NASA Technical Memorandum 110255

An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot

Ricky W. Butler
Langley Research Center, Hampton, Virginia

May 1996

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

Contents

Abstract

This paper presents an introduction to capturing software requirements in the PVS formal language. The object of study is a simplified digital autopilot that was motivated in part by the mode control panel of NASA Langley's Boeing 737 research aircraft. The paper first presents the requirements for this autopilot in English and then steps the reader through a translation of these requirements into formal mathematics. Along the way deficiencies in the English specification are noted and repaired. Once completed, the formal PVS requirement is analyzed using the PVS theorem prover. and shown to maintain an invariant over its state space.

KEYWORDS: Formal Methods, Software Requirements, Verification, Autopilot

1 Introduction

In this paper, the process of translating requirements into a formal language will be explored. The chosen target specification language is SRI International's PVS Language[10], however, the modeling techniques used in this paper can be used with other formal specification languages that are based upon higher-order logic. The exposition is centered around the analysis of a simple autopilot that is somewhat related to a early Boeing-737 autopilot. The requirements for this autopilot are first given in English and then translated in a step-by-step manner into PVS. This "requirements capture" process is discussed in detail.

All of the PVS language features that are used are explained thoroughly to make the paper self-contained. More detailed information about PVS can be obtained from [9, 8, 14]. Also, several tutorial introductions to PVS are available [6, 3, 16, 4, 11, 15].

2 Example Application

The techniques of formal specification and verification of an avionics subsystem will be demonstrated on a very simplified example of a mode-control panel. An informal, English-language specification of the mode-control panel representative of what software developers typically encounter in practice will be presented. The process of clarifying and formalizing the English specification into a formal specification, often referred to as requirements capture, will then be illustrated.

2.1 English Specification of the Example System

This section presents the informal, English-language specification of the example system. The English specification is annotated with section numbers to facilitate references back to the specification in later sections.

1. The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values, as shown in Figure 1. The system supports the following four modes:

attitude control wheel steering (`att_cws`*)*
flight path angle selected (`fpa_sel`*)*
altitude engage (`alt_eng`*)*
calibrated air speed (`cas_eng`*)*

Only one of the first three modes can be engaged at any time. However, the `cas_eng` *mode can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes,* `att_cws`*,* `fpa_sel`*, or* `alt_eng`*, should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.*

2. There are three displays on the panel: and altitude [ALT], flight path angle [FPA], and calibrated air speed [CAS]. The displays usually show the current values for the altitude, flight path angle, and air speed of the aircraft. However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display. This is the target or "pre-selected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to 25,000 feet, he will dial 25,000 into the altitude display window and then press the `alt_eng` *button to engage*

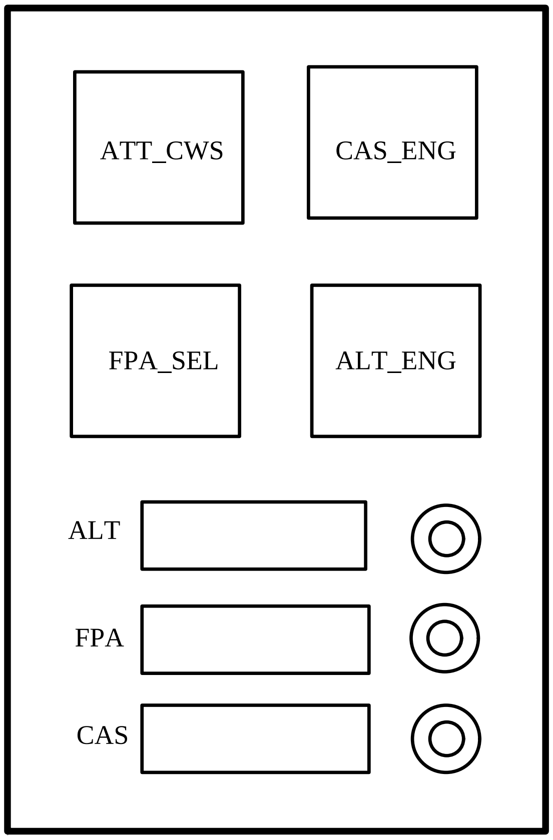


Figure 1: Mode Control Panel

the altitude mode. Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value.

3. If the pilot dials in an altitude that is more than 1,200 feet above the current altitude and then presses the `alt_eng` *button, the altitude mode will not directly engage. Instead, the altitude engage mode will change to "armed" and the flight-path angle select mode is engaged. The pilot must then dial in a flight-path angle for the flight-control system to follow until the aircraft attains the desired altitude. The flight-path angle select mode will remain engaged until the aircraft is within 1,200 feet of the desired altitude, then the altitude engage mode is automatically engaged.*

4. The calibrated air speed and the flight-path angle values need not be pre-selected before the corresponding modes are engaged—the current values displayed will be used. The pilot can dial-in a different target value after the mode is engaged. However, the altitude must be pre-selected before the altitude engage button is pressed. Otherwise, the command is ignored.

5. The calibrated air speed and flight-path angle buttons toggle on and off every time they are pressed. For example, if the calibrated air speed button is pressed while the system is already in calibrated air speed mode, that mode will be disengaged. However, if the attitude control wheel steering button is pressed while the attitude control wheel steering mode is already engaged, the button is ignored. Likewise, pressing the altitude engage button while the system is already in altitude engage mode has no effect.

Because of space limitations, only the mode-control panel interface itself will be modeled in this example. The specification will only include a simple set of commands the pilot can enter plus the functionality needed to support mode switching and displays. The actual commands that would be transmitted to the flight-control computer to maintain modes, etc., are not modeled.

2.2 Formally Specifying the Example System

In this section, we will describe the process of formally specifying the mode-control panel described in English in the previous section. Quotes cited from the English specification will be annotated with the corresponding section number in parentheses. The goal is to completely describe the system requirements in a mathematical notation, yet not overly constrain the implementation.

This system collects inputs from the pilot and maintains the set of modes that are currently active. Thus, it is appropriate to model the system as a state machine. The state of the machine will include the set of modes that are active, and the pilot inputs will be modeled as events that transition the system from the current state (S_c) to a new state (S_n):

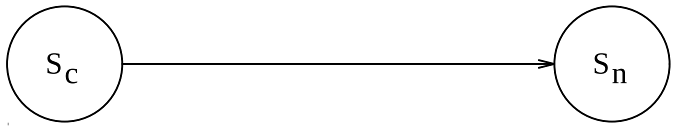


Figure 2: State transition function

The arrow represents a transition function, `nextstate`, which is a function of the current state and an event, say `ev`:

$$S_n = \texttt{nextstate}(S_c, \texttt{ev}).$$

The goal of the formal specification process is to provide an unambiguous elaboration of the `nextstate` function. This definition must be complete; i.e., it must provide a next state for all possible events and all possible states. Thus, the first step is to elaborate all possible events and the content of the state of the machine. The system will be specified using the PVS specification language [12, 16, 14, 8].

2.3 Events

The pilot interacts with the mode-control panel by pressing the mode buttons and by dialing pre-selected values into the display. The pilot actions of pressing one of the four buttons will be named as follows: `press_att_cws`, `press_cas_eng`, `press_alt_eng`, and `press_fpa_sel`. The actions of dialing a value into a display will be named as follows: `input_alt`, `input_fpa`, and `input_cas`. The behavior of the mode-control panel also depends upon the following inputs it receives from sensors: `alt_reached`, `fpa_reached`, and `alt_gets_near`. In PVS, the set of events are specified as follows:

```
events: TYPE = { press_att_cws, press_cas_eng, press_alt_eng,
                 press_fpa_sel, input_alt, input_fpa, input_cas,
                 alt_reached, alt_gets_near, fpa_reached }
```

2.4 State Description

The state of a system is a collection of attributes that represents the system's operation. In the example system, the set of active modes would certainly be a part of the system state. Also, the values in the displays that show altitude, flight-path-angle and air-speed and the readings from the airplane sensors would be included in the state. It is important to find a set of attributes that enable a full description of the system behavior and an efficient method of representing these attributes.

One possible approach to describing the system state for the example is to use a set to delineate which modes are active. For example, { `att_cws`, `cas_eng` } would represent the state of the system where both `att_cws` and `cas_eng` are engaged but `alt_eng` and `fpa_sel` are not engaged. The `alt_eng` mode has the additional capability of being armed. Thus, a better approach to describing the example system state is to associate with each mode one of the following values: `off`, `armed`, or `engaged`. In PVS, a type or domain can be defined with these values:

```
mode_status: TYPE = {off, armed, engaged}
```

The state descriptor is as follows:

```
[# % RECORD
att_cws: mode_status,
cas_eng: mode_status,
fpa_sel: mode_status,
alt_eng: mode_status,
#] % END
```

For example, the record

```
[att_cws=engaged, cas_eng=engaged, fpa_sel=off, alt_eng=off]
```

would indicate a system where both `att_cws` and `cas_eng` are `engaged` while `fpa_sel` and `alt_eng` are `off`. However, there is still a problem with the state descriptor. In the example system only `alt_eng` can be `armed`. Thus, more restrictive domains are needed for the modes other than `alt_eng`. This can be accomplished by defining the following sub-type of `mode_status`:

```
off_eng: TYPE = {mode: mode_status | mode = off OR mode = engaged }
```

Thus, the type `off_eng` has two values: `off` and `engaged`. The state descriptor is thus modified to become:

```
[# % RECORD
att_cws: off_eng,
cas_eng: off_eng,
fpa_sel: off_eng,
alt_eng: mode_status
#] % END
```

The mode panel also maintains the state of the displays. To simplify the example, the actual values in the displays will not be represented. Instead, the state descriptor will only keep track of whether the value is a pre-selected value or the actual value read from a sensor. Thus, the following type is added to the formalism:

```
disp_status: TYPE = {pre_selected, current }
```

and three additional fields are added to the state descriptor:

```
alt_disp: disp_status
fpa_disp: disp_status
cas_disp: disp_status
```

The behavior of the mode-control panel does not depend upon the actual value of "altitude" but rather on the relationship between the actual value and the pre-selected value in the display. The following "values" of altitude are sufficient to model this behavior:

```
altitude_vals: TYPE = { away, near_pre_selected, at_pre_selected },
```

and the following is added to the state descriptor:

```
altitude: altitude_vals.
```

The final state descriptor is:

```
states: TYPE=[# % RECORD
        att_cws:  off_eng,
        cas_eng:  off_eng,
        fpa_sel:  off_eng,
        alt_eng:  mode_status,
        alt_disp: disp_status,
        fpa_disp: disp_status,
        cas_disp: disp_status,
        altitude: altitude_vals
        #] END
```

2.5 Formal Specification of `nextstate` Function

Once the state descriptor is defined, the next step is to define a function to describe the system's operation in terms of state transitions. The `nextstate` function can be defined in terms of ten sub-functions, one for each event, as follows:

```
event: VAR events
st: VAR states
nextstate(st,event): states =
        CASES event OF
                press_att_cws: tran_att_cws(st),
                press_alt_eng: tran_alt_eng(st),
                press_fpa_sel: tran_fpa_sel(st),
                press_cas_eng: tran_cas_eng(st),
                input_alt    : tran_input_alt(st),
                input_fpa    : tran_input_fpa(st),
                input_cas    : tran_input_cas(st),
                alt_reached  : tran_alt_reached(st),
                fpa_reached  : tran_fpa_reached(st),
                alt_gets_near: tran_alt_gets_near(st)
        ENDCASES
```

The `CASES` statement is equivalent to an `IF-THEN-ELSIF-ELSE` construct. For example, if the event is `press_fpa_sel` then `nextstate(st,event)` = `tran_fpa_sel(st)`. The next step is to define each of these subfunctions.

2.5.1 Specifying the `att_cws` Mode

The `tran_att_cws` function describes what happens to the system when the pilot presses the `att_cws` button. This must be specified in a manner that covers all possible states of the system. According to the English specification, the action of pressing this button attempts to engage this mode if it is `off`. Changing the `att_cws` field to `engaged` is specified as follows:

```
st WITH [att_cws := engaged].
```

The `WITH` statement is used to alter a record in PVS. This expression produces a new record that is identical to the original record `st` in every field except `att_cws`. Of course, this is not all that happens in the system. The English specification also states that, *"Only one of the [* `att_cws`, `fpa_sel`, *or* `cas_eng`*] modes can be engaged at any time" (1).* Thus, the other modes must become something other than `engaged`. It is assumed that this means they are turned off. This would be indicated as:

```
st WITH [att_cws := engaged, fpa_sel := off,
         alt_eng := off].
```

The English specification also states that when a mode is disengaged, *"...the display reverts to showing the 'current' value" (2)*:

```
st WITH [att_cws := engaged, fpa_sel := off,
         alt_eng := off, alt_disp := current,
         fpa_disp := current].
```

Also, the English specification says that, *"...if the attitude control wheel steering button is pressed while the attitude control wheel steering mode is already engaged, the button is ignored" (5).* Thus, the full definition is:

```
tran_att_cws(st): states =
        IF att_cws(st) = off THEN
                st WITH [att_cws := engaged, fpa_sel := off,
                         alt_eng := off, alt_disp := current,
                         fpa_disp := current]
        ELSE st %% IGNORE: state is not altered at all
        ENDIF
```

The formal specification has elaborated exactly what the new state will be. The English specification does not address what happens to the ALT display or the FPA display when they are pre-selected and the corresponding mode *is not* currently engaged. The English specification only covers the situation where the corresponding mode *is* engaged. However, the formal specification does explicitly indicate what will happen: the displays are returned to showing the current value. The process of developing a complete mathematical specification often leads to the discovery of ambiguities in the English specification. In a real system design process, the developer would communicate with the system designers to clarify what behaviors were intended for these cases. The clarifications from the system designers would then be incorporated into the formal specification and also clarified in the accompanying English system description. If, for example, the system designers responded that the pre-selected displays should only remain the same if the corresponding mode is `off`, the following specification would be appropriate:

```
tran_att_cws(st): states =
        IF att_cws(st) = off THEN
                st WITH [att_cws := engaged,  fpa_sel := off,
                         alt_eng := off,
                         alt_disp := IF alt_eng(st) = off THEN alt_disp(st)
                                     ELSE current ENDIF
                         fpa_disp := IF fpa_sel(st) = off THEN fpa_disp(st)
                                     ELSE current ENDIF
                        ]
        ELSE st %% IGNORE: state is not altered at all
        ENDIF
```

We realize that this situation will arise in several other situations as well. In particular, whenever a mode (other than `cas_eng`) becomes `engaged`, should any other "pre_selected" displays be changed to `current` or should they remain `pre_selected`? We decide to consult the system designers. They agreed that the displays should be returned to `current` and suggested that the following be added to the English specification

> *6. Whenever a mode other than* `cas_eng` *is engaged, all other pre-selected displays should be returned to* `current`.

We have labeled this as requirement 6.

2.5.2 Specifying the cas_eng Mode

The `tran_cas_eng` function describes what happens to the system when the pilot presses the `cas_eng` button. This is the easiest mode to specify because its behavior is completely independent of the other modes. Pressing the `cas_eng` button merely toggles this mode on and off. The complete function is:

```
tran_cas_eng(st): states =
        IF cas_eng(st) = off THEN
           st WITH [cas_eng := engaged]
        ELSE
           st WITH [cas_eng := off, disp_cas := current]
        ENDIF
```

This specification states that if the `cas_eng` mode is currently `off`, pressing the button will engage the mode. If the mode is `engaged`, pressing the button will turn the mode `off`. Thus, the button acts like a switch. When `cas_eng` is disengaged the corresponding display, `disp_cas`, is returned to `current`.

2.5.3 Specifying the `fpa_sel` Mode

The `tran_fpa_sel` function describes the behavior of the system when the `fpa_sel` button is pressed. The English specification states that this mode *"need not be pre-selected (4)"*. Thus, whether the FPA display is pre-selected or not the outcome is the same:

```
IF fpa_sel(st) = off THEN
   st WITH [fpa_sel := engaged, att_cws := off,
            alt_eng := off, alt_disp := current]
```

Note that the `fpa_sel` mode engaged and the `att_cws` and `alt_eng` are turned off as well. This was included because the English specification states that, *"Engaging any of the first three modes will automatically cause the other two to be disengaged" (1).* Also note that this specification indicates that `alt_disp` is set to `current` because the `alt_eng` mode has been disengaged. This was done because the English specification states that, *"Once the target value is achieved or the mode is disengaged, the display reverts to showing the 'current' value" (2).* If the `alt_eng` mode is not currently active, the `WITH` update does not actually change the value, but merely updates that attribute to the value it already holds.

Since PVS requires that functions be completely defined, we must also cover the case where `fpa_sel` is already engaged. We consult the English specification and find *The calibrated air speed and flight-path angle buttons toggle on and off every time they are pressed. (5).* We interpret this to mean that if the `fpa_sel` button is pressed while it is currently `engaged`, the mode will be turned `off`. This is specified as follows:

```
st WITH [fpa_sel := off, fpa_disp := current]
```

Because the mode is disengaged, the corresponding display is returned to `current`. We realize that we also must cover the situation where the `alt_eng` mode is `armed` and the `fpa_sel` is engaged. In fact, section (3) of the English specification indicates that this will occur when one presses the `alt_eng` button and the airplane is far away from the pre-selected altitude. However, Section (3) does not tell is whether the disengagement of `fpa_sel` will also disengage the armed `alt_eng` mode. We decide to consult the system designers. They inform us that pressing the `fpa_sel` button should turn off both the `fpa_sel` and `alt_eng` mode in this situation. Thus, we modify the state update statement as follows:

```
st WITH [fpa_sel := off,  alt_eng := off,
         fpa_disp := current, alt_disp := current]
```

The complete specification is thus:

```
IF fpa_sel(st) = off THEN
    st WITH [fpa_sel := engaged, att_cws := off,
             alt_eng := off, alt_disp := current]
 ELSE
    st WITH [fpa_sel := off, fpa_disp := current,
             alt_eng := off, alt_disp := current]
ENDIF
```

The perspicacious reader may have noticed that there is a mistake in this formal specification. The rest of us will discover it when a proof is attempted using a theorem prover in the later section entitled, "Formal Verification of the Example System."

2.5.4 Specifying the `alt_eng` Mode

The `alt_eng` mode is used to capture a specified altitude and hold it. This is clearly the most difficult of the four to specify since it has a complicated interaction with the `fpa_sel` mode.

The English specification states that, *"The altitude must be pre-selected before the altitude engage button is pressed" (4)*. This is interpreted to mean that the command is simply ignored if an altitude has not been pre-selected. Consequently, the specification of `tran_alt_eng` begins:

```
tran_alt_eng(st): states =
   IF alt_disp(st) = pre_selected THEN
      ...
   ELSE % IGNORE command
   ENDIF
```

This specifies that the system state will change as a result of pressing the `alt_eng` button only if the `alt_disp` is `pre_selected`.

We must now proceed to specify the behavior when the `IF` expression is true. The English specification indicates that if the aircraft is more than 1,200 feet from the target altitude, this request will be put on hold (the mode is said to be armed) and the `fpa_sel` mode will be `engaged` instead. The English specification also says that, *"The pilot must then dial in a flight-path angle at this point" (3)*. The question arises whether the `fpa_sel` engagement should be delayed until this is done. Another part of the English specification offers a clue, *"The calibrated air speed and flight-path angle values need not be pre-selected before the corresponding modes are engaged" (4)*. Although this specifically addresses the case of pressing the `fpa_sel` button and not the situation where the `alt_eng` button indirectly turns this mode on, we suspect that the behavior is the same. Nevertheless, we decide to check with the system designers to make sure. The system designers explain that this is the correct interpretation and that this is the reason the mode is called "flight-path angle select" rather than "flight-path angle engage."

The behavior must be specified for the two situations: when the airplane is near the target and when it is not. There are several ways to specify this behavior. One way is for the state specification to contain the current altitude in addition to the target altitude. This could be included in the state vector as two numbers:

```
target_altitude: number
actual_altitude: number
```

The first number contains the value dialed in and the second the value last measured by a sensor. The specification would then contain:

```
IF abs(target_altitude - actual_altitude) > 1200 THEN
```

where `abs` is the absolute value function. If the behavior of the mode-control panel were dependent upon the target and actual altitudes in a multitude of ways, this would probably be the proper approach. However, in the example system the behavior is only dependent upon the relation of the two values to each other. Therefore, another way to specify this behavior is by abstracting away the details of the particular values and only storing information about their relative values in the state descriptor record. In particular, the altitude field of the state record can take on one of the following three values:

`away`	the pre-selected value is $>$ 1,200 feet away
`near_pre_selected`	the pre-selected value is $<=$ 1,200 feet away
`at_pre_selected`	the pre-selected value $=$ the actual altitude

The two different situations can then be distinguished as follows:

```
IF altitude(st) /= away THEN
```

When the value is not `away`, the `alt_eng` mode is immediately engaged. This is specified as follows:

```
IF altitude(st) /= away THEN
   st WITH [alt_eng := engaged, att_cws := off,
            fpa_sel := off, fpa_disp := current]
```

Note that not only is the `alt_eng` mode engaged, but this specification indicates that several other modes are affected as well. This was done because the English specification states that, *"Engaging any of the first three modes will automatically cause the other two to be disengaged" (1).* Thus, both `att_cws` and `fpa_sel` are turned off when `alt_eng` is `engaged`. This specification also returns the FPA display to `current` as required in the English specification *"Once the target value is achieved or the mode is disengaged, the display reverts to showing the 'current' value." (2).*

Now the behavior of the system must be specified for the other situation, when the aircraft is away from the target altitude. In this case `fpa_sel` is `engaged` and `alt_eng` is `armed`:

```
ELSE
   st WITH [fpa_sel := engaged, att_cws := off,
            alt_eng := armed]
```

As before, the `att_cws` mode is also turned off.

So far we have not considered whether the behavior of the system should be different if the `alt_eng` mode is already `armed` or `engaged` The English specification states that, *"Pressing the altitude engage button while the system is already in altitude engage mode has no effect" (5).* However, there is no information about what will happen if the mode is `armed`. Once again, the system designers are consulted and we are told that the mode-control panel should ignore the request in this case as well. Thus, the first `IF` expression should be augmented to include a test of `alt_eng(st) = off`:

```
IF alt_eng(st) = off AND alt_disp(st) = pre_selected THEN
```

and the complete specification of `tran_alt_eng` becomes:

```
tran_alt_eng(st): states =
   IF alt_eng(st) = off AND alt_disp(st) = pre_selected THEN
      IF altitude(st) /= away THEN  %% ENGAGED
         st WITH [att_cws := off, fpa_sel := off, alt_eng := engaged,
                  fpa_disp := current]
      ELSE                          %% ARMED
         st WITH [att_cws := off, fpa_sel := engaged, alt_eng := armed]
      ENDIF
   ELSE
      st %% IGNORE request
   ENDIF
```

Note that the last `ELSE` takes care of both the `armed` and `engaged` cases.

2.5.5 Input To Displays

The next three events that can occur in the system are `input_alt`, `input_fpa`, and `input_cas`. These occur when the pilot dials a value into one of the displays. The `input_alt` event corresponds to the subfunction of `nextstate` named `tran_input_alt`. The obvious thing to do is to set the appropriate fields as follows:

```
st WITH [alt_disp := pre_selected]
```

This is certainly appropriate when `alt_eng` is `off`. However, we must carefully consider the two cases: (1) when the `alt_eng` mode is `armed` and (2) when it is `engaged`. In this case, the pilot is *changing* the target value after the `alt_eng` button has been pressed. The English specification required that the `alt_eng` mode be `pre_selected` before it could become `engaged`, but did not rule out the possibility that the pilot could change the target value once it was `armed` or `engaged`. We consult the system designers once again. They inform us that entering a new target altitude value should return the `alt_eng` mode to `off` and the pilot must press the `alt_eng` button again to re-engage the mode. We add the following to the English specification:

> *7. If the pilot dials in a new altitude while the* `alt_eng` *button is already engaged or armed, then the* `alt_eng` *mode is disengaged and the* `att_cws` *mode is engaged. If the* `alt_eng` *mode was armed then the* `fpa_sel` *should be disengaged as well.*

The reason given by the system designers was that they didn't want the altitude dial to be able to automatically trigger a new active engagement altitude. They believed it was safer to force the pilot to press the `alt_eng` button again in order to change the target altitude.

Thus, the specification of `tran_input_alt` is:

```
tran_input_alt(st): states =
    IF alt_eng(st) = off THEN
          st WITH [alt_disp := pre_selected]
    ELSIF alt_eng(st) = armed OR alt_eng(st) = engaged THEN
          st WITH [alt_eng := off, alt_disp := pre_selected,
                   att_cws := engaged,
                   fpa_sel := off, fpa_disp := current]
    ELSE st %%  no change needed already preselected
    ENDIF
```

The other input event functions are similar:

```
tran_input_fpa(st): states =
   IF fpa_sel(st) = off THEN st WITH [fpa_disp := pre_selected]
   ELSE st
   ENDIF

tran_input_cas(st): states =
   IF cas_eng(st) = off THEN st WITH [cas_disp := pre_selected]
   ELSE st
   ENDIF
```

2.5.6 Other Actions

There are other events that are not initiated by the pilot but that still affect the mode-control panel: changes in the sensor input values. As described previously, rather than including the specific values of the altitude sensor, the state descriptor only records which of the following is true of the pre-selected altitude value:

`away`	`the pre_selected value is > 1,200 feet away`
`near_pre_selected`	`the pre_selected value is <= 1,200 feet away`
`at_pre_selected`	`the pre_selected value = the actual altitude`

Events must be defined that correspond to significant changes in the altitude so as to affect the value of this field in the state. Three such events affect the behavior of the panel:

`alt_gets_near`	`the altitude is now near the pre_selected value but not equal`
`alt_reached`	`the altitude reaches the pre_selected value`
`alt_gets_away`	`the altitude is no longer near the pre_selected value`

The transition subfunction associated with the first event must consider the case where the `alt_eng` mode is `armed` because the English spec states that *" The flight-path angle select mode will remain engaged until the aircraft is within 1,200 feet of the desired altitude, then the altitude engage mode is automatically engaged." (3)*

```
tran_alt_gets_near(st): states =
   IF alt_eng(st) = armed THEN
      st WITH [altitude := near_pre_selected,
               alt_eng :=  engaged,
               fpa_sel := off, fpa_disp := current]
   ELSE
      st WITH [altitude:= near_pre_selected]
   ENDIF
```

The subfunction associated with second event is similar, because we can't rule out the possibility that the event `alt_reached` may occur without `alt_gets_near` occurring first:

```
tran_alt_reached(st): states =
  IF alt_eng(st) = armed THEN
     st WITH [altitude := at_pre_selected, alt_disp := current,
              alt_eng :=  engaged, fpa_sel := off, fpa_disp := current]
  ELSE
     st WITH [altitude:= at_pre_selected, alt_disp := current]
  ENDIF
```

Note that in this case, the `alt_disp` field is returned to `current` because the English specification states, *"Once the target value is achieved or the mode is disengaged, the display reverts to showing the 'current' value." (2).*

The third event (i.e. `alt_gets_away`) is problematic in some situations. If the `alt_eng` mode is `engaged`, is it even possible for this event to occur? The flight-control system is actively holding the altitude of the airplane at the pre-selected value. Thus, unless there is some major external event such as a wind-shear phenomenon, this should never occur. Of course, a real system should be able to accommodate such unexpected events. However, to shorten this example, it will be assumed that such an event is impossible, and thus is not included in the specification as a possible event[1].

There is one other event corresponding to the flight-path angle being reached `fpa_reached`. This event has no impact on the behavior of the panel other than changing the status of the FPA display.

```
tran_fpa_reached(st): states = st WITH [fpa_disp := current]
```

2.6 Initial State

The formal specification must include a description of the state of the system when the mode-control panel is first powered on. One way to do this would be to define a particular constant, say `st0`, that represents the initial state:

```
st0: states = (# att_cws := engaged, cas_eng := off,
                 fpa_sel := off, alt_eng := off,
                 alt_disp := current, fpa_disp := current,
                 cas_disp := current, altitude := away #)
```

Alternatively, one could define a predicate (i.e., a function that returns true or false) that indicates when a state is equivalent to the initial state:

```
is_initial(st): bool =
att_cws(st) = engaged AND cas_eng(st) = off AND fpa_sel(st) = off
              AND alt_eng(st) = off AND alt_disp(st) = current AND
              AND fpa_disp(st) = current AND cas_disp(st) = current
```

Note that this predicate does not specify that the altitude field must have a particular value (e.g. `away`) . Thus, this predicate defines an equivalence class of states, not all identical, in which the system could be initially. This is the more realistic way to specify the initial state since it does not designate any particular "altitude" value.

2.7 Formal Verification of the Example System

The formal specification of the mode-control panel is complete. But how does the system developer know that the specification is correct? Unlike the English specification, the formal specification is known to be detailed and precise. But it could be *unambiguously wrong*. Since this is a requirements specification, there is no higher-level specification against which to prove this one. Therefore, ultimately the developer must rely on human inspection to insure that the formal specification is "what was intended." Nevertheless, the specification can be analyzed in a formal way. In particular, the developer can postulate properties that he believes should be true about the system and attempt to prove that the formal specification satisfies these properties. This process serves to reinforce the belief that the specification is what was intended. If the specification cannot be proved to meet the desired properties, the problem in the specification must be found or the property must be modified

[1]The situation where the `alt_eng` mode is not `engaged` or `armed` is not difficult to specify, but also does not add anything pedagogically significant to the specification.

until the proof can be completed. In either case, the developer's understanding of and confidence in the system is increased.

In the English specification of the mode-control panel, there were several statements made that characterize the overall behavior of the system. For example, *"One of the three modes [*`att_cws`*,* `fpa_sel`*, or* `alt_eng`*] should be* `engaged` *at all times" (1).* This statement can be formalized, and it can be proved that no matter what sequence of events occurs, this will remain true of the system. Properties such as this are often called system invariants. This particular property is formalized as follows:

```
att_cws(st) = engaged
OR fpa_sel(st) = engaged
OR alt_eng(st) = engaged
```

Another system invariant can be derived from the English specification: *"Only one of the first three modes [*`att_cws`*,* `fpa_sel`*,* `alt_eng`*] can be engaged at any time" (1).* This can be specified in several ways. One possible way is as follows:

```
(alt_eng(st) /= engaged OR fpa_sel(st) /= engaged)
AND (att_cws(st) = engaged IMPLIES
          alt_eng(st) /= engaged AND fpa_sel(st) /= engaged)
```

Finally, it would be prudent to insure that whenever `alt_eng` is `armed`, that `fpa_sel` is `engaged`:

```
(alt_eng(st) = armed IMPLIES fpa_sel(st) = engaged).
```

All three of these properties can be captured in one predicate (i.e. a function that is true or false) as follows:

```
valid_state(st): bool =
                (att_cws(st) = engaged
                OR fpa_sel(st) = engaged
                OR alt_eng(st) = engaged)
        AND (alt_eng(st) /= engaged OR fpa_sel(st) /= engaged)
        AND (att_cws(st) = engaged IMPLIES
                alt_eng(st) /= engaged AND fpa_sel(st) /= engaged)
        AND (alt_eng(st) =armed IMPLIES fpa_sel(st) = engaged)
```

The next step is to prove that this is always true of the system. One way to do this is to prove that the initial state of the system is valid, and that if the system is in a valid state before an event, then it is in a valid state after an event, no matter what event occurs. In other words, we must prove the following two theorems:

```
initial_good: THEOREM is_initial(st) IMPLIES valid_state(st)

nextstate_good: THEOREM valid_state(st) IMPLIES
                              valid_state(nextstate(st,event))
```

These two theorems effectively prove by induction that the system can never enter a state that is not valid. Both of these theorems are proved by the single PVS command `GRIND`. The PVS system replays the proofs in 192 secs. on a Sparc 20 with 32 Megabytes of memory. The following proof reduces the execution time to 57.0 secs:

```
(""
 (SKOSIMP*)
 (EXPAND "nextstate")
 (LIFT-IF)
 (GROUND)
 (("1" (HIDE -1) (GRIND))
  ("2" (HIDE -1 2) (GRIND))
  ("3" (HIDE -1 2 3) (GRIND))
  ("4" (HIDE -1 2 3 4) (GRIND))
  ("5" (HIDE -1 2 3 4 5) (GRIND))
  ("6" (HIDE -1 2 3 4 5 6) (GRIND))
  ("7" (HIDE -1 2 3 4 5 6 7) (GRIND))
  ("8" (HIDE -1 2 3 4 5 6 7 8) (GRIND))
  ("9" (HIDE -1 2 3 4 5 6 7 8 9) (GRIND))
  ("10" (HIDE 1 3 4 5 6 7 8 9 10) (GRIND))))
```

As mentioned earlier, the specification of `fpa_sel` contains an error. On the attempt to prove the `nextstate_good` theorem on the erroneous version of `fpa_sel` described earlier, the prover stops with the following sequent:

```
nextstate_good :

-1   fpa_sel(st!1) = engaged
-2   press_fpa_sel?(event!1)
  |-------
1   att_cws(st!1) = engaged
2   alt_eng(st!1) = engaged
3   press_att_cws?(event!1)
4   press_alt_eng?(event!1)
```

The basic idea of a sequent is that one must prove that one of the statements after the `|-------` is provable from the statements before it. In other words, one must prove:

```
        -1 AND -2 ===> 1 OR 2 OR 3 OR 4
```

Immediately we see that formulas {3} and {4} are impossible, because `press_fpa_sel?(event!1)` tells us that `event!1 = press_fpa_sel` and not `press_att_cws` or `press_alt_eng`. Thus, we must establish {1} or {2}. However, this is impossible. But, there is nothing in this sequent to require that `att_cws(st!1) = engaged` or `alt_eng(st!1) = engaged`. Thus, it is obvious at this point that something is wrong with the specification or the proof. It is clear that the difficulty surrounds the case when the event `press_fpa_sel` occurs, so we examine `tran_fpa_sel` more closely. We realize that the specification should have set `att_cws` to `engaged` as well as turning off the `fpa_sel` mode and `alt_eng` mode:

```
  tran_fpa_sel(st): states =
     IF fpa_sel(st) = off THEN
         st WITH [fpa_sel := engaged, att_cws := off, alt_eng := off,
                  alt_disp := current]
      ELSE
         st WITH [fpa_sel := off, fpa_disp := current,
                  att_cws := engaged,
                  alt_eng := off, alt_disp := current]
     ENDIF
```

This modification is necessary because otherwise the system could end up in a state where no mode was currently active. After making the correction, the proof succeeds.

It should be pointed out that the predicate `good` must be sufficiently strong to make the induction go through. Alternatively one can prove theorems of the form:

```
            reachable(st) IMPLIES good(st)
```

where `reachable(st)` is a predicate that delineates all states that are reachable from the initial state. The predicate `reachable` is typically recursively defined, so the theorem in this form must be proved by using induction explicitly.

3 A Different Decomposition

Many systems can be specified using the state-machine method illustrated in this chapter. However, as the state-machine becomes complex, the specification of the state transition functions can become exceedingly complex. Therefore, many different approaches have been developed to define the state machine in a manner that is convenient for different applications. Some of the more widely known are decision tables, decision trees, state-transition diagrams, state-transition matrices, Statecharts, Superstate and R-nets [7].

Although these methods effectively accomplish the same thing—the delineation of the state machine—they vary greatly in their input format. Some use graphical notation, some use tables, and others use language constructs. Some industries have used table-oriented methods because they believe that they are more readable for specifications that require a large number of pages. An interesting new approach is being developed by the Naval Research Lab called the Software Cost Reduction (SCR) method [5]. In the SCR method, a system is a collection of state machines (called mode classes) operating in parallel. The states of the machines are called modes. The transition of these state machines can be triggered by mode changes in other machines or by a change in an input variable.

Some of the above methods decompose the specification by mode rather than by event. This approach may be easier to comprehend and thus easier to validate by human inspection. To explore this possibility, the auto-pilot will be re-specified in a manner that treats each mode in isolation. Rather than decompose the state-transition function into subfunctions according to the triggering event, we will decompose the function into subfunctions that describe each of the state components (i.e., buttons, displays, etc.) in isolation from each other:

```
      NEXTSTATE(st,event): states =
               st WITH [att_cws := NEXT_ATT_CWS(st,event),
                        cas_eng := NEXT_CAS_ENG(st,event),
                        fpa_sel := NEXT_FPA_SEL(st,event),
                        alt_eng := NEXT_ALT_ENG(st,event),
                        cas_disp := NEXT_CAS_DISP(st,event),
                        fpa_disp := NEXT_FPA_DISP(st,event),
                        alt_disp := NEXT_ALT_DISP(st,event),
                        altitude := NEXT_ALTITUDE(st,event)]
```

Each of these subfunctions are then defined by a table. For example, the `att_cws` mode is defined as follows:

att_cws	event	new mode
off	press_att_cws	engaged
	press_fpa_sel WHEN fpa_sel(s) = engaged	engaged
	press_alt_eng WHEN alt_eng(s) = armed	engaged
	press_alt_eng WHEN alt_eng(s) = engaged	engaged
	ALL others	off
engaged	press_fpa_sel WHEN fpa_sel(s) = engaged	off
	press_alt_eng WHEN alt_disp(s) = pre	
	AND alt_eng(s) = off	off
	ALL others	engaged

The first column lists all possible states of `att_cws` prior to the occurrence of an event. The second column delineates all of the events which change the state of the `att_cws` mode. The event can be simple as in the first row (i.e. `press_att_cws`) or it may be restricted to trigger a transition only when the state satisfies the predicate following the `WHEN` keyword. The third column lists the new state of the `att_cws` mode after the occurrence of the event.

A still more compact table can be constructed using the PVS table syntax. Each of the triggering events are listed as column headers:

The NEXT_ATT_CWS Mode

	press_att_cws	press_fpa_sel	press_alt_eng	input_alt	ELSE
off	engaged	Pfpa(st)	off	Ialt(st)	off
engaged	engaged	Pfpa(st)	Palt(st)	engaged	engaged

where

```
Palt(st) = IF alt_eng(st) = off AND alt_disp(st) = pre_selected
           THEN off ELSE engaged ENDIF,
Pfpa(st) = IF fpa_sel(st) = engaged THEN engaged ELSE off ENDIF,
Ialt(st) = IF alt_eng(st) = armed OR alt_eng(st) = engaged THEN engaged
           ELSE att_cws(st) ENDIF
```

The special function definitions are necessary when the behavior is dependent upon factors other than the current event and the current value of the mode being defined. For example, when the `att_cws` behavior is dependent upon the current value of the `alt_eng` mode. In fact wherever there is a `WHEN` clause, a special subfunction will have to be defined. Although this approach is not as readable as the previous table, it is formalizable within PVS.

To make the tables more compact and usable, the following name changes will be made:

`old name`	new name
pre_selected	pre
armed	arm
engaged	eng
near_pre_selected	near
at_pre_selected	at
press_att_cws	P_cws
press_cas_eng	P_cas
press_fpa_sel	P_fpa
press_alt_eng	P_alt
input_alt	I_alt
input_fpa	I_fpa
input_cas	I_cas
alt_reached	alt_re
alt_gets_near	alt_ne
fpa_reached	fpa_re

3.1 Re-specifying the Autopilot in a Mode Decomposition Style

Using this approach, all of the modes can be defined:

The NEXT_ATT_CWS Mode

	P_cws	P_fpa	P_alt	I_alt	ELSE
off	eng	Pfpa(st)	off	Ialt(st)	off
eng	eng	Pfpa(st)	Palt(st)	eng	eng

where

```
Palt(st) = IF alt_eng(st) = off AND alt_disp(st) = pre_sel THEN off ENDIF,
Pfpa(st) = IF fpa_sel(st) = eng THEN eng ELSE off ENDIF,
Ialt(st) = IF alt_eng(st) = arm OR alt_eng(st) = eng THEN eng
           ELSE att_cws(st) ENDIF
```

The NEXT_CAS_ENG Mode

	P_cas	I_cas	ELSE
off	eng	off	off
eng	off	eng	eng

The NEXT_FPA_SEL Mode

	P_cws	P_fpa	P_alt	I_alt	I_fpa	alt_re	alt_ne	ELSE
off	off	eng	Palt(st)	off	off	off	off	off
eng	Pcws(st)	off	Palt(st)	Ialt(st)	eng	altr(st)	altr(st)	eng

where

```
Pcws(st) = IF att_cws(st) = off THEN off ELSE fpa_sel(st) ENDIF,
Palt(st) = IF alt_eng(st) = off AND alt_disp(st) = pre_sel THEN
              IF altitude(st) (st)/= away THEN off
              ELSE eng ENDIF,
           ELSE fpa_sel(st) ENDIF,
Ialt(st) = IF  alt_eng(st) = arm OR alt_eng(st) = eng THEN off
           ELSE fpa_sel(st) ENDIF,
altr(st) = IF alt_eng(st) = arm THEN off
           ELSE fpa_sel(st) ENDIF
```

The NEXT_ALT_ENG Mode

	P_cws	P_fpa	P_alt	I_alt	ELSE
off	off	off	Palt(st)	off	off
arm	Pcws(st)	off	arm	off	els(st)
eng	Pcws(st)	off	eng	off	eng

where

```
Palt(st) =  IF alt_disp(st) = pre_sel THEN
               IF altitude(st) /= away THEN eng ELSE arm ENDIF
            ELSE alt_eng(st) ENDIF),
els(st)  = IF event = alt_ne OR event = alt_re THEN eng
           ELSE arm ENDIF,
Pcws(st) = IF att_cws(st) = off THEN off ELSE alt_eng(st) ENDIF
```

The NEXT_CAS_DISP Mode

	P_cas	I_cas	ELSE
current	current	Icas(st)	current
pre_sel	Pcas(st)	Icas(st)	pre_sel

where

```
Pcas(st) = IF cas_eng(st) = eng THEN current
           ELSE cas_disp(st) ENDIF,
Icas(st) = IF cas_eng(st) = off THEN  pre_sel
           ELSE cas_disp(st) ENDIF
```

The NEXT_FPA_DISP Mode

	P_cws	P_fpa	P_alt	I_fpa	fpa_re	I_alt	alt_re	alt_ne	ELSE
current	current	current	current	Ifpa(st)	current	current	current	current	current
pre_sel	Pcws(st)	Pfpa(st)	Palt(st)	Ifpa(st)	current	Ialt(st)	alre(st)	alre(st)	pre_sel

where

```
        Pcws(st) = IF att_cws(st) = off THEN current
                   ELSE fpa_disp(st) ENDIF,
        Pfpa(st) = IF fpa_sel(st) = eng THEN current
                   ELSE fpa_disp(st) ENDIF,
        Palt(st) = IF alt_eng(st) = off AND alt_disp(st) = pre_sel AND
                      altitude(st) /= away THEN current
                   ELSE fpa_disp(st) ENDIF,
        Ifpa(st) = IF fpa_sel(st) = off THEN pre_sel
                   ELSE fpa_disp(st) ENDIF,
        Ialt(st) = IF alt_eng(st) = arm OR alt_eng(st) = eng THEN
                   current ELSE fpa_disp(st) ENDIF,
        alre(st) = IF alt_eng(st) = arm THEN current
                   ELSE fpa_disp(st) ENDIF
```

The NEXT_ALT_DISP Mode

	P_cws	P_fpa	I_alt	alt_re	ELSE
current	current	current	Ialt(st)	current	current
pre_sel	Pcws(st)	current	pre_sel	current	pre_sel

where

```
        Pcws(st) = IF att_cws(st) = off THEN current ELSE alt_disp(st) ENDIF,
        Ialt(st) = IF alt_eng(st) = off THEN pre_sel
                   ELSIF alt_eng(st)(st) = pre THEN alt_disp(st)
                   ELSE pre_sel ENDIF
```

The NEXT_ALTITUDE Mode

	alt_ne	alt_re	ELSE
away	near	at	away
near	near	at	near
at	near	at	at

This method of specifying the autopilot can be shown to be equivalent to the former method, as follows:

```
        test: LEMMA nextstate(st,event) = NEXTSTATE(st,event)
```

3.2 Some Observations About Mode Decomposition

The second method of specifying the autopilot has the advantage that each of the input buttons and displays are specified separately. Although conceptually this is a reasonable thing to do, it resulted in a more complex formulation for this example problem. This approach would work nicely where the interactions between the different modes (buttons and displays) were minimal. However, in this example, where the interactions were extensive, the resulting tables were complex. The presence of the extra functions such as `Pfpa` and `Ialt` clearly reveal these interactions. One could argue that it is a good thing for interactions such as these to clutter up the specification because it will serve as deterrent. But where such interactions are inevitable, this approach may lead to a more complex presentation.

Another reason that this system may not lend itself well to the mode decomposition style is that it is largely driven by external commands, i.e. from a pilot, rather that changes in variables of other state-machines. This type of system is naturally described by enumerating the effect of a pilot input (such as pressing the `att_cws` button) on *all* of the different parts of the system state in one place:

```
tran_att_cws(st): states =
        IF att_cws(st) = off THEN
                st WITH [att_cws := engaged, fpa_sel := off,
                         alt_eng := off,alt_disp := current,
                         fpa_disp := current]
        ELSE st %% IGNORE: state is not altered at all
        ENDIF
```

The mode-decomposition style approach specifies the result of a pilot input over many tables. For example, to understand the effect of the pilot pressing the `P_cws` button, every table containing a column labeled `P_cws` must be consulted. This includes the tables for `NEXT_ATT_CWS`, `NEXT_FPA_SEL`, `NEXT_ALT_ENG`, `NEXT_FPA_DISP` and `NEXT_ALT_DISP`. Also note that for each of these tables except `NEXT_ATT_CWS`, a special function `Pcws` had to be defined, because the behavior is dependent upon the current value of `att_cws(st)`. The mode-decomposition style specification is also larger—761 words rather than 373 words.

There is another factor that should be considered when making this comparison. Tables such as these can be used by system engineers and human factors engineers to develop the original requirements. We performed this exercise assuming that the requirements development phase had already been completed, and the formalists were merely trying to "capture" these pre-existing requirements. It may well turn out that the mode-decomposition style tables are more useful in this type of activity because the *expose* the complexity in a more *explicit* manner. Only the experience of applying these techniques to real systems will reveal the best approach.

It should also be noted that the conclusions of this section are based solely on the experience from this one example. It is that quite different conclusions will be reached for other problems.

4 Some More Analysis

The previous verification section made the observation that it is necessary that the predicate `good` be sufficiently strong in order for the proof to go through. To see this, we will weaken the predicate as follows:

```
good(st): bool = (att_cws(st) = engaged OR fpa_sel(st) = engaged
                      OR alt_eng(st) = engaged) AND
                 (alt_eng(st) /= engaged OR fpa_sel(st) /= engaged) AND
                 (att_cws(st) = engaged IMPLIES
                     alt_eng(st) /= engaged AND fpa_sel(st) /= engaged)
```

This causes the proof of `nextstate_good`

```
nextstate_good: THEOREM good(st) IMPLIES good(nextstate(st,event))
```

to terminate as follows:

```
-1    armed?(alt_eng(st!1))
-2    engaged?(att_cws(st!1))
  |-------
1    engaged?(fpa_sel(st!1))
```

This sequent is concerned about an "unreachable" state, namely, a state where `att_cws` and `fpa_sel` are `engaged` and `alt_eng` is `armed`. The problem is that the predicate `good` is no longer filtering out the non-reachable states. This illustrates why this simple verification approach will only work when the predicate `good` is only true for states that are actually reachable from the initial state via a sequence of `nextstate` executions.

The following approach is more general:

```
n: VAR nat
ev: VAR events
pst: VAR states
reachable_in(n,st): RECURSIVE bool =
                        IF n = 0 THEN is_initial(st)
                        ELSE
                          (EXISTS pst,ev: st = nextstate(pst,ev) AND
                                          reachable_in(n-1,pst))
                        ENDIF MEASURE n

is_reachable(st): bool = (EXISTS n: reachable_in(n,st))

reachable_good: THEOREM is_reachable(st) IMPLIES good(st)
```

Using this approach, the predicate `good` can be any desired property and does not have to evaluate to false for non-reachable states. Sometimes a non-recursive expression (call it `reachable?`) can developed that logically follows from `is_reachable`.

```
reachable?(st): bool =  (att_cws(st) = engaged OR fpa_sel(st) = engaged
              OR alt_eng(st) = engaged) AND
                 (alt_eng(st) /= engaged OR fpa_sel(st) /= engaged) AND
                 (att_cws(st) = engaged IMPLIES
                     alt_eng(st) /= engaged AND fpa_sel(st) /= engaged) AND
                   (alt_eng(st) = armed IMPLIES fpa_sel(st) = engaged)

is_reach_implies: LEMMA is_reachable(st) IMPLIES reachable?(st)
```

Theorems of the form `is_reachable(st) IMPLIES good(st)` follow trivially from lemmas of the form `reachable? IMPLIES good(st)`.

Using our new approach, we can establish some additional properties about our specification in a straight-forward manner. Accordingly we scrutinize the English specification for some additional global properties to test the formal specification against. For example, the English specification "*Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value. (2)*", leads to the following theorems:

```
safety1: THEOREM reachable?(st) AND fpa_sel(st) = engaged AND
                  fpa_sel(nextstate(st,event)) = off IMPLIES
                     fpa_disp(nextstate(st,event)) = current

safety2: THEOREM reachable?(st) AND alt_eng(st) = engaged AND
                  event /= input_alt AND
                  alt_eng(nextstate(st,event)) = off IMPLIES
                     alt_disp(nextstate(st,event)) = current
```

Note that the `safety2` theorem needs the additional premise that `event /= input_alt`, because `input_alt` disengages the `alt_eng` mode but immediately preselects it again.

5 Other Methods

It would be interesting to compare the performance of model-checking techniques against the PVS command `GRIND` for problems such as these where there is only a single finite state machine. The `GRIND` command appears to be complete for such problems though I imagine that it will quickly be overcome by computational complexity as the problem size increases. Some of the more widely known methods are SMV [1], Murϕ [2] and COSPAN [13]. These require that the state-space of the machine be finite. Our example specification has a finite state space. However, if the values of pre-selected and measured altitude had not been abstracted away, the state-space would have been infinite.

6 Conclusions

The preceding discussion illustrates the process that one goes through in translating an English specification into a formal one. Although the example system was contrived to demonstrate this feature, the process demonstrated is typical for realistic systems, and the English specification for the example is actually more complete than most because the example system is small and simple.

The formal specification process forces one to clearly elaborate the behavior of a system in detail. Whereas the English specification must be examined in multiple places and interpreted in order to make a judgment about the desired system's behavior, the formal specification completely defines the behavior. Thus, the requirements capture process includes making choices about how to interpret the informal specification. Traditional software development practices force the developer to make these interpretation choices (consciously or unconsciously) during the process of creating the design or implementation. Many of the choices are hidden implicitly in the implementation without being carefully thought out or verified by the system designers, and the interpretations and clarifications are seldom faithfully recorded in the requirements document. On the other hand, the formal methods process exposes these ambiguities early in the design process and forces early and clear decisions, which are fully documented in the formal specification.

References

[1] Burch, J. R.; Clarke, E.M.; McMillan, K.L.; Dill, D.L.; L.J.; and Hwang: Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, vol. 98, no. 2, 1992, pp. 142–170.

[2] Burch, J. R.; and Dill, David L.: Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification, CAV '94*, Stanford, CA, June 1994, pp. 68–80.

[3] Butler, Ricky W.: *An Elementary Tutorial on Formal Specification and Verification Using PVS.* NASA Technical Memorandum 108991, Sept. 1993.

[4] Butler, Ricky W.; and Johnson, Sally C.: Formal Methods For Life-Critical Software. In *Computing in Aerospace 9 Conference*, San Diego, CA, Oct. 1993, pp. 319–329.

[5] Constance Heitmeyer, Carolyn Gasarch, Alan Bull; and Labaw, Bruce: SCR*: A Toolset for Specifying and Analyzing Requirements. In *Tenth Annual Conference on Computer Assurance (COMPASS 95)*, Gaithersburg, MD, June 1995, pp. 109–122.

[6] Crow, Judy; Owre, Sam; Rushby, John; Shankar, Natarajan; and Srivas, Mandayam: A Tutorial Introduction to PVS. In *WIFT'95 Workshop on Industrial-strength Formal Specification Techniques*, Boca Raton, Florida USA, Apr. 1995.

[7] Davis, Alan M.: A Comparison of Techniques for the Specification of External System Behavior. *Communications of the ACM*, vol. 31, no. 9, 1988, pp. 1098–1115.

[8] Owre, S.; Shankar, N.; and Rushby, J. M.: *The PVS Specification Language (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[9] Owre, S.; Shankar, N.; and Rushby, J. M.: *User Guide for the PVS Specification and Verification System (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[10] Owre, Sam; Rushby, John; ; Shankar, Natarajan; and von Henke, Friedrich: Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, vol. 21, no. 2, Feb. 1995, pp. 107–125.

[11] Owre, Sam; Rushby, John; Shankar, Natarajan; and Srivas, Mandayam: A Tutorial Using PVS For Hardware Verification. In *Second International Conference on Theorem Proving in Circuit Design, Theory, Practice, and Experience*, Bad Herrenalb, Germany, Sept. 1994.

[12] Owre, Sam; Rushby, John M.; and Shankar, Natarajan: PVS: A Prototype Verification System. In Kapur, Deepak, editor 1992:, *11th International Conference on Automated Deduction (CADE)*, vol. 607 of *Lecture Notes in Artificial Intelligence*, Saratoga, NY, June 1992, Springer Verlag, pp. 748–752.

[13] Sabnani, K.; Aggarwal, S.; and Kurshan, R. P.: A Calculus For Protocol Specification and Validation. In *Protocol Specification, Testing, and Verification III.* Elsevier Science Publishers B.V. (North Holland), 1983.

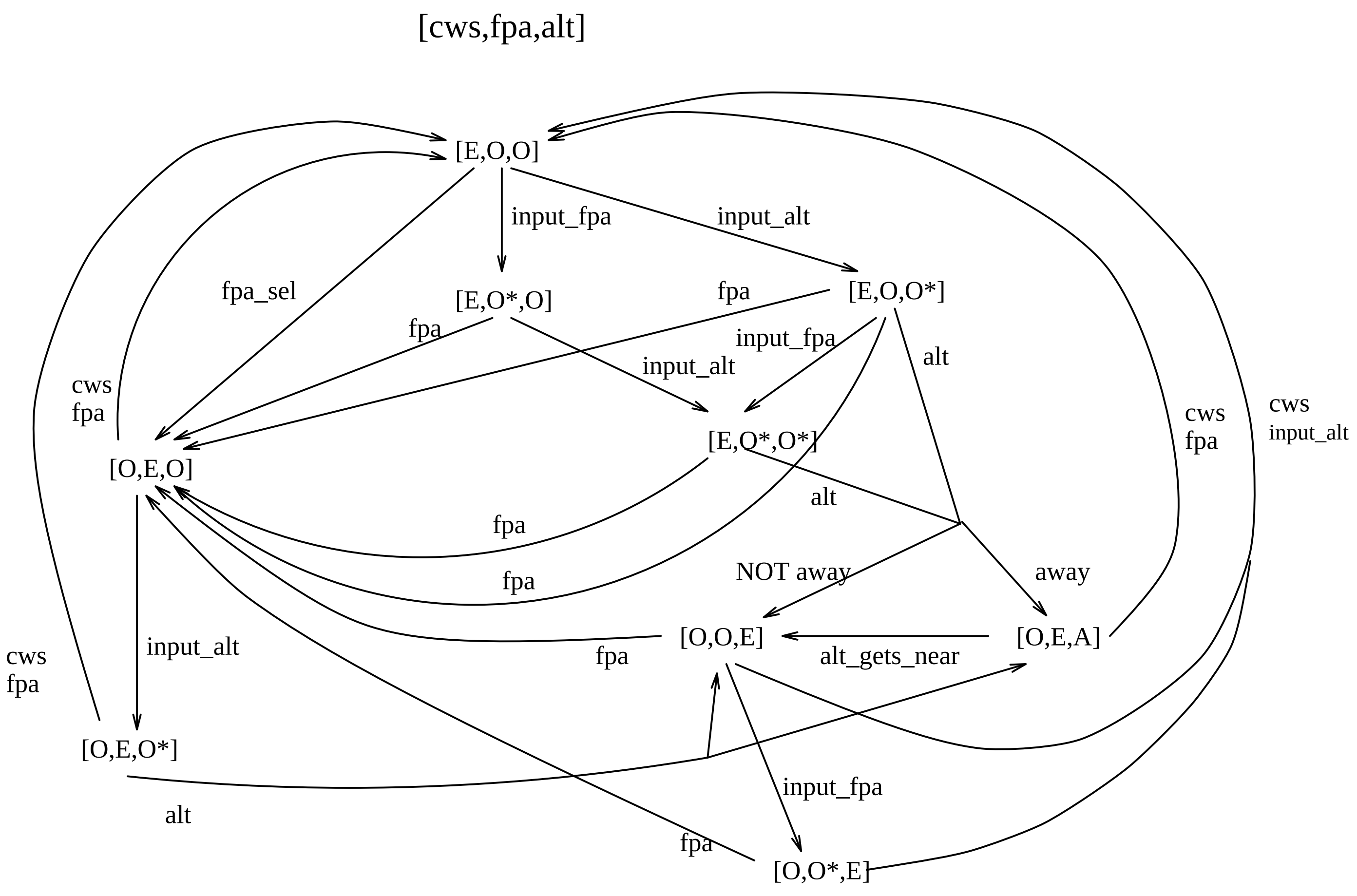
[14] Shankar, N.; Owre, S.; and Rushby, J. M.: *The PVS Proof Checker: A Reference Manual (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[15] Shankar, Natarajan: Verification of Real-Time Systems Using PVS. pp. 280–291.

[16] Shankar, Natarajan; Owre, Sam; and Rushby, John: *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

Appendix A: A Graphical View Of State Space

Figure 3 shows a picture of the state space of the auto-pilot with the `cas_eng` mode excluded. Note



NOOPS:	[E,O,O]: alt, cws	[O,E,O]: alt, input_fpa
	[E,O*,O]: cws, input_fpa, alt	[O,E,O*]: input_fpa, input_alt
	[E,O,O*]: cws, input_alt	[O,E,A]: alt, input_alt, input_fpa
	[E,O*,O*]: cws, input_fpa, input_alt	[O,O,E]: alt
		[O,O*,E]: alt, input_fpa, input_alt

Figure 3: State Space Diagram

that the state vector is [`att_cws,fpa_sel,alt_eng`] and

E	=	`engaged`
P	=	`pre-selected`
A	=	`armed`
O	=	`off`

For example, `[E,O,O]` is the state where `att_cws = engaged, fpa_sel = off` and `alt_eng = off`. An asterisk is placed next to the letter to indicate that the corresponding display has been preselected. For example, `[E,O*,O]` indicates that the `fpa_sel` mode is `off` and the FPA display has been pre-selected.

Appendix B: Complete Listing of Specification

In these section a complete listing of the specification is given for the Example System. This example can also be retrieved via the internet at

```
url: http://atb-www.larc.nasa.gov/fm.html
```

The listings are included here for easy reference.

```
defs: THEORY
BEGIN

  md_status: TYPE = off, armed, engaged

  off_eng: TYPE = md: md_status | md = off OR md =engaged

  disp_status: TYPE = pre_selected, current

  altitude_vals: TYPE = away, near_pre_selected, at_pre_selected

  states: TYPE =  [# % RECORD
                  att_cws: off_eng,
                  cas_eng: off_eng,
                  fpa_sel: off_eng,
                  alt_eng: md_status,
                  alt_disp: disp_status,
                  fpa_disp: disp_status,
                  cas_disp: disp_status,
                  altitude: altitude_vals
                  #]


  events: TYPE = press_att_cws, press_cas_eng, press_fpa_sel, press_alt_eng,
                  input_alt, input_fpa, input_cas, alt_reached,
                  alt_gets_near,  fpa_reached

  event: VAR events
  st: VAR states
  nextstate(event,st): states

END defs
```

```
tran: THEORY
BEGIN

  IMPORTING defs

  event: VAR events
  st: VAR states

  tran_att_cws(st): states =
     IF att_cws(st) = off THEN
        st WITH [att_cws := engaged, fpa_sel := off, alt_eng := off,
                 alt_disp := current, fpa_disp := current]
     ELSE st %% IGNORE
     ENDIF

  tran_cas_eng(st): states =
     IF cas_eng(st) = off THEN
        st WITH [cas_eng := engaged]
     ELSE
        st WITH [cas_eng := off, cas_disp := current]
     ENDIF

  tran_fpa_sel(st): states =
     IF fpa_sel(st) = off THEN
         st WITH [fpa_sel := engaged, att_cws := off, alt_eng := off,
                  alt_disp := current]
      ELSE
         st WITH [fpa_sel := off, fpa_disp := current,
                  att_cws := engaged,
                  alt_eng := off, alt_disp := current]
     ENDIF

  tran_alt_eng(st): states =
     IF alt_eng(st) = off AND alt_disp(st) = pre_selected THEN
        IF altitude(st) /= away THEN  %% ENG
           st WITH [att_cws := off, fpa_sel := off, alt_eng := engaged,
                    fpa_disp := current]
        ELSE                  %% ARM
           st WITH [att_cws := off, fpa_sel := engaged, alt_eng := armed]
        ENDIF
     ELSE
        st %% IGNORE request
     ENDIF

  tran_input_alt(st): states =
      IF alt_eng(st) = off THEN
            st WITH [alt_disp := pre_selected]
      ELSIF alt_eng(st) = armed OR alt_eng(st) = engaged THEN
            st WITH [alt_eng := off, alt_disp := pre_selected,
                     att_cws := engaged,
                     fpa_sel := off, fpa_disp := current]
      ELSE st %%  no change needed already preselected
      ENDIF

  tran_input_fpa(st): states =
      IF fpa_sel(st) = off THEN st WITH [fpa_disp := pre_selected]
      ELSE st
      ENDIF

   tran_input_cas(st): states =
      IF cas_eng(st) = off THEN st WITH [cas_disp := pre_selected]
      ELSE st
      ENDIF

   tran_alt_gets_near(st): states =
      IF alt_eng(st) = armed THEN
         st WITH [altitude := near_pre_selected,
                  alt_eng :=  engaged,
                  fpa_sel := off, fpa_disp := current]
      ELSE
         st WITH [altitude:= near_pre_selected]
      ENDIF

   tran_alt_reached(st): states =
     IF alt_eng(st) = armed THEN
        st WITH [altitude := at_pre_selected, alt_disp := current,
                 alt_eng :=  engaged, fpa_sel := off, fpa_disp := current]
     ELSE
        st WITH [altitude:= at_pre_selected, alt_disp := current]
     ENDIF

   tran_fpa_reached(st): states = st WITH [fpa_disp := current]

   nextstate(st,event): states =
       CASES event OF
          press_att_cws: tran_att_cws(st),
          press_alt_eng: tran_alt_eng(st),
          press_fpa_sel: tran_fpa_sel(st),
          press_cas_eng: tran_cas_eng(st),
          input_alt    : tran_input_alt(st),
          input_fpa    : tran_input_fpa(st),
          input_cas    : tran_input_cas(st),
          alt_reached  : tran_alt_reached(st),
          fpa_reached  : tran_fpa_reached(st),
          alt_gets_near: tran_alt_gets_near(st)
       ENDCASES


 END tran

 panel: THEORY
 BEGIN

   IMPORTING tran

   event: VAR events
   st: VAR states

   st0: states =  (#
                   att_cws := engaged,
                  cas_eng := off,
                  fpa_sel := off,
                  alt_eng := off,
                  alt_disp := current,
                  fpa_disp := current,
                  cas_disp := current,
                  altitude := away
                  #) ;

  good(st): bool =  (att_cws(st) = engaged OR fpa_sel(st) = engaged
                        OR alt_eng(st) = engaged) AND
                   (alt_eng(st) /= engaged OR fpa_sel(st) /= engaged) AND
                   (att_cws(st) = engaged IMPLIES
                       alt_eng(st) /= engaged AND fpa_sel(st) /= engaged) AND
                   (alt_eng(st) = armed IMPLIES fpa_sel(st) = engaged)


  is_initial(st): bool =        att_cws(st) = engaged
                            AND cas_eng(st) = off
                            AND fpa_sel(st) = off
                            AND alt_eng(st) = off
                            AND alt_disp(st) = current
                            AND fpa_disp(st) = current
                            AND cas_disp(st) = current

  st0_good: LEMMA good(st0)

  initial_good: THEOREM is_initial(st) IMPLIES good(st)

  nextstate_good: THEOREM good(st) IMPLIES good(nextstate(st,event))

  nextstate_good2: THEOREM good(st) IMPLIES good(nextstate(st,event))

END panel
```