the formula ψ is not required to become true eventually as it is in the strong-until case.

While temporal operators are not available in PVS, its higher order features allow us to create definitions for expressing conditional invariants based on temporal concepts. We postulate a *hold-until* formalism that takes three predicate conditions to express invariants over restricted subsequences of states. Let the (higher-order) predicate $HU(T, H, U, I)$ define the condition that H must hold over all states within each sequence bounded by a point at which trigger condition T becomes true and a point at which stop condition U becomes true. Thus, upon T the invariant H holds until U obtains.

Predicates are applied to states generated from the sequence of inputs I. If $I_1, I_2, \ldots$ are inputs presented to the system, the states $S_1, S_2, \ldots,$ result after repeatedly applying the main transition function to initial state S_0. Let $CS(I)$ denote the current state after cumulatively processing all the inputs from sequence I. HU is defined recursively by applying its three predicate arguments to $CS(\langle I_1, \ldots, I_n \rangle)$ for all initial subsequences of I.

A hold-until property is one where an invariant H holds over each sequence of states bracketed by the earliest occurrences of the trigger condition T and the termination condition U:

$$
\begin{array}{llllllll}
S_0, S_1, \ldots, & S_{i-1}, & S_i, & S_{i+1}, & \ldots, & S_{j-1}, & S_j, & S_{j+1}, \ldots \\
 & \neg T, & T & & & \neg U, & U & \\
 & & H, & H, & \ldots, & H & &
\end{array}
$$

If T first becomes true on S_i, and U first becomes true on S_j, then H must hold on states S_i through S_{j-1}. If there are multiple occurrences of segments satisfying the hold-until conditions, then H must hold over each such segment.

After considering the nature of the SAFER requirements, we settled on five property classes based on the following types of assertions.

1. Transition function outputs that result from specific inputs.
2. Relationships between pairs of successive states.
3. Unconditional state invariants applying to all states.
4. Hold-until invariants over sequences of states bracketed by triggering and terminating conditions.
5. Hold-until invariants concerning "frames," which include input, output, and previous/next states.

Class 5 can be seen as a generalization of class 4, provided for those cases where we wish to make statements about particular transitions or input/output values. The distinction between these two classes and the significance of frames will be clarified in Sections 3.3 and 5.1.

The five property classes are matched with canonical formulas, each having an assumed structure:

1. $P(s) \wedge Q(in) \supset R(out(tran(in, s)))$
2. $P(s_1) \wedge Q(s_2) \wedge s_2 = state(tran(in, s_1)) \supset R(s_1, s_2)$
3. $P(CS(I))$
4. $HU(\lambda s\colon T(s), \lambda s\colon H(s), \lambda s\colon U(s), I)$
5. $HU(\lambda f\colon T(f), \lambda f\colon H(f), \lambda f\colon U(f), I)$

Generic functions *tran*, *out*, and *state* are abbreviations for the specific instances `SAFER_control`, `output`, and `state` found in Figure 3.

The ordinary invariants of class 3 also can be expressed as special cases of class 4 formulas, which follows from this equivalence:

$$P(CS(I)) \equiv HU(\lambda s\colon true, \lambda s\colon P(s), \lambda s\colon false, I)$$

Invariants in the form of an implication also have a hold-until expression:

$$
\begin{array}{l}
Q(CS(I)) \supset P(CS(I)) \equiv \\
HU(\lambda s\colon true, \lambda s\colon P(s), \lambda s\colon \neg Q(s), I)
\end{array}
$$

While these equivalences would allow class 3 to be subsumed by class 4, there is no advantage in doing so, as the hold-until proofs include more overhead than the more direct proofs used for the unconditional invariants.

3.3 Hold-Until Invariants in PVS

Formalizing the hold-until style of invariants is easily accomplished with the following PVS definitions. We use the built-in type `list`, which is semantically equivalent to a Lisp-style list, to represent system input sequences. First, introduce the function `cur_state` as the definition for CS.

```
cur_state(inputs: input_list)
         : RECURSIVE avionics_state =
 CASES inputs OF
   null: initial_avionics_state,
   cons(input, rest):
     state(SAFER_control(input, cur_state(rest)))
 ENDCASES
 MEASURE length(inputs)
```

Let state predicates be represented by the following function type:

```
state_pred:  TYPE = [avionics_state -> bool]
```

Then add the (higher order) predicates `holding` and `hold_until` to express the invariant properties as recursive functions on input sequences, given the three predicate arguments T, H, and U. Note that the recursion proceeds backward in time, from recent to earlier events. The auxiliary function `holding` indicates whether the invariant should apply to the current state.

```
holding(trigger, until: state_pred,
        inputs: input_list): RECURSIVE bool =
 CASES inputs OF
   null: NOT until(initial_avionics_state)
           AND trigger(initial_avionics_state),
   cons(input, rest):
      NOT until(cur_state(inputs)) AND
      (trigger(cur_state(inputs)) OR
       holding(trigger, until, rest))
 ENDCASES
 MEASURE length(inputs)
```

The top-level function `hold_until` defines the predicate HU by applying the function `holding`.

```
hold_until(trigger, hold, until: state_pred,
           inputs: input_list): RECURSIVE bool =
 CASES inputs OF
   null: holding(trigger, until, null)
           IMPLIES hold(initial_avionics_state),
   cons(input, rest):
      hold_until(trigger, hold, until, rest) AND
      (holding(trigger, until, inputs)
         IMPLIES hold(cur_state(inputs)))
 ENDCASES
 MEASURE length(inputs)
```

A frame-oriented version of the hold-until invariant is readily constructed from the state-oriented version. The primary difference is the use of a "frame" structure for each transition that records the input, output, and both previous and next state values.

```
frame_event: TYPE =
     [# input:     avionics_inputs,
        prev, cur: avionics_state,
        output:    avionics_outputs #]

cur_frame(inputs: {il: input_list | il /= null})
         : frame_event =
  LET input = car(inputs), rest = cdr(inputs) IN
  (# input  := input,
     prev   := cur_state(rest),
     cur    := cur_state(inputs),
     output :=
       output(SAFER_control(input,
                            cur_state(rest))) #)
```

The other definitions are close enough to the class 4 versions that we omit the details. Greater precision is available through frame invariants. Rather than merely conditioning a property on being in a certain state, it is possible to identify the exact input event that gives rise to the state. The individual transitions marking the endpoints of a segment of interest also can be better characterized. For example, by allowing a predicate to refer to both `prev(f)` and `cur(f)`, it is possible to constrain the trigger condition to select the desired transition. Section 5.1 provides some concrete illustrations of the frame-based hold-until invariants and how they can be used in practice.

3.4 SAFER Properties

Based on the foregoing classification scheme for system properties, the case study proceeded by drafting a variety of properties distributed across the five classes. A total of 42 properties formed the final set, with the breakdown by class being 9, 12, 1, 9, and 11. The choice was driven by a desire for a representative sample of properties. In the class 1 case, many more could have been formulated, but because most concerned the selection of thrusters under various input values, it was sufficient to provide one formula for each major combination of input values. A comprehensive set would have been quite large, which might be desirable in practice but would be excessive here. Conversely, class 3 contains only one property. In this case, there was little that could be expressed in the form of an unconditional invariant. Nearly all of the meaningful invariant properties needed the hold-until form. We could have dispensed with class 3 entirely, but decided to retain it for completeness and its potential use in future applications.

Three PVS theories were used to encode the SAFER properties. Each imports the PVS definitions provided by the SAFER model. The complete text of these theories is available electronically [4].

4 Customized Prover Strategies

Despite steady progress, mechanized theorem proving remains a labor-intensive activity. Customizing the proof process is a promising way to reduce the workload and make deduction more accessible to less experienced practitioners.

4.1 PVS Theorem Prover

PVS proof notation is based on a Gentzen-style sequent calculus, which gives the prover a convenient and symmetric structure for normalizing logical formulas. In a PVS sequent, (numbered) antecedent formulas serve as premises for establishing (numbered) consequent formulas:

```
[-1] <antecedent 1>
[-2] <antecedent 2>
  |-------
 [1] <consequent 1>
 [2] <consequent 2>
```

Antecedents are considered to form a conjunction while consequents form a disjunction. Every user-invoked prover command (inference rule) causes one or more new sequents to be generated, moving the proof closer to completion. Rules are considered primitive, whereas strategies are higher level operations that apply rules and other strategies. The PVS concept of strategies resembles the "tactics" of LCF [17], HOL [5], and related theorem provers.

PVS provides over 50 built-in prover rules and strategies. Exactly which are primitive is immaterial to a user, who invokes them all in the same manner. By providing built-in strategies, PVS offers a series of increasingly muscular prover operations. The more powerful operations increase automation, but tend to be more expensive in machine resources as well as more speculative, sometimes steering the proof in unproductive directions. An example is the built-in strategy `grind`, which attempts a wide variety of steps in hopes of completing the current proof. When it succeeds, `grind` is indeed a powerful assistant. When it fails, it can lead to explosive proof growth.

To automate highly repetitive proofs, or to cope with special cases, PVS users can write their own strategies. A set of PVS-supplied strategy constructors plus a set of Common Lisp macros are the key mechanisms for such development. Predefined Lisp functions also are available for accessing various parts of the current sequent. Users may add conventional Lisp functions to carry out intermediate computations. By collecting all pertinent definitions in a file called `pvs-strategies`, users can have custom strategies automatically included in any prover session.

A simple example follows, in which the strategy constructor `try` is used to exploit antecedent equalities.

Table 1. Prover commands for strategy-based proofs.

Class	No. of formulas	Prover command
1	9	`(state-tran)`
2	12	`(state-tran *)`
3	1	`(state-invar "inputs")`
4	9	`(hold-until "inputs")`
5	11	`(frame-until "inputs")`

```
(defstep replace-hide (n)
  (try (replace n) (hide n) (skip))
  "(replace-hide n) : does replace on formula n
   then hides the original formula on success."
  "~%Replacing lhs by rhs from ~a")
```

Given the number of an antecedent formula as the argument `n`, the `replace-hide` strategy first tries to apply the `replace` command, which performs a term substitution throughout the sequent, replacing instances of the left-hand side of an equality with the right-hand side. If successful, the strategy then hides antecedent `n`, removing it from the current sequent. If unsuccessful, no further action is taken. The two strings at the bottom of the `defstep` definition provide user documentation and a message for the prover to display after carrying out the strategy.

4.2 Strategy Development

For the SAFER study we developed four top-level strategies, aimed at formulas of the types described in Section 3.2. Table 1 shows the PVS prover commands used to invoke all of the property proofs. Several lower-level strategies were likewise developed as part of the effort, some of which might find application elsewhere.

Proofs for class 1 and 2 properties, if carried out using only low-level prover rules, are tedious but unchallenging, generally based on repeated function expansion and simplification steps. One of the more powerful built-in rules or strategies, such as `grind`, might be expected to fully automate such proofs. Some functions in the SAFER model, however, contain tables of moderate complexity, leading to a great deal of multiway branching. This is a common characteristic of models that describe realistic artifacts. Wholesale expansion of the full function tree might easily result in combinatorial explosion before simplification has a chance to occur. Hence, simply unleashing `grind` or its relatives on the SAFER properties is unlikely to be productive.

To automate these proofs requires a more measured approach, one that is cognizant of the model functions and how to expand them in an appropriate manner. After encoding this knowledge, one must construct a specialized strategy for each property class, tailored to the assumed structure of the formulas for which it is in-

```
(defstep state-tran
        (&optional (exp-fnums +) rewrites)
 (let ((auto-rewr
        (cons 'auto-rewrite
              (append rewrites
                      (constant-rewrites)))))
   (then (skosimp*)
        auto-rewr (assert)
        (split-disjunctions*)
        (expand-rec-desc exp-fnums)
        (general-rewrites)
        (ground) (lift-if)
        (grind)))
 "(state-tran): prove state transition properties
  by expansion, replace & hide, then grind."
 "~%Invoking state transition property strategy")
```

Fig. 6. PVS strategy to prove state transition properties.

tended. In the case of the invariant classes, this means setting up and guiding a suitable induction proof.

An example of the custom prover strategies is shown in Figure 6, which depicts the strategy `state-tran`, used for proving transition properties (classes 1 and 2). Note that at the PVS prover interface, arguments to rules and strategies are normally not evaluated. Thus within a strategy definition, expression evaluation conventions are a hybrid of those from Common Lisp and those peculiar to the PVS prover.

The basic idea behind `state-tran` is to perform an optional set of rewrites, selectively expand functions in a SAFER-friendly order, alternate expansion with equality substitution and simplification, then invoke the standard PVS "hammers," `ground` and `grind`, to finish off the proof. Within the definition of `state-tran`, a sequence of proof steps is invoked using the strategy constructor `then`. The formula is first Skolemized (`skosimp*`), then a set of automatic rewrite rules is installed using the list built by the `let` expression, followed by a simplification step (`assert`). After splitting disjunctions, `state-tran` expands functions using `expand-rec-desc`, and follows with additional rewriting. The strategy concludes by invoking three built-in prover commands. By using custom SAFER knowledge early in the proof, we hope to drastically reduce the complexity of the intermediate sequents so that the final proof stage becomes tractable for the built-in prover commands.

At the core of the `state-tran` definition is another custom strategy called `expand-rec-desc`, whose goal is to expand functions by traversing the state-transition function tree in a manner similar to recursive descent. Actually, it proceeds down the tree in "layers," where each layer represents a band of function definitions having common expressions in their expansions. Each iteration in the descent causes all the functions at a layer to be expanded, then an attempt is made to consume antecedent equalities using `replace-hide`, described in Section 4.1. By replacing with terms from equalities, the next round of expansions will cause some simplification by eliminating impossible branches in conditional expressions.

To make use of `expand-rec-desc`, it is necessary to declare the SAFER function hierarchy within the strategy file so it is available during proofs. Only the upper subtree needs to be declared; the lowest level functions can safely be handled by `grind` at the end. The structure is actually a list of lists rather than a tree. For the SAFER properties, three layers usually suffice:

```
(defun spec-fn-tree ()
  '(("SAFER_control")
    ("selected_actuators" "AAH_transition"
     "grip_command")
    ("mask_AAH" "integrated_commands"
     "selected_thrusters") ))
```

For property classes 3–5, induction proofs over the list data type are necessary. For class 3 proofs, the strategy `state-invar` was introduced to manage the induction, which proceeds by using `grind` to dispense with the base case, then invokes `state-tran` to carry out the induction step.

```
(defstep state-invar (var)
 (let ((base-case  '(grind))
       (induc-step (list 'state-tran '*
                         (invariant-rewrites)))
       (cases (list base-case induc-step)))
   (branch (induct var) cases))
   "(state-invar VAR) : proves state invariant by
    induction on VAR."
   "~%Inducting and simplifying on ~a")
```

The strategy constructor `branch` is used to distribute the base case and induction step instructions to the results of invoking the built-in prover command `induct`. Once this split has been made, the induction step proof is carried out by treating it as a slightly specialized instance of a single-transition proof.

Proving hold-until invariants involves a more elaborate strategy than `state-invar`. The mostly generic `state-tran` strategy is replaced by more specialized ones that introduce a few steps to handle the `hold_until` definition. In addition, the following lemma is needed to carry forward hold and until-conditions from the induction hypothesis.

```
state_holding: LEMMA
    hold_until(trigger, hold, until, inputs) AND
    holding(trigger, until, inputs)
  IMPLIES
    hold(cur_state(inputs)) AND
    NOT until(cur_state(inputs))
```

Class 4 formulas are proved using the `hold-until` strategy (details omitted), which begins its induction-step branch by carefully expanding the special functions `hold_until`, `holding`, and `cur_state`. Then the lemma `state_holding` is brought in, and the rest of the proof resembles the transition case. A similar process is followed by strategy `frame-until` for class 5 formulas.

5 Summary of Proof Results

Applying the methodology described in Sections 3 and 4 produced encouraging results. We present a sampling of the SAFER properties formulated and proved using strategies for the five classes. Refer to Figure 3 for the role of various specification functions.

5.1 Sample Properties

We begin with two formulas that are examples of class 1 properties. The first expresses the property mentioned earlier, that no thrusters should be fired when AAH is off and no hand grip command is present.

```
no_thrusters_axes_off: LEMMA
  all_axes_off(active_axes(AAH_state(a_st))) AND
  grip_command(grip_command(a_in),
               MODE(HCM_switches(a_in)))
    = null_six_dof
IMPLIES
  prop_actuators(output(SAFER_control(a_in,a_st)))
    = null
```

State component `active_axes` is a boolean vector indexed by rotational axis, and `all_axes_off` tests if every element is false.

The second property asserts which thrusters should be fired when a particular nonnull hand grip command is present.

```
Y_Z_thrusters: LEMMA
  all_axes_off(active_axes(AAH_state(a_st))) AND
  grip_command(grip_command(a_in),
               MODE(HCM_switches(a_in))) =
    (# tran := null_tran_command WITH
                 [(Y) := POS, (Z) := NEG],
       rot  := null_rot_command #)
IMPLIES
  prop_actuators(output(SAFER_control(a_in,a_st)))
    = (: R2R, R4R, R2F, R4F :)
```

This formula requires that if AAH is off and a grip command requests positive Y thrust and negative Z thrust, the individual thrusters `R2R, R4R, R2F, R4F` should be actuated.

Next is an example of a class 2 property, relating the two successive states associated with a transition. In the following formulas, `toggle(AAH_state(s))` refers to the state/mode within the diagram of Figure 4.

```
timeout_pressed_once: LEMMA
  toggle(AAH_state(s1)) = AAH_on AND
  toggle(AAH_state(s2)) = pressed_once AND
  s2 = state(SAFER_control(a_in, s1))
IMPLIES
  timeout(AAH_state(s2))
     = clock(s1) + click_timeout
```

Thus, when a transition to `pressed_once` occurs, the double-click timeout begins by setting the `timeout` state component.

An ordinary state invariant (class 3) is shown below. It asserts that all rotation axes are AAH-inactive whenever the overall mode is `AHH_off`.

```
axes_off_AAH: LEMMA
  toggle(AAH_state(cur_state(inputs))) = AAH_off
IMPLIES
  NOT active_axes(AAH_state(cur_state(inputs)))(r)
```

The next two formulas show examples of hold-until state invariants. The first is a direct expression of the property presented in Section 3.1:

> Once AAH is turned off for a rotational axis it remains off until a new AAH cycle is initiated.

```
rot_axis_stays_off: LEMMA
 hold_until(
   LAMBDA s: toggle(AAH_state(s)) = AAH_on AND
             NOT active_axes(AAH_state(s))(r),
   LAMBDA s: NOT active_axes(AAH_state(s))(r),
   LAMBDA s: toggle(AAH_state(s)) = AAH_started,
   inputs)
```

The three lambda expressions specify the trigger, hold, and until-conditions. For any axis `r`, the trigger condition is true when AAH is on and axis `r` becomes inactive. It must remain inactive until reaching `AAH_started`.

The second invariant expresses a simple fact about the AAH state diagram, namely, that if the state `AAH_on` is reached, the state `AAH_started` cannot recur until after `AAH_off` is reached.

```
on_to_off_path: LEMMA
 hold_until(
   LAMBDA s: toggle(AAH_state(s)) = AAH_on,
   LAMBDA s: toggle(AAH_state(s)) /= AAH_started,
   LAMBDA s: toggle(AAH_state(s)) = AAH_off,
   inputs)
```

The last two formulas show examples of hold-until invariants on frames rather than states. Recall the frame data structure shown in Section 3.3. The first formula expresses the property that once the AAH state reaches `AAH_started`, it remains is this state until the pushbutton is released and returned to the up position.

```
AAH_started_exit: LEMMA
 hold_until(
   LAMBDA f: toggle(AAH_state(prev(f)))
                = AAH_started,
   LAMBDA f: toggle(AAH_state(cur(f)))
                = AAH_started,
   LAMBDA f: AAH(HCM_switches(input(f)))
                = button_up,
   inputs)
```

Note how the until-condition is expressed in terms of an input value as the event of interest and makes no mention of the state.

The last example is a more ambitious property, triggered by the transition from `AAH_on` to `pressed_once`, i.e., when the button is depressed to begin a double-click sequence. The invariant shows that the timeout time set at this transition is not exceeded when the state advances to and remains in `AAH_closing`.

Table 2. Summary of PVS theories and property proofs.

Category	Classes	Formulas	Avg. time
Transition properties	1,2	21	1.6 sec
State invariants	3,4	10	6.4 sec
Frame invariants	5	11	6.1 sec

```
closing_path_duration: LEMMA
 hold_until(
   LAMBDA f: toggle(AAH_state(prev(f))) = AAH_on
        AND  toggle(AAH_state(cur(f)))
               = pressed_once
        AND  t = clock(cur(f)) + click_timeout,
   LAMBDA f: t = timeout(AAH_state(cur(f))) + 1
        AND  (toggle(AAH_state(cur(f)))
                   = AAH_closing
               IMPLIES clock(cur(f)) <= t),
   LAMBDA f: NOT (toggle(AAH_state(cur(f)))
                     = pressed_once OR
                  toggle(AAH_state(cur(f)))
                    = AAH_closing),
   inputs)
```

In this final example, the trigger condition pinpoints a particular transition using the prev and cur parts of the frame record. The added machinery of a frame-based invariant is needed because the exact transition must be identified to obtain the correct timeout time, which is set only on first arriving at the pressed_once state. A variable t, implicitly a universally quantified variable, is introduced in the formula to represent this time.

Also, closing_path_duration is a case that straddles the feasibility boundary for model checking. Proving this formula requires reasoning about linear arithmetic, which is a decidable theory fully automated by PVS. While model checkers vary in their numerical abilities, many are unable to handle common types of arithmetic reasoning, forcing the analyst to finesse the problem or accept a less thorough analysis.

For the interested reader, all PVS proof files and the strategy source file needed to reproduce our results are electronically accessible [4].

5.2 Proof Results

Table 2 shows the average proof times for the different classes of properties. These figures reflect the prover execution times for PVS version 2.3 running under Solaris on a 333 MHz Sun Ultra 5 platform. All 42 properties were proved by invoking a single top-level strategy. No further user input was required.

During the early stages, failed proof attempts were usually caused by improperly stated formulas where the conjectures turned out not to be theorems. Often these resulted from incomplete understanding of the model. Most common was overlooking that an extra transition is sometimes required before the effects of an event are fully propagated. This produced valuable insights about the model itself.

A less frequent case of proof difficulty occurred when the desired invariant was true, but the formula was not strong enough to carry an induction proof. Adding a supporting condition to the invariant is enough to repair such formulas. This deficiency is quite common in general when attempting induction proofs.

The most interesting case of proof failure is one that led to the discovery of a flaw in the original model. While attempting to prove closing_path_duration, shown in Section 5.1, we realized there was an input sequence that would violate the timeout requirement.

In the guidebook's version of the AAH protocol, the transition from pressed_once back to AAH_on was missing. This omission allowed a button-up event, while in state pressed_once, to proceed to AAH_closing, where a button-down event on the very next frame would advance to pressed_twice, thereby escaping the timeout path if the button had been held too long while residing in state pressed_once. Admittedly, this is unlikely to occur in practice because it requires the button to move up and then down again within 5 msec. Nevertheless, it was a logical possibility whose presence was detected by the inability to prove a behavioral property.

Consequently, the version of the SAFER model used in our study differs slightly from that found in the guidebook. An additional change was introduced in response to an issue raised by Agerholm and Larsen [1] concerning the masking of AAH control law outputs when AAH mode is off.

A further change was prompted by Mikk [10], whose experimental modeling and analysis of SAFER uncovered a case where the second button press of a double click could be late by a single clock tick and still be accepted. While this is not a problem in practice because an overage of one tick on a timeout is insignificant, the model was nevertheless revised to eliminate this anomaly. Both this issue and the previously mentioned timeout problem were the result of a convention that gives button-down events precedence over secondary transitions such as timeouts. Paying insufficient attention to the consequences of this choice led to the protocol errors.

5.3 Discussion

Comparing our results to what might be achieved using other techniques, we observe that the SAFER model is well suited to automatic analysis. Class 1 and 2 properties, for instance, could be handled by symbolic execution with heavy simplification. Many of SAFER's state components have bounded types, posing opportunities for model checking in the invariant properties. Even the unbounded types might be handled using parametric model checking techniques [8]. Hence, this example should be useful for comparing approaches to analysis based on model checking and theorem proving.

Overall, many target models are infinite state systems, pushing them toward the theorem provers' natural domain. Specification languages based on general logics require none of the aggressive abstraction needed for representation using model checking languages. In the SAFER case study, all the data object reasoning fell within the scope of the PVS prover's decision procedures. When more difficult mathematical characteristics are present, such as nonlinear arithmetic, obtaining the same degree of proof automation becomes more challenging. The proof strategies used in this study, however, could be extended to accommodate richer data domains. If lemmas about nonlinear arithmetic or sequences or trees should be needed in proofs, strategies can be programmed to import and instantiate them. Theorem provers have used lemma databases for years. What we propose is to make the search and application of lemmas more highly tailored to particular classes of formulas and application domains. Tailoring will help control proof growth and extend the limits of practicality.

Problem domain experts, lightly trained in formal methods, should be able to apply the methods of this study. With the help of a seasoned prover consultant, requirements analysts could engage in formal modeling and lightweight theorem proving using custom strategies. Despite a weak ability to generate counterexamples, theorem proving's advantages remain formidable. The expressiveness of a language like PVS, coupled with specialized techniques such as the hold-until invariants, allow nontrivial properties to be written naturally. Contrast this with some varieties of model checking, where certain properties are difficult to capture using CTL-like languages, possibly leading inexperienced users to write erroneous formulas. While more development and experimentation with our approach are warranted, particularly to reduce brittleness and sensitivity to form within strategies, it offers promise to practitioners who are willing to learn basic theorem proving techniques.

For these reasons, we feel the approach presented offers a way to analyze simple requirements as easily as model checking, while simultaneously avoiding the steep cliff of infeasibility that finite state techniques face with more demanding requirements. Analysis effort will increase more gradually with strategy-based proving, yielding a wider range of applicability with no loss of automation at the low end.

6 Conclusion

We have presented a method for achieving highly automated proofs of certain requirements model properties. The method is applicable to systems naturally modeled as state machines, which is often the case for real-time embedded controllers. Not all models will enjoy the degree of proof automation exhibited by the SAFER requirements, but we expect most systems to have at least some properties that can be handled by these techniques.

Further development along these lines could produce analogous classes of models and properties whose traits readily lend themselves to comparable styles of automation. Custom PVS strategies work well for this purpose, although the user community would benefit from greater insight into the prover's mechanisms. We anticipate future efforts by research groups and problem domain experts to exploit the strategy capability of PVS. Eventually, a library of models, definitions, and supporting strategies would do much to further the goal of transferring deduction technology to the requirements engineering community.

Acknowledgments

The author acknowledges the cooperation and support of NASA Langley researchers during the course of this study, in particular, Ricky Butler and Paul Miner. Earlier discussions with Judy Crow of SRI also were helpful in developing ideas. Erich Mikk provided useful insights and comments on the model formulation. Ongoing support of the PVS toolset by SRI has been beneficial. This work was supported in part by the National Aeronautics and Space Administration under Contract NAS1-96014.

References

1. Sten Agerholm and Peter Gorm Larsen. Modeling and validating SAFER in VDM-SL. In *Lfm97, Fourth NASA Langley Formal Methods Workshop*, Hampton, Virgina, September 1997. NASA Conference Publication 3356.
2. Ricky W. Butler, James L. Caldwell, Victor A. Carreno, C. Michael Holloway, Paul S. Miner, and Ben L. Di Vito. NASA Langley's research and technology transfer program in formal methods. In *Tenth Annual Conference on Computer Assurance (COMPASS 95)*, Gaithersburg, MD, June 1995.
3. Judith Crow and Ben Di Vito. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.
4. Ben Di Vito. PVS source files for SAFER. Available at `http://shemesh.larc.nasa.gov/~bld/safer/`.
5. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, England, 1993.
6. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

7. C. L. Heitmeyer, R. D. Jeffords, and B. L. Labaw. Consistency checking of SCR-style requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
8. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1):110–122, December 1997.
9. N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):694–707, September 1994.
10. Erich Mikk. Christian-Albrechts-University of Kiel. Private communication, 12 April 1999.
11. *Project Requirements Document for the Simplified Aid for EVA Rescue (SAFER) Flight Test Project*, December 1992. NASA JSC-24691.
12. *Simplified Aid for EVA Rescue (SAFER) Flight Test Project - Flight Test Article Prime Item Development Specification*, July 1994. NASA JSC-25552.
13. *Simplified Aid for EVA Rescue (SAFER), Operations Manual*, September 1994. NASA JSC-26283.
14. NASA Office of Safety and Mission Assurance, Washington, DC. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*, July 1995. NASA-GB-002-95, Release 1.0.
15. NASA Office of Safety and Mission Assurance, Washington, DC. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion*, May 1997. NASA-GB-001-97, Release 1.0. Available at `http://eis.jpl.nasa.gov/quality/Formal_Methods/`.
16. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
17. L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, Cambridge, England, 1987.
18. John Rushby. Formal methods and digital systems validation for airborne systems. NASA Contractor Report 4551, December 1993.
19. John Rushby. Formal methods and their role in digital systems validation for airborne systems. NASA Contractor Report 4673, August 1995.
20. M. K. Srivas and S. P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.