21 Formal Methods

Sally C. Johnson
NASA Langley Research Center

Ricky W. Butler
NASA Langley Research Center

21.1 Introduction

With each new generation of aircraft, the requirements for digital avionics systems become increasingly complex, and their development and validation consumes an ever-increasing percentage of the total development cost of an aircraft. The introduction of life-critical avionics, where failure of the computer hardware or software can lead to loss of life, brings new challenges to avionics validation. The FAA recommends that catastrophic failures of the aircraft be "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" and suggests probabilities of failure on the order of 10^{-9} per flight hour [FAA, 1988].

There are two major reliability factors to be addressed in the design of ultrareliable avionics: hardware component failures and design errors. Physical component failures can be handled by using redundancy and voting. This chapter addresses the problem of design errors. Design errors are errors introduced in the development phase rather than the operational phase. These may include errors in the specification of the system, discrepancies between the specification and the design, and errors made in implementing the design in hardware or software. The discussion in this chapter centers mainly around software design errors; however, the increasing use of complex, custom-designed hardware instead of off-the-shelf components that have stood the test of time makes this discussion equally relevant for hardware.

The problem with software is that the complexity exceeds our ability to have intellectual control over it. Our intuition and experience is with continuous systems, but software exhibits discontinuous behavior. We are forced to separately reason about or test millions of sequences of discrete state transitions. Testing of software sufficient to assure ultrahigh reliability has been shown to be infeasible for systems of realistic complexity [Butler and Finelli, 1993], and fault tolerance strategies cannot be relied upon because of the demonstrated lack of independence between design errors in multiple versions of software [Knight and Leveson, 1986]. Since the reliability of ultrareliable software cannot be quantified, life-critical avionics software must be developed in a manner that concentrates on producing a correct design and implementation rather than on quantifying reliability after a product is built. This chapter describes how rigorous analysis employing formal methods can be applied to the software development process. While not yet standard practice in industry,* formal methods is included as an alternate means of compliance in the DO178B standard for avionics software development.

21.2 Fundamentals of Formal Methods

Formal methods is the use of formal mathematical reasoning or logic in the design and analysis of computer hardware and software. Mathematical logic serves the computer system designer in the same way that calculus serves the designer of continuous systems—as a notation for describing systems and as an analytical tool for calculating and predicting the behavior of systems.

Formal logic provides rules for constructing arguments that are sound because of their form and independent of their meaning, and consequently can be manipulated with a computer program. Formal logic provides rules for manipulating formulas in such a manner that, given valid premises, only valid conclusions are deducible. The manipulations are called a proof. If the premises are true statements about the world, then the soundness theorems of logic guarantee that the conclusion is also a true statement about the world. Furthermore, assumptions about the world are made explicit and are separated from the rules of deduction.

Formal methods can be roughly divided into two basic components: specification and verification. Each of these is discussed below.

21.2.1 Formal Specification

Formal specification is the use of notations derived from formal logic to define (1) the requirements that the system is to achieve, (2) a design to accomplish those requirements, and (3) the assumptions about the world in which a system will operate. The requirements explicitly define the functionality required from the system as well as enumerating any specific behaviors that the system must meet, such as safety properties.

A design specification is a description of the system itself. In practice, a set of hierarchical specifications is often produced during the design process ranging from a high-level, abstract representation of the system to a detailed implementation specification, as shown in Figure 21.1. The highest-level specification might

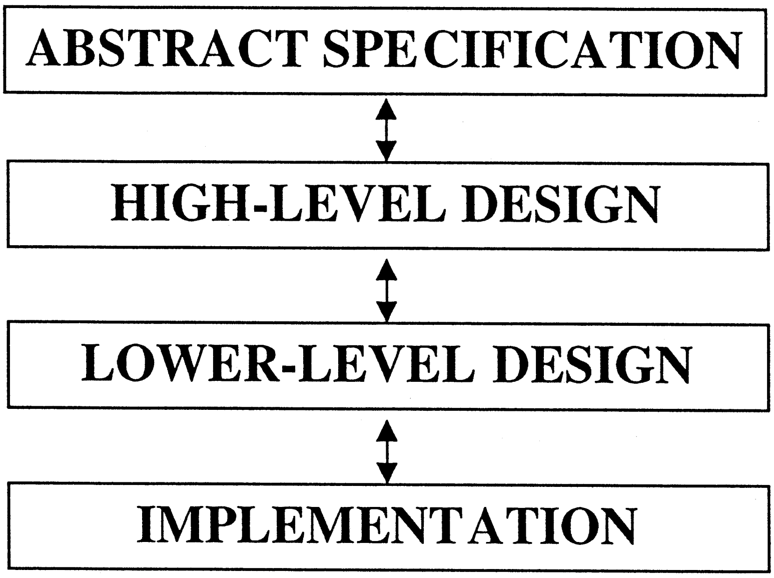

FIGURE 21.1 Hierarchy of formal specifications.

*However, Intel, Motorola, and AMD are using formal methods in the development of microchips.

describe only the basic requirements or functionality of the system in a state-transition format. The lowest-level specification would detail the algorithms used to implement the functionality. The hierarchical specification process guides the design and analysis of a system in an orderly manner, facilitating better understanding of the basic abstract functionality of the system as well as unambiguously clarifying the implementation decisions. Additionally, the resulting specifications serve as useful system documentation for later analysis and modification.

An avionics system can be thought of as a complex interface between the pilot and the aircraft. Thus, the specification of the avionics system relies on a set of assumptions about the behaviors of the pilot and the response of the aircraft. Likewise, the specification of a modular subsystem of an integrated avionics system will rely on a set of assumptions about the behaviors and interfaces of the other subsystems. The unambiguous specification of interfaces between subsystems also prevents the problem of developers of different subsystems interpreting the requirements differently and arriving at the system integration phase of software development with incompatible subsystems.

21.2.2 Formal Verification

Formal verification is the use of proof methods from formal logic to (1) analyze a specification for certain forms of consistency and completeness, (2) prove that the design will satisfy the requirements, given the assumptions, and (3) prove that a more detailed design implements a more abstract one. The formal verifications may simply be paper-and-pencil proofs of correctness; however, the use of a mechanical theorem prover to ensure that all of the proofs are valid lends significantly more assurance and credibility to the process. The use of a mechanical prover forces the development of an argument for an ultimate skeptic who must be shown every detail.

In principle, formal methods can accomplish the equivalent of exhaustive testing, if applied to the complete specification hierarchy from requirements to implementation. However, such a complete verification is rarely done in practice because it is difficult and time-consuming. A more pragmatic strategy is to concentrate the application of formal methods on the most critical portions of a system. Parts of the system design that are particularly complex or difficult to comprehend can also be good candidates for more rigorous analysis. Although a complete formal verification of a large, complex system is impractical at this time, a great increase in confidence in the system can be obtained by the use of formal methods at key locations in the system.

There are several publicly available theorem-proving toolkits. These tools automate some of the tedious steps associated with formal methods, such as *typechecking* of specifications and conducting the simplest of proofs automatically. However, these tools must be used by persons skilled in mathematical logic to perform rigorous proofs of complex systems.

21.2.3 Limitations

For many reasons, formal methods do not provide an absolute guarantee of perfection, even if checked by an automated theorem prover. First, formal methods cannot guarantee that the top-level specification is what was intended. Second, formal methods cannot guarantee that the mathematical model of the physical world, such as aircraft control characteristics, is accurate. The mathematical model is merely a simplified representation of the physical world. Third, the formal verification process often is only applied to part of the system, usually the critical components. Finally, there may be errors in the formal verification tools themselves.* Nevertheless, formal methods provide a significant capability for discovering/removing errors in large portions of the design space.

21.3 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

*We do not believe this to be a significant source of undetected errors in formal specifications or verifications.

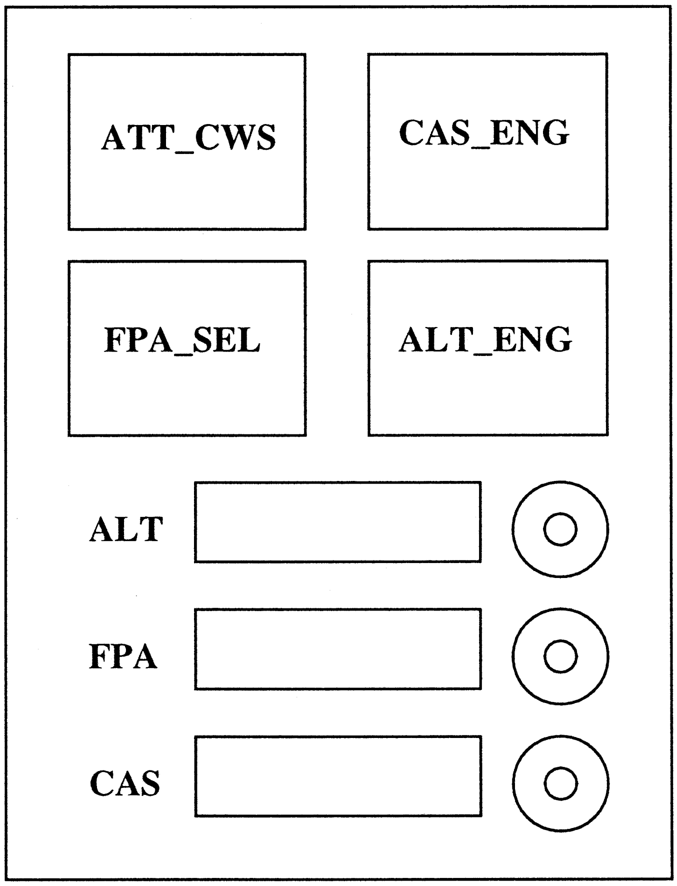

FIGURE 21.2 Mode-control panel.

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.

21.3.1 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 21.2. The system supports the following four modes:*

   attitude control wheel steering (`att_cws`),
   flight-path angle selected (`fpa_sel`),
   altitude engage (`alt_eng`), *and*
   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: air speed, flight-path angle, and altitude. The displays usually show the current values for the air speed, flight-path angle, and altitude 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 "preselected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to 25,000 feet, he will dial 250 into the altitude display window* and then press the* `alt_eng` *button to engage 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 1200 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 1200 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 preselected before the corresponding modes are engaged — the current values displayed will be used. The pilot can dial in a different value after the mode is engaged. However, the altitude must be preselected 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.

21.3.2 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):

$$S_c \longrightarrow S_n$$

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

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

*Altitude is expressed in terms of "flight level," which is measured in increments of 100 ft.

**In a real mode-control panel, the buttons would be lit with various colored lights to indicate which modes are currently selected and whether "preselected" or "current" values are being displayed. Because of space limitations, the display lights will not be included in the example specification.

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 [Owre et al., 1993].

21.3.2.1 Events

The pilot interacts with the mode-control panel by pressing the mode buttons and by dialing preselected 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_fpa_sel,
                press_alt_eng, input_alt, input_fpa, input_cas,
                alt_gets_near, alt_reached, fpa_reached}
```

21.3.2.2 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. Formalization of the system state description entails determining which attributes are necessary to fully describe the system's operation, then choosing a suitable formalism for representing those 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. Also, 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, fpa_sel is off and alt_eng is 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-types of mode_status:

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

*In the PVS language, a record definition begins with [# and ends with #]. The % denotes that the remainder of the line contains a comment and is ignored.

The type `off_eng` has two values: `off` and `engaged`. The state descriptor is thus corrected 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 preselected 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, and
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 preselected 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
```

21.3.2.3 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 subfunctions, 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_cas_eng: tran_cas_eng(st),
  press_fpa_sel: tran_fpa_sel(st),
  press_alt_eng: tran_alt_eng(st),
  input_alt    : tran_input_alt(st),
  input_fpa    : tran_input_fpa(st),
  input_cas    : tran_input_cas(st),
  alt_gets_near: tran_alt_gets_near(st),
  alt_reached  : tran_alt_reached(st),
  fpa_reached  : tran_fpa_reached(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 step is to define each of these subfunctions.

21.3.2.4 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].
```

The English specification also 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_set := 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 a preselected `alt_eng` display or preselected `fpa_sel` display when the `att_cws` button is pressed. However, the formal specification does explicitly indicate what will happen: these modes are turned off. The process of developing a complete mathematical specification has uncovered this ambiguity in the English specification. If the displays should not be changed when the corresponding mode is off, then 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 cases as well. In particular, whenever a mode becomes engaged, should any other preselected displays be changed to current or should they remain preselected? We decide to consult the system designers. They agree that the displays should be returned to current and suggest that the following be added to the English specification:

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

21.3.2.5 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, cas_disp := 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.

21.3.2.6 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 preselected*" (4). Thus, whether the mode is off or preselected, 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 not only is the `fpa_sel` mode engaged, but `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." 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). Thus, the altitude display must be specified to return to "current" status. If the `alt_eng` mode was 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." We interpret this to mean that if the fpa_sel button is pressed while the mode is 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 preselected altitude. However, Section (3) does not tell us 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:

```
    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,  alt_eng := off,
                  fpa_disp := current, 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."

21.3.2.7 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 preselected before the altitude engage button is pressed*" (4). This is interpreted to mean that the command is simply ignored if it has not been preselected. Consequently, the specification of tran_alt_eng begins:

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

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

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 1200 ft 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 preselected 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 preselected value is > 1200 feet away
`near_pre_selected`	the preselected value is <= 1200 feet away
`at_pre_selected`	the preselected value is = 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:

```
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 just as in the `tran_fpa_sel` subfunction.

Now the behavior of the system must be specified for the other case, 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. 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 st WITH [att_cws := off, fpa_sel := engaged,
                      alt_eng := armed] %% ARMED
        ENDIF
   ELSE st %% IGNORE request
   ENDIF
```

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

21.3.2.8 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 field 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 preselected 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 reengage 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.*

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 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]
  ELSE % 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
  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
```

21.3.2.9 Other Actions

There are other events that are not initiated by the pilot but that still affect the mode-control panel; in particular, 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 preselected altitude value: away, near_pre_selected or at_pre_selected. 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 preselected value
alt_reached	the altitude reaches the preselected value
alt_gets_away	the altitude is no longer near the preselected value

The transition subfunction associated with the first event must consider the case where the *alt_eng* mode is armed because the English specification states that, "*The flight-path angle select mode will remain engaged until the aircraft is within 1200 feet of the desired altitude, then the altitude engage mode is automatically engaged*" (3). Thus we have:

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

The subfunction associated with the second event is similar because we cannot 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]
   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).

However, the third event 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 preselected value. Thus, unless there is some major external event such as a windshear 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. The situation where the alt_eng mode is not engaged or armed would not be difficult to model, but also would not be particularly interesting to demonstrate.

There are three possible events corresponding to each of the other displays. These are all straightforward and have no impact on the behavior of the panel other than changing the status of the corresponding display. For example, the event of the airplane reaching the preselected flight-path angle is fpa_reached.

The specification of the corresponding subfunction tran_fpa_reached is

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

21.3.2.10 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 fpa_disp(st)  = current AND
     cas_disp(st) = current
```

Note that this predicate does not specify that the altitude field must have the value "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.

21.3.3 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 proven to meet the desired properties, the problem in the specification must be found or the property must be modified 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 proven 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_valid: THEOREM is_initial(st) IMPLIES valid_state(st)
nextstate_valid: 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 17.8 s on a 450 MHz PC (Linux) with 384 MB of memory.

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

```
   nextstate_valid :
[21] fpa_sel(st!1)= engaged
[22] press_fpa_sel?(event!1)
  u-------
[1] att_cws(st!1)= engaged
[2] alt_eng(st!1)= engaged
[3] press_att_cws?(event!1)
[4] press_alt_eng?(event!1)
```

*In order for this strategy to work, the invariant property (i.e., `valid_state`) must be sufficiently strong for the induction to go through. If it is too weak, the property may not be provable in this manner even though it is true. This problem can be overcome by either strengthening the invariant (i.e., adding some other terms) or by decomposing the problem using the concept of reachable states. Using the latter approach, one first establishes that a predicate "`reachable (st)`" delineates all of the reachable states. Then one proves that all reachable states are valid, i.e., `reachable (st)` $\Rightarrow$ `valid_state(st)`.

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]
```

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. There is nothing in this sequent to require that `att_cws(st!1) = engaged` or that `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,  alt_eng := off,
                 att_cws := engaged,
                 fpa_disp := current, 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.

Thus we see that formal verification can be used to establish global properties of a system and to detect errors in the specifications.

21.3.4 Alternative Methods of Specifying Requirements

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 elaborate this machine. Some of the more widely known are decision tables, decision trees, state-transition diagrams, state-transition matrices, Statecharts, Superstate, and R-nets [Davis, 1988].

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. Aerospace industries have typically used the table-oriented methods because they are considered the most readable when the specification requires large numbers of pages. Although there is insufficient space to discuss any particular method in this chapter, expression of a part of the mode-control panel using a table-oriented notation will be illustrated briefly.

Consider the following table describing the `att_cws` mode:

att_cws	Event	New Mode
off	press_att_cws	engaged
	press_fpa_sel WHEN fpa_sel /= off	engaged
	input_alt WHEN alt_eng /= off	engaged
	all others	off
engaged	press_att_cws	engaged
	press_alt_eng WHEN alt_eng = off AND alt_disp = pre_selected	off
	press_fpa_sel WHEN fps_sel = 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 lists all possible events, and the third column lists the new state of the att_cws mode after the occurrence of the event. Note also that some events may include other conditions. This table could be specified in PVS as follows:

```
att_cws_off: AXIOM att_cws(st) = off IMPLIES
att_cws(nextstate(event, st)) =
          IF (event = press_att_cws) OR
             (event = press_fpa_sel AND fpa_sel(st) /= off) OR
             (event = input_alt AND alt_eng(st) /= off) THEN engaged
          ELSE off
          ENDIF
att_cws_eng: AXIOM att_cws(st) = engaged IMPLIES
       att_cws(nextstate(event,st)) =
         IF (event = press_alt_eng AND alt_eng(st) = off
                       AND alt_disp(st) = pre_selected) OR
            (event = press_fpa_sel AND fps_sel(st) = off) THEN off
         ELSE engaged
         ENDIF
```

This approach requires that nextstate be defined in pieces (axiomatically) rather than definitionally. If this approach is used, it is necessary to analyze the formal specification to make sure that nextstate is completely defined. In particular, it must be shown that the function's behavior is defined for all possible values of its arguments and that there are no duplications. This must be performed manually if the tables are not formalized. The formalization can be done using a general-purpose theorem prover such as PVS or using a special-purpose analysis tool such as Tablewise* [Hoover and Chen, 1994].

When the specification can be elaborated in a finite-state machine, there are additional analysis methods available that are quite powerful and fully automatic. These are usually referred to as model-checking techniques. Some of the more widely known tools are SMV [Burch et al., 1992] and Murphi [Burch and Dill, 1994]. These require that the state-space of the machine be finite. Our example specification has a finite state space. However, if the values of chosen and measured altitude had not been abstracted away, the state-space would have been infinite.

21.4 Some Additional Observations

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 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.

*The Tablewise tool was previously named Tbell.

A more detailed version of this paper:

R. W. Butler, An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot, NASA TM-110255, May 1996, pp. 33.

is available at `http://techreports.larc.nasa.gov/ltrs/ltrs.html`. Recent work has looked at using formal methods to detect and eliminate mode confusion in flight guidance systems [Miller and Potts, 1999; Butler et al., 1998].

Defining Terms

Invariant: A property of a system that always remains true throughout all operational modes.

Mode: A particular operational condition of a system. The mode-control panel controls switching between operational conditions of the flight-control system.

Predicate: A function that returns true or false.

State: A particular operational condition of a system. A common method of representing the operational functioning of a system is by enumerating all of the possible system states and transitions between them. This is referred to as a state-transition diagram or finite-state machine representation.

Typechecking: Verification of consistency of data types in a specification. The detailed use of data types to differentiate between various kinds of objects, when supported by automated typechecking, can make a specification more readable, maintainable, and reliable.

References

Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., and Hwang, L.J., 1992. Symbolic Model Checking: 10^{20} States and Beyond, *Inf. Comput.* 98(2):142–170.

Burch, J.R. and Dill, David L., 1994. Automatic Verification of Pipelined Microprocessor Control, *Computer-Aided Verification, CAV'94,* Stanford, CA, pp. 68–80, June.

Butler, R.W. and Finelli, G.B. 1993. The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software, *IEEE Trans. Software Eng,* 19(1):3–12.

Butler, R. W., Miller, S. P., Potts, J. N., and Carreno, V. A, A Formal Methods Approach to the Analysis of Mode Confusion, *17th Digital Avionics Syst. Conf.,* Bellevue, WA, October 31–November 6, 1998.

Davis, A.M., 1988. A Comparison of Techniques for the Specification of External System Behavior., *CACM*, 31(9):1098–1115.

FAA, 1988. System Design and Analysis, Advisory Circular AC 25.1309-1A, U.S. Department of Transportation, Washington, D.C., June.

Hoover, D.N. and Chen, Z., 1994. Tbell: A Mathematical Tool for Analyzing Decision Tables, NASA Contractor Rep. 195027, November.

Knight, J.C. and Leveson, N.G., 1991. An Experimental Comparison of Software Fault-Tolerance and Fault Elimination, *IEEE Trans. Software Eng.,* SE-12(1):96–109.

Miller, S.P. and Potts, J. N., 1999. Detecting Mode Confusion Through Formal Modeling and Analysis, NASA/CR-1999-208971, January.

Owre, S., Shankar, N., and Rushby, J.M., 1993. The PVS Specification Language (Beta Release), Computer Science Laboratory, SRI International, Menlo Park, CA, 1993.

Further Information

A good introduction to the fundamentals of mathematical logic is presented in *Mathematical Logic* by Joseph R. Schoenfield.

The application of formal methods to digital avionics systems is discussed in detail in *Formal Methods and Digital Systems Validation for Airborne Systems,* NASA Contractor Report 4551, by John Rushby. Rushby's report was written for certifiers of avionics systems and gives useful insights into how formal methods can be effectively applied to the development of ultra-reliable avionics systems.

Two NASA Guidebooks on formal methods: “Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion” [NASA/TP-98-208193], 1998, and “Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner’s Companion” [NASA-GB-001-97], 1997 are available on the Web at `http://eis.jpl.nasa.gov/quality/Formal_Methods/`.

The complete specification and proofs for the mode-control panel example described in this chapter can be obtained at `http://shemesh.larc.nasa.gov/fm/ftp/larc/mode-control-ex/`. Several other formal methods application examples and papers can also be obtained from that site.