NASA/CR-1999-209332
ICASE Report No. 99-18

Analyzing Mode Confusion via Model Checking

Gerald Lüttgen
ICASE, Hampton, Virginia

Victor Carreño
NASA Langley Research Center, Hampton, Virginia

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

May 1999

ANALYZING MODE CONFUSION VIA MODEL CHECKING

GERALD LÜTTGEN[*] AND VICTOR CARREÑO[†]

Abstract. *Mode confusion* is one of the most serious problems in *aviation safety.* Today's complex *digital flight decks* make it difficult for pilots to maintain awareness of the actual states, or *modes*, of the flight deck automation. NASA Langley leads an initiative to explore how formal techniques can be used to discover possible sources of *mode confusion.* As part of this initiative, a *flight guidance system* was previously specified as a finite Mealy automaton, and the *theorem prover* PVS was used to reason about it.

The objective of the present paper is to investigate whether *state-exploration techniques*, especially *model checking*, are better able to achieve this task than theorem proving and also to compare several *verification tools* for the specific application. The flight guidance system is modeled and analyzed in Murϕ, SMV, and Spin. The tools are compared regarding their *system description language*, their practicality for analyzing mode confusion, and their capabilities for *error tracing* and for animating *diagnostic information.* It turns out that their strengths are complementary.

Key words. mode confusion, model checking, modeling, state exploration, verification tools

Subject classification. Computer Science

1. Introduction. Although *digital system automation* in the flight deck of aircrafts has contributed to *aviation safety*, we are starting to experience some undesirable side effects as a result of the high degree of automation. Automation has significantly reduced the overall pilot workload; however, in some instances the workload has just been re-distributed, causing short periods of very high workloads. This is usually the case during transition periods when the aircraft moves from one phase of flight to another or when data re-entry is necessary due to, e.g., route changes. It is during these transitional phases that pilots may get confused about the states, or *modes*, of the flight deck automation. *Mode confusion* may cause pilots to interact inappropriately with the on-board automation, with possibly catastrophic consequences. Indeed, incidents and accidents in aviation are increasingly attributed to this aspect of *pilot-automation interaction* [2].

NASA Langley Research Center, in partnership with avionics manufacturers and other organizations, is engaged in a program to explore ways to minimize the impact of mode confusion on aviation safety. One approach being studied is to identify the sources of mode confusion by *formally modeling* and *analyzing* avionics systems in order to determine if such sources exist in the systems. The *mode logic* of a *flight guidance system* was selected as a target system to develop this approach and to determine its feasibility. The flight guidance system offers a realistic avionics system and has been modeled and specified in many notations and languages including CoRE [9, 21], SCR [14, 20], Z [10, 31], ObjecTime [22, 28], and PVS [5, 25]. In the PVS effort, the behavior of the flight guidance system was encoded as a finite state machine. Properties, identified as possible sources of mode confusion by engineers, pilots, and experts in *human factors* [18], were

[*]Institute for Computer Applications in Science and Engineering (ICASE), Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681, e-mail: luettgen@icase.edu. This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the first author was in residence at ICASE.

[†]Assessment Technology Branch, NASA Langley Research Center, Hampton, VA 23681, e-mail: v.a.carreno@larc.nasa.gov.

defined in the PVS language. Some of these properties include *inconsistent behavior*, *ignored crew inputs*, and *indirect mode changes*. Proofs in the PVS model, which encodes a Mealy automaton, were undertaken to either show that a property holds or to discover conditions that preclude the property from becoming true. The employed style of theorem proving resembles a form of *state exploration*. Hence, the question arises whether state exploration techniques, such as *model checking* [6, 8, 26], are better suited for this task. The issues to be considered are whether model checking techniques are appropriate, whether the modeling and verification consumes less resources than theorem proving, and whether unsuccessful verification attempts return sufficient information which lead an engineer to potential design flaws. In order to get answers to our questions, we model and analyze the mode logic by applying three popular and publicly available *state-exploration/model-checking tools*, namely Murϕ [7, 23], SMV [19, 29], and Spin [15, 30].

The results of this paper show that all three model checking tools have the capability of modeling the mode logic of the flight guidance system and analyzing properties related to mode confusion. However, each verification tool has its own strengths and weaknesses. Therefore, we put our emphasis on comparing the suitability of Murϕ, SMV, and Spin in the context of our application. We draw our comparison along three aspects: (1) the suitability of the tools' languages for *modeling* the mode logic, (2) the suitability of the tools for *specifying* and *verifying* the *mode confusion properties* of interest, and (3) the tools' ability to generate and *animate diagnostic information*. The first aspect is of importance because it influences the way in which we model the example system. The second aspect refers to the expressiveness of the language in which system properties are encoded, and also to the degree of orthogonality between the specification of the system and the specification of its properties. The third aspect is perhaps the most important one for engineers since system designs are often incorrect in early design stages. Finally, it should be noted that our comparative case study is not intended to determine which verification tool is 'the best.' All comparisons made only refer to a certain class of applications; the main characteristics of the flight guidance system are its *synchronous*, *reactive*, and *deterministic behavior*.

The remainder of this paper is organized as follows. Section 2 gives an overview of the flight guidance system, of its mode logic, and of potential sources for mode confusion. Sections 3, 4, and 5 show the modeling and analysis of the mode logic in Murϕ, SMV, and Spin, respectively. Section 6 discusses the strengths of each verification tool for our application and refers to related work, while Section 7 contains our conclusions and directions for future work. Finally, the appendices include the full models of the mode logic.

2. Flight Guidance Systems and Mode Logics.

The *flight guidance system* is a central component of the *flight control system* (see Fig. 2.1). It continuously determines the difference between the actual state of an aircraft – its position, speed, and attitude as measured by its *sensors* – and its desired state as given by the crew and/or the *flight management system*. In response, the flight guidance system generates commands to minimize this difference, which the *autopilot* may translate into movements of the aircraft's *actuators*. These commands are calculated by *control law algorithms* that are selected by the *mode logic*.

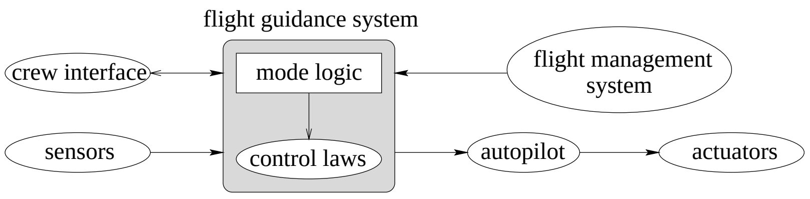

Fig. 2.1. *Flight control system.*

In the following we focus on the mode-logic part of flight guidance systems. Especially, we leave out the modeling of the control laws and, if no confusion arises, use interchangeably the terms flight guidance system and mode logic. For the purposes of this paper, it suffices to understand the functionality of the mode logic and how it is decomposed into sub-components. The flight guidance system essentially acts as a deterministic machine which is composed of several synchronous sub-machines. It receives *events* from its *environment* – i.e., the *crew interface*, the *aircraft's sensors*, and the *flight management system* – in a *nondeterministic* fashion and reacts to them by changing its state appropriately. The functionality of a flight guidance system varies with application, vendor, customer preferences, and other factors.

Fig. 2.2 shows a typical mode logic for a business jet/commuter jet flight guidance system. The mode logic can be represented and modeled by three interacting components: the *lateral guidance*, the *vertical guidance*, and the *flight director*. The mode of the flight director – which can be either *cues*, *no-cues*, or *off* – determines whether or not the flight guidance system is being used as a navigational aid either manually by the crew or automatically through the autopilot. The lateral guidance subsumes the *roll* mode (Roll), the *heading* mode (HDG), the *navigation* mode (NAV), and the *lateral go-around* mode (LGA), whereas the vertical guidance subsumes the *pitch* mode (Pitch), the *vertical speed* mode (VS), and the *vertical go-around* mode (VGA). Each mode can be either *cleared* or *active*, with the navigation mode having additional sub-states in the *active* state. The behavior of each component places constraints on the other components. For example, when the flight director is on, there must be exactly one mode active in the lateral guidance and one mode active in the vertical guidance. In some situations, an external event may require several simultaneous mode changes. Indeed, the behavior of the flight guidance system reflects a kind of two-level semantics similar to *Statecharts* [13], where both semantic levels are not independent but connected via the *synchrony hypothesis* [1]. This hypothesis guarantees that a system completes its reaction to an external event before the next external event arrives.

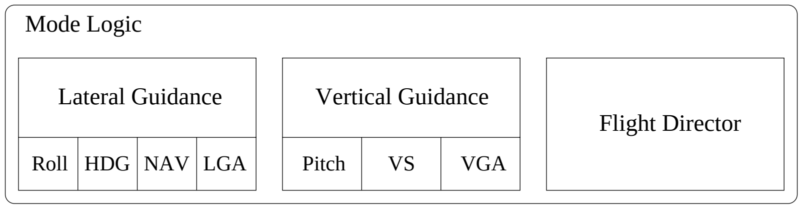

FIG. 2.2. *Architecture of the model logic of the flight guidance system.*

Before we discuss the modeling of the flight guidance system within the verification tools Murϕ, SMV, and Spin, we briefly mention some properties of our system, which can be classified as *mandatory properties* and *mode confusion properties*. Some of the former properties are: (i) if the flight director is off, all lateral and vertical guidance modes must be cleared, (ii) if the flight director is on, then exactly one lateral and one vertical mode is active, and (iii) the lateral and vertical default modes are activated when the flight director is on and when all other modes are cleared. These and other mandatory properties must be true if we have accurately modeled the system. Regarding mode confusion, several categories are identified in [18]. We have selected three categories to use in the analysis of our system: (1) *inconsistent behaviors*, i.e., a crew interface input (switch, dial, etc.) has different functionality for different system states, (2) *ignored operator inputs*, i.e., a crew input does not result in a change of state, and (3) *indirect mode changes*, i.e., the system changes its state although no crew input is present. To discover if there are possible sources of mode confusion, we formulate the negation of each property – there are no inconsistent behaviors, no ignored inputs, and no indirect mode changes – and try to prove it. Conditions that prevent us from successfully completing the proof, manifested by *unprovable subgoals* in a theorem prover and *error traces* in model-checking tools, are the ones we intend to uncover. As expected, this process is labor intensive when using theorem proving [22]. The work described here investigates if model checking is a more efficient way of performing the analysis.

3. Modeling the Mode Logic in Murϕ. The *Murϕ Verification System* [7, 23], a state-exploration tool developed by David Dill's group at Stanford University, consists of a *compiler* and a *description language*. The compiler takes a Murϕ description and generates a C++ *special-purpose verifier* for it. This verifier can then be used for checking *assertions* and *deadlock behavior* of the system under consideration.

The Murϕ description language is a *high-level language* which borrows from many constructs found in programming languages, such as Pascal. It may be used to model synchronous as well as asynchronous hardware and software systems which can be compiled into finite *Kripke structures*, i.e., finite automata whose states are attached with the semantic information of interest. Murϕ descriptions may include declarations of *constants*, *finite data-types* (such as Booleans, enumeration types, finite subranges of integers, record types, and array types), *global* and *local variables*, and unnested *procedures* and *functions*. Moreover, they contain *transition rules* for describing system behavior, a description of the *initial states*, and a set of *state invariants* and *assertions*. Each transition rule may consist of a *guard* – which is never needed in our application scenario – and an *action*, i.e., a statement which modifies the values of global variables. A *state* in Murϕ's execution model is an assignment to all global variables in the description under investigation. A *transition* is then determined by a rule, taken nondeterministically from the set of transition rules whose condition is true in the current state. The rule's execution updates all or some global variables according to its action.

Table 3.1
Specification of module simple guidance *in Murϕ*

```
TYPE sg_modes   : ENUM { cleared, active };
TYPE sg_events  : ENUM { activate, deactivate, switch, clear };
TYPE sg_signals : ENUM { null, activated, deactivated };

PROCEDURE simple_guidance(VAR mode:sg_modes; event:sg_events; VAR signal:sg_signals);
BEGIN
  IF mode=cleared THEN SWITCH event CASE activate   : signal := activated;    mode := active;
                                    CASE deactivate : signal := null;
                                    CASE switch     : signal := activated;    mode := active;
                                    CASE clear      : signal := null;
                       END;
  ELSE                 SWITCH event CASE activate   : signal := null;
                                    CASE deactivate : signal := null;         mode := cleared;
                                    CASE switch     : signal := deactivated;  mode := cleared;
                                    CASE clear      : signal := deactivated;  mode := cleared;
                       END;
END; END;
```

In the center of the Murϕ model of the flight guidance system is the deterministic procedure `fgs`. This procedure encodes the system's reaction to some environment event `env_ev` entering the mode logic. For the purposes of this paper it is not important to name the fourteen different environment events interacting with the mode logic. However, by declaring a transition rule for each environment event `env_ev` as `RULE "rule_for_env_event" BEGIN fgs(env_ev); END`, we model the nondeterministic behavior of the environment which arbitrarily chooses the event entering the system at each synchronous step. Please observe that our encoding of the environment does not require us to store event names explicitly in a global variable, but rather to inject them to `fgs` via a call-by-value parameter. Due to space constraints we do not completely specify procedure `fgs` here. Instead, we concentrate on modeling the vertical-guidance component of the flight guidance system. Let us define the modes of the vertical-guidance component as instantiations of an abstract data-type module `simple_guidance`, specified in Murϕ as procedure, which encodes each mode's behavior as a Mealy automaton behaving like a Boolean switch (cf. Table 3.1). The module is parameterized by the mode `mode` under consideration (as call-by-reference parameter), the input event `event` (as call-by-value parameter), and the output event `signal` (as call-by-reference or return parameter). The parameters are of enumeration types `sg_mode`, `sg_events`, and `sg_signals`, respectively, where type `sg_mode` ranges over the values `cleared` and `active`, type `sg_event` ranges over `activate`, `deactivate`, `switch`, and `clear`, and type `sg_signal` ranges over `null`, `activated`, and `deactivated`. The body of `simple_guidance` specifies the reaction of a mode to input event `event`, with respect to its current state `mode`. This reaction is described by an *if-statement*, two *case-selections*, and *assignments* to variable `mode` and return parameter `signal`.

Table 3.2
Specification of module vertical guidance *in Murϕ*

```
VAR pitch, vs, vga : sg_modes;

PROCEDURE vertical_guidance(env_ev:env_events);
VAR sig : sg_signals;
BEGIN CLEAR sig;
  IF  pitch_event(env_ev) THEN simple_guidance(pitch, pitch_conv(env_ev), sig);
                               IF    sig=activated   THEN simple_guidance(vs,    deactivate, sig);
                                                          simple_guidance(vga,   deactivate, sig);
                               END;
  ELSIF  vs_event(env_ev) THEN simple_guidance(vs, vs_conv(env_ev), sig);
                               IF    sig=activated   THEN simple_guidance(pitch, deactivate, sig);
                                                          simple_guidance(vga,   deactivate, sig);
                               ELSIF sig=deactivated THEN simple_guidance(pitch, activate,   sig);
                               END;
  ELSIF vga_event(env_ev) THEN simple_guidance(vga, vga_conv(env_ev), sig);
                               IF    sig=activated   THEN simple_guidance(pitch, deactivate, sig);
                                                          simple_guidance(vs,    deactivate, sig);
                               ELSIF sig=deactivated THEN simple_guidance(pitch, activate,   sig);
                               END;
END; END;
```

We can now specify the vertical-guidance module as a procedure, called `vertical_guidance` (cf. Table 3.2), by employing procedure `simple_guidance` for describing the behavior of the modes `pitch`, `vs`, and `vga`, which are defined as global variables. The task of procedure `vertical_guidance` is firstly to recognize whether the environment event `env_ev` passed to the system refers to mode Pitch, to mode VS, or to mode VGA. This is achieved with help of the three auxiliary functions `pitch_event`, `vs_event`, and `vga_event`, respectively. Then `env_ev` is translated to an event of type `sg_events` via the auxiliary functions `pitch_conv`, `vs_conv`, and `vga_conv`, respectively, and passed to the mode to which it belongs. If this mode is activated by the event, i.e., `simple_guidance` returns value `activated` via local variable `sig`, then the other two modes must instantly be deactivated by invoking `simple_guidance` with respect to the appropriate modes and event `deactivate`. It should be mentioned that the above modeling of components `simple_guidance` and `vertical_guidance` is carried over one-to-one from the PVS model of the flight guidance system, which was developed by NASA Langley and Rockwell Collins [5, 22]. In fact, every PVS construct used in [22] corresponds to a construct in Murϕ's description language. However, we sometimes find it useful to translate functions in PVS to procedures in Murϕ that have an additional call-by-reference parameter for returning the computed value. In PVS, only functions can be specified, although procedures would sometimes be more preferable from a software-engineering point of view.

TABLE 3.3
Specification of some mode confusion properties *in Murϕ*

```
VAR old_pitch, old_vs, old_vga : sg_modes;

PROCEDURE mode_confusion_properties(env_ev:env_events);
BEGIN
  ALIAS mode_change : pitch != old_pitch | vs != old_vs | vga != old_vga; DO
    IF env_ev=vs_switch_hit THEN
      -- check for response to pressing VS button
      assert (old_vs=cleared -> vs=active ) "vs_toggle_1";
      assert (old_vs=active  -> vs=cleared) "vs_toggle_2";
    END;
    -- search for ignored crew inputs (property violated)
    assert (crew_input(env_ev) -> mode_change) "search_for_ignored_crew_inputs";
    -- no unknown ignored crew inputs
    assert ((crew_input(env_ev) & !ignored_crew_input(ev)) -> mode_change) "no_unknown_ignored";
    -- search for indirect mode changes (property violated)
    assert (!crew_input(env_ev) -> !mode_change) "search_for_indirect_mode_changes";
    -- no unknown indirect mode changes
    assert ((!crew_input(env_ev) & !indirect_mode_change(env_ev)) -> !mode_change) "no_unknown...";
  END;
  -- update state variables
  old_pitch := pitch;  old_vs := vs;  old_vga := vga;
END;
```

We now turn our focus to specifying mode confusion properties. As states are generated by the Murϕ verifier, `assert` statements, that were explicitly included in the action of a rule, are checked. If some assertion is violated – i.e., the `assert` statement is evaluated to false in some reachable system state – the Murϕ verifier halts and outputs the string which the user associated with the `assert` statement under consideration. Moreover, the verifier outputs diagnostic information which consists of a sequence of states leading from the initial state to the error state. The verifier also halts if the current state possesses no successor states, i.e., if it is deadlocked. Let us return to the three categories of mode confusion mentioned in Section 2 by showing how an exemplary property of each category can be stated as an assertion.

In the system description of our mode logic, we encapsulate all assertions in the single procedure `mode_confusion_properties`, which is invoked as the last statement in procedure `fgs` and which takes the current environment event `env_ev` as parameter (cf. Table 3.3; the notation "`--`" introduces a comment line in Murϕ, `!=` denotes *inequality*, and `|`, `&`, `!`, and `->` stand for *logical disjunction*, *conjunction*, *negation*, and *implication*, respectively). Since all mode confusion properties of interest concern the transition from one system state to the next, we need to store the global variables' values of the previously visited state. For this purpose, we introduce new global variables `old_pitch`, `old_vs`, and `old_vga`. The need for this overhead arises because Murϕ's verification capabilities are restricted to reason about simple *state invariants* only and not about more general "*state transition invariants.*" Therefore, such state transition invariants need to be encoded as state invariants, which doubles the size of the state vector for our system description. The first two assertions in Table 3.3, belonging to the first category of mode confusion properties, state that environment event `vs_switch_hit` acts like a toggle with respect to mode VS, i.e., (i) if mode VS was in state `cleared` and event `vs_switch_hit` arrived, then it is now in state `active`, and (ii) analogously with exchanged roles of `cleared` and `active`.

As example of the second category of mode confusion properties, we check whether no crew inputs are ignored, i.e., whenever an event that originated from the crew enters the mode logic, then at least one global variable changes its value. We can specify this property as implication `crew_input(env_ev) -> mode_change`, where `crew_input` is a Boolean function determining whether environment event `env_ev` originates from the crew and where `mode_change` is a shortcut, introduced as an `ALIAS` statement in Murϕ. As expected (cf. Section 2), this mode confusion property does not always hold. Using the error trace returned by Murϕ helps us in identifying the causes, as is our objective. We do not go into the details here but mention that we filter out the identified cause by including an additional predicate `ignored_crew_input`, stating the negation of the cause, in the premise of the assertion (cf. Table 3.3). We then re-run the Murϕ verifier and iterate the described process until the assertion becomes true, thereby gradually capturing all crew-input scenarios responsible for mode confusion. When comparing our approach to the one taken in PVS [22] – i.e., trying theorem proving until either obtaining proof goal `true` or until reaching an unsatisfiable proof goal – we feel that ours is more effective. We discovered that the variant of `ignored_crew_input` used in the PVS model is stronger than necessary, thereby wrongfully identifying some situations as sources of mode confusion. The difficulty with the analysis in PVS is the following. Murϕ returns error traces that pinpoint the condition violating the assertion, whereas in a PVS failed proof the condition must be extracted from a proof sequent consisting of assertions and subgoals. Extracting conditions from a proof sequent is often more time-consuming and usually requires a better understanding of the system's behavior.

Similar to the assertion "*checking for ignored crew inputs*" we approach the third category of mode confusion properties. The property we consider is "*no indirect mode changes,*" which prohibits a system's state to change if the current environment event is not originated by the crew. Using Murϕ, we discover the conditions that invalidate this property. As before, we weaken the property by introducing a predicate `indirect_mode_change`, until all sources of indirect mode changes are detected. The mandatory properties mentioned in Section 2 are formalized as Murϕ `invariant` statements and proved. The difference between an `assert` statement and an `invariant` statement in Murϕ is that the former appears in the system description part of the model, while the latter is orthogonal to the system description. The reason for specifying mode confusion properties in the system description is their reference to the auxiliary variables `old_pitch`, `old_vs`, and `old_vga`. In order to keep the state space small, the auxiliary variables must be re-assigned to the actual values of `pitch`, `vs`, and `vga`, respectively, *before* a step of the synchronous system is completed.

Summarizing, Murϕ's description language turned out to be very useful for our task, especially since the already developed PVS model of the flight guidance system [22] could be simply carried over. Unfortunately, Murϕ's capability for expressing system properties is quite restrictive, forcing us to encode state transition invariants as state invariants, thereby doubling the number of global variables and, as a consequence, Murϕ's memory requirements. The full Murϕ model subsumes about 30 assertions and leads to a Kripke structure having 242 states. In each state of the Kripke structure any of the 14 environment events may potentially enter the system; this gives $3\,388 = 242 \times 14$ transitions in total. The state-space exploration undertaken by the Murϕ verifier took under 2 seconds on a SUN SPARCstation 20. This is an impressive result when compared to the semi-automatic proofs in PVS [22].

4. Modeling the Mode Logic in SMV. The SMV system [19, 29], originally developed by Ken McMillan at Carnegie-Mellon University, is a *model-checking tool* for verifying *finite-state systems*, described in a simple description language, against specifications in the *temporal logic* CTL [6, 8]. The SMV verifier implements a *symbolic* model-checking algorithm [4] based on *Binary Decision Diagrams* (BDDs) [3].

SMV's description language is a very simple, yet elegant language for specifying finite Kripke structures, which has the feel of a *hardware description language.* The language's data types are *Booleans* (where *false* and *true* are encoded as `0` and `1`, respectively), *enumeration types*, and *arrays.* Its syntax resembles a style of *parallel assignments*, and its semantics is similar to single assignment *data flow languages.* For structuring specifications, SMV allows *modular* hierarchical descriptions. In contrast to Murϕ, SMV descriptions are not compiled into a special-purpose verifier, but are *interpreted* instead. The interpreter makes sure that the specified system is indeed implementable by checking for *multiple assignments* to the same variable, *circular assignments*, and *type errors.* The SMV language also includes constructs for stating system specifications in the temporal logic (fair)CTL [8], which allows one to express a rich class of temporal properties, including *safety*, *liveness*, and *fairness properties.* In the present application of the synchronous flight guidance system, we focus on safety properties, to which invariants belong.

Table 4.1
Specification of module simple guidance *in SMV*

```
MODULE simple_guidance(activate, deactivate, switch, clear)
VAR    mode        :  {cleared, active};
ASSIGN init(mode)  := cleared;
       next(mode)  := case deactivated | deactivate : cleared;
                           activated                : active;
                           1                        : mode;
                      esac;
DEFINE activated   := (mode=cleared) & (activate | switch);
       deactivated := (mode=active ) & (clear    | switch);
```

A *module description* in SMV consists of four parts: (1) the `MODULE` clause, stating the module's name and a list of formal (call-by-reference) parameters, (2) the `VAR` clause, declaring (global) variables needed for describing the module's behavior, (3) the `ASSIGN` clause, which specifies the initial value of all variables (cf. `init`) and how each variable is updated from state to state (cf. `next`), and (4) the `DEFINE` clause, which allows one to introduce abbreviations for more complex terms. Similar to the Murϕ model, the main module `MAIN` of our SMV specification encodes the environment of the flight guidance system, which nondeterministically sends events to the mode logic. This is done by defining variable `env_ev` of enumeration type `env_events`, which contains all environment events, and by adding "`init(env_ev) := env_events; next(env_ev) := env_events`" to the `ASSIGN` clause. Analogous to the Murϕ model, we specify a module `simple_guidance` (see Table 4.1) and, thereby, show how Mealy machines may be encoded in SMV. Module `simple_guidance` takes the four input events `activate`, `deactivate`, `switch`, and `clear` – which can be either absent or present – as parameters. The state associated with `simple_guidance` is variable `mode` which may adopt values `cleared` and `active`. Note that the values of enumeration types are encoded by the SMV interpreter using a collection of Boolean variables, such that transition relations can be represented by BDDs. The initial value `init(mode)` of `mode` is `cleared`. The behavioral part of `simple_guidance` is described in the `next(mode)` statement, which consists of a `case` expression. The value of this expression is determined by the first expression on the right hand side of the colon such that the condition on the left hand side is true. The symbols, `=`, `&`, and `|` stand for *equality*, *logical conjunction*, and *logical disjunction*, respectively. The terms `activated` and `deactivated` are defined as abbreviations of more complex terms in the `DEFINE` clause. The values of `mode`, `activated` and `deactivated` are accessible from outside the module. Therefore, a `DEFINE` clause may be used for encoding output events of Mealy automata.

Table 4.2
Specification of module vertical guidance *in SMV*

```
MODULE vertical_guidance(vs_pitch_wheel_changed, vs_switch_hit, ga_switch_hit,
                         sync_switch_pressed,    ap_engaged_event)
VAR pitch : simple_guidance(pitch_activate, pitch_deactivate,            0,         0);
    vs    : simple_guidance(             0,    vs_deactivate, vs_switch_hit,        0);
    vga   : simple_guidance(             0,   vga_deactivate, ga_switch_hit, vga_clear);
DEFINE pitch_activate   := (vs_event    & vs.deactivated)  | (vga_event & ga.deactivated) |
                           vs_pitch_wheel_changed;
       pitch_deactivate := (vs_event    & vs.activated)    | (vga_event & ga.activated);
       vs_deactivate    := (pitch_event & pitch.activated) | (vga_event & ga.activated);
       vga_deactivate   := (pitch_event & pitch.activated) | (vs_event  & vs.activated);
       vga_clear        := ap_engaged_event | sync_switch_pressed;
       pitch_event      := vs_pitch_wheel_changed;
       vs_event         := vs_switch_hit;
       vga_event        := ap_engaged_event | sync_switch_pressed |
                           ga_switch_hit;
```

Before we model the vertical guidance component, we comment on why we have encoded the input event of the `simple_guidance` Mealy machine using four different signal lines – i.e., adopting a hardware-description language point of view – instead of a single event of some enumeration type subsuming all four values. If `activate`, `deactivate`, `switch`, and `clear` were combined in an enumeration type, a syntactic – though not semantic – circularity would be introduced which could not be resolved by SMV, i.e., our description of the mode logic would be rejected. Another difference between `simple_guidance` as a *module* in SMV and as an *abstract data-type* in Murϕ is that the mode variable is encapsulated within the SMV module, whereas it is a call-by-reference parameter in Murϕ's abstract data type. We feel that SMV reflects the architecture of the flight guidance system better, since `mode` belongs to component `simple_guidance` and should not be declared outside.

The behavior of each mode of the vertical guidance model (cf. Table 4.2), Pitch, VS, and VGA, can now be described by instantiating the module `simple_guidance`, as is done in the `VAR` clause of module `vertical_guidance`. Thereby, global variables `pitch.mode`, `vs.mode`, and `vga.mode` are created as part of the state vector of our SMV model. All actual parameters of each `simple_guidance` module can be specified as Boolean terms on the input parameters of module `vertical_guidance`, which are essentially environment events triggering an action regarding the vertical aircraft axis. Note that the Boolean functions `pitch_event`, `vs_event`, and `vga_event` used in the Murϕ description are encoded here in the `DEFINE` clause of `vertical_guidance`. Our modeling of `vertical_guidance` is self-explanatory and visualizes the differences between the SMV and the Murϕ languages. While in Murϕ each synchronous step of the flight guidance system can be modeled by a *sequential algorithm*, it must be described in SMV by *parallel assignments*.

Table 4.3
Specification of some mode confusion properties *in SMV*

```
DEFINE mode_change := !(vertical.pitch.mode = cleared <-> AX vertical.pitch.mode = cleared) |
                      !(vertical.pitch.mode = active  <-> AX vertical.pitch.mode = active ) | ...

-- check for response to pressing VS button
SPEC AG (vertical.vs.mode=cleared & env_ev=vs_switch_hit -> AX vertical.vs.mode=active)
SPEC AG (vertical.vs.mode=active  & env_ev=vs_switch_hit -> AX vertical.vs.mode=cleared)
-- search for ignored crew inputs (property violated)
SPEC AG (crew_input -> mode_change)
-- no unknown ignored crew inputs
SPEC AG (crew_input & !ignored_crew_input -> mode_change)
-- search for indirect mode changes (property violated)
SPEC AG (!crew_input -> !mode_change)
-- no unknown indirect mode changes
SPEC AG ((!crew_input & !indirect_mode_change) -> !mode_change)
```

In SMV, properties are specified in the temporal logic *Computational Tree Logic* (CTL) [6, 8]. *Fairness constraints* may also be imposed on SMV models but are not needed for our purposes since we are strictly interested in invariants related to aspects of mode confusion. In SMV, temporal properties are introduced within the same file as the system description by the keyword `SPEC`. We do not need to introduce CTL formally here, as we use only a very limited sublanguage of it. All of our properties are of the form `AG`ϕ, where `AG` stands for "`always generally`," i.e., every state on every path through the system satisfies property ϕ. The formula `AX`ϕ expresses that all successor states of the current state satisfy formula ϕ. In this light, the first formula in Table 4.3, related to checking the response to pressing the VS button, states: "every reachable state in the underlying Kripke structure of the model satisfies that, if mode VS in `vertical_guidance` is currently `cleared` and event `vs_switch_hit` enters the system, then mode VS in `vertical_guidance` is `active` in every successor state of the current state." Note that the symbols `->` and `<->` used in Table 4.3 stand for *logical implication* and *equivalence*, respectively. The identifiers `mode_change`, `crew_input`, `indirect_mode_change`, and `ignored_crew_input` are abbreviations of Boolean expressions defined in a `DEFINE` clause, as exemplarily shown for `mode_confusion`. The presence of operator `AX` in CTL remedies the need to keep track of old values of mode variables. Thereby, the size of the associated state vector of the SMV model is cut in half when compared to the Murϕ model. Moreover, a fully orthogonal treatment of model and property specifications is achieved. The SMV system verified about thirty assertions in slightly more than half a second using 438 BDD nodes and allocated less than 1 MByte memory on a SUN SPARCstation 20. The two properties "*search for ignored crew inputs*" and "*search for indirect mode changes*" were invalidated as in the Murϕ model. The returned error traces – reporting the assignments of each variable and each identifier declared in a `DEFINE` clause in every state of the traces – are of help in identifying potential problems with the model. SMV also includes an *interactive mode* which provides a very simple assistant for interactive debugging. The state space of the SMV model consists of 3 388 states, which corresponds to the 242 states of the Murϕ model since the actual environment event – out of 14 possible events – must be stored in a variable in SMV ("$242 \times 14 = 3\,388$").

Summarizing, SMV performed very well on our example and showed the suitability of *symbolic* model checking to the flight guidance system. In fact, the mode logic's behavior can be described by Boolean terms and, thus, represented efficiently using BDDs. CTL turned out to be an excellent language for specifying mode confusion properties due to the presence of next-state operator `AX`. SMV's modeling language has the feel of a hardware description language and is not as high level as Murϕ's language. However, SMV's module concept allowed us to model the *architecture*, but not the functionality, of the flight guidance system one-to-one to the original PVS specification [22].

5. Modeling the Mode Logic in Spin. Last, but not least, we explore the utility of the verification tool Spin [15, 16, 30], which was developed by Gerard Holzmann at Bell Labs, for our case study. Spin is designed for analyzing the logical consistency of *concurrent systems*. It is especially targeted towards modeling and reasoning about distributed systems, such as communication protocols, where several concurrent processes exchange messages by communicating synchronously via handshaking or asynchronously via buffered channels. The description language of Spin, called Promela, allows one to specify *nondeterministic processes*, *message channels*, and *variables* in a C-like syntax. Given a system description in Promela, whose semantics is again defined as a Kripke structure, Spin can – in contrast to Murϕ and SMV – perform random or interactive *simulations* of the system's execution. Similar to Murϕ, it can generate a special-purpose verifier, i.e., a C-program, which performs an exhaustive exploration of the system's state space. Such a state exploration may – among other things – check for *deadlocks* and *unreachable code*, validate *invariants*, and verify properties specified in a *linear-time logic* [8, 11]. Linear-time logic is not as expressive as the branching-time logic *fair*CTL employed in SMV. However, it is rich enough to specify all properties of interest in this paper. Spin's verifier was implemented having memory efficiency in mind, e.g., it includes optional *partial-order techniques* [12] and *bitstate hashing* [17].

TABLE 5.1
Specification of the main process init *in Spin*

```
init{ env_ev=null;
      do
      :: atomic{if                           /* loop body encodes 1 synchronous step   */
                :: env_ev=vs_switch_hit      /* nondeterministic choice of env. event  */
                :: ...                       /* 13 more cases, one for each env. event */
                fi;
                fgs(env_ev);                 /* perform synchronous step               */
                env_ev=null }                /* env. event is no longer needed         */
      od }
```

Since our flight guidance system is a synchronous system, it falls out of the intended scope of Spin. Nevertheless, we show that Spin allows us to successfully carry out our case study. The Promela fragment depicted in Table 5.1 encodes our synchronous model using a single process, namely Spin's main process `init`. Here, the global variable `env_ev` is of type `mtype`, which contains an enumeration of all event and signal names that may occur in the mode logic. Promela's type system supports basic data types (such as `bit`, `bool`, and `byte`), as well as *arrays*, *structures* (i.e., records), and *channels*. Unfortunately, it only allows a single declaration of an *enumeration type*, which must be named `mtype`. The statement `atomic` in `init` attempts to execute all statements in its body in one *indivisible* step. Especially, it does not store intermediate states which might arise during the execution of the body. Thus, we may use this construct for encoding our complex algorithm – see procedure `fgs` of the Murϕ model in Section 3 – performing a single synchronous step. The *repetition statement* `do` together with the nested *nondeterministic-choice statement* `if` nondeterministically chooses which environment event to assign to variable `env_ev`. Since `env_ev` is no longer needed outside of `fgs` it is reset to dummy value `null` and, thus, does not contribute to the observable state space. The reason that we have not simply spelled out `fgs(vs_switch_hit)`, and so on for each environment variable, is that – as we argue below – `fgs` needs to be implemented as an *inline*. Expanding this long inline fourteen times turns out to be inefficient.

Table 5.2
Specification of module simple guidance *in Spin*

```
inline simple_guidance(mode, event, signal)
{ if :: mode==cleared -> if :: event==activate   -> signal=activated;   mode=active
                            :: event==deactivate -> signal=null
                            :: event==switch     -> signal=activated;   mode=active
                            :: event==clear      -> signal=null
                         fi
     :: mode==active  -> if :: event==activate   -> signal=null
                            :: event==deactivate -> signal=null;        mode=cleared
                            :: event==switch     -> signal=deactivated; mode=cleared
                            :: event==clear      -> signal=deactivated; mode=cleared
                         fi
  fi }
```

Promela does not possess any kind of procedure construct other than the process declaration `proctype`. However, we may not introduce additional processes to the main process `init`, since then our model would not reflect a synchronous system any more. The only construct of Promela, which we can use for resembling the architecture of the flight guidance system, is the `inline` construct which may take (call-by-reference) parameters, such as the parameters `mode`, `event`, and `signal` for component `simple_guidance` (cf. Table 5.2). When compiling a Promela description, each occurrence of `simple_guidance` in `vertical_guidance` is replaced with its body. The modes instantiating the parameter `mode` are global variables of type `bit`, where `cleared` and `active` are defined to represent the constants `0` and `1`, respectively, using the preprocessor command `#define`. The events and signals `clear`, `activate`, `deactivate`, `switch`, `null`, `activated`, and `deactivated` are of type `mtype`. The body of `simple_guidance` contains the Promela statement `if`. Its behavior is defined by a nondeterministic selection of one of its executable options, which are separated by double colons, and by executing it. In our case, each option consists of a guarded expression, which is executable if the expression on the left of `->` evaluates to true in the current system state under consideration, and returns the result of evaluating the expression on the right hand side. As in the programming language C, the symbols `==` and `=` stand for the *equality* operator and the *assignment* operator, respectively. Using the Promela description of `simple_guidance`, we can specify component `vertical_guidance` as another `inline` parameterized by environment event `env_ev` (cf. Table 5.3). The body of `vertical_guidance` is self-explanatory and similar to the one of Murϕ. It should only be noted that guard `else` is always executable and that expression `skip` leaves the current system state unchanged. Moreover, the Boolean functions `pitch_event`, `vs_event`, and `vga_event` are spelled out as `inline`s here.

TABLE 5.3
Specification of module vertical guidance *in Spin*

```
bit pitch_mode=cleared;  bit vs_mode=cleared;  bit vga_mode=cleared;

inline pitch_event(env_ev)      { env_ev==vs_pitch_wheel_changed }
inline vs_event(env_ev)         { env_ev==vs_switch_hit }
inline vga_event(env_ev)        { env_ev==ga_switch_hit || env_ev==ap_engaged_event || ... }
inline vertical_guidance(env_ev)
{ if :: pitch_event(env_ev) ->
        simple_guidance(activate, pitch_mode, pitch_signal);
        if :: pitch_signal==activated -> simple_guidance(deactivate,  vs_mode,  vs_signal);
                                         simple_guidance(deactivate, vga_mode, vga_signal)
           :: else                    -> skip
        fi
     :: vs_event(env_ev) ->
        simple_guidance(switch, vs_mode, vs_signal);
        if :: vs_signal==activated    -> simple_guidance(deactivate, pitch_mode, pitch_signal);
                                         simple_guidance(deactivate,   vga_mode,   vga_signal)
           :: vs_signal==deactivated  -> simple_guidance(  activate, pitch_mode, pitch_signal)
           :: else                    -> skip
        fi
     :: vga_event(env_ev) ->
        if :: env_ev==ga_switch_hit   -> simple_guidance(switch, vga_mode, vga_signal)
           :: else                    -> simple_guidance( clear, vga_mode, vga_signal)
        fi;
        if :: vga_signal==activated   -> simple_guidance(deactivate, pitch_mode, pitch_signal);
                                         simple_guidance(deactivate,    vs_mode,    vs_signal)
           :: vga_signal==deactivated -> simple_guidance(  activate, pitch_mode, pitch_signal)
           :: else                    -> skip
        fi
     :: else -> skip
  fi }
```

The verification technique we employed in Spin for reasoning about the flight guidance system, namely *assertions*, is similar to the one we used in Murϕ. More precisely, Promela's assertion statement `assert` aborts the state exploration conducted by Spin's verifier whenever its argument expression evaluates to false in some system state associated with the assertion statement. Our specification of the mode confusion properties are

TABLE 5.4
Specification of some mode confusion properties *in Spin*

```
bit old_pitch_mode=cleared;  bit old_vs_mode=cleared;  bit old_vga_mode=cleared;

/* check for response to pressing VS button             */
assert(!(old_vs_mode==cleared) || (vs_mode==active));
assert(!(old_vs_mode==active ) || (vs_mode==cleared));
/* search for ignored crew inputs (property violated)   */
assert(!(crew_input) || mode_change);
/* no unknown ignored crew inputs                       */
assert(!(crew_input && !(ignored_crew_input)) || mode_change);
/* search for indirect mode changes (property violated) */
assert(!(!(crew_input)) || !(mode_change));
/* no unknown indirect mode changes                     */
assert(!(!(crew_input) && !(indirect_mode_change)) || !(mode_change));
/* save the current mode values                         */
old_pitch_mode=pitch_mode;  old_vs_mode=vs_mode;  old_vga_mode=vga_mode;
```

depicted in Table 5.4, where '`!`', '`&&`', and '`||`' stand for the logical connectives *not*, *and*, and *or*, respectively. Moreover, the symbols `/*` and `*/` denote the begin and end of comments. In our specification, `crew_input`, `mode_change`, `ignored_crew_input`, and `indirect_mode_change`, which are defined as Boolean functions in Murϕ, are simply introduced via `#define`s. In order to encode expression `mode_change`, we have to keep – as in the Murϕ model – a copy of the 'old' values of all global variables of interest. Stating the mode confusion properties in Spin's linear-time logic would not have any advantages over using assertions. The reason is that Spin's version of linear-time logic does not include the *next-state operator*, as we used for specifying these properties in SMV. This is because many verification methods employed in Spin, such as partial order techniques, have essentially no beneficial effects when the next-state operator is present. The verification results returned by the Spin verifier are similar to the ones for Murϕ. The Spin model of the flight guidance system also possesses 242 states and 3 388 transitions (+ 1 "dummy" transition). Unfortunately, Spin crashes and core dumps when analyzing the invalid assertions *search for ignored crew inputs* and *search for indirect mode changes*. However, it still writes an error trace which can be fed into Spin's simulator. No other violated assertions were detected during the exhaustive state-space search which took under 2 seconds and required about 2.6 MBytes memory on a SUN SPARCstation 20. It should be pointed out that a previous effort by a NASA contractor to analyze a variant of the flight guidance system using Spin was unsuccessful because of an intractably large state space [24]. Unfortunately, from the report it is not clear what the exact causes are. We suspect that the manner in which the model was constructed is one of the main causes of the intractable state space, which was then checked for invariant properties using Spin's *bitstate hashing algorithm* [17].

Summarizing, the modeling and verification of our flight guidance system was feasible in Spin but less elegant than in Murϕ. This is mainly because of the lack of procedure and function constructs in Promela, which had to be encoded using inlines and `#define`s. However, our criticism is qualified by the fact that Spin is actually not intended for modeling and reasoning about *synchronous systems*. If one is interested in asynchronous, concurrent systems, Spin provides the process declaration `proctype` as a means for encapsulating system components. We would like to see a richer type system in Spin, which can handle more than one `mtype` definition. Type checking is a powerful tool for detecting inconsistencies and saves us a lot of time compared to checking specifications by hand. Also, we wish for the next-state operator to be included in Spin's linear-time logic. Similar to our comments for Murϕ we remark that this would cut the size of the state vector and Spin's memory requirements approximately in half. Especially useful to us were Spin's capabilities to simulate Promela models and to feed back error traces – *illustrating* the cause of an assertion's invalidity – into the simulator. Simulations helped us to identify the causes of ignored crew inputs and indirect mode changes in a very time-efficient manner. Beside the feature of monitoring variables, we found it useful that Spin highlights the part of the Promela description corresponding to the system state under investigation. The absence of rich simulation capabilities in Murϕ and SMV makes Spin the tool of choice for discovering design flaws interactively. Finally, Spin's nice graphical user interface, referred to as `Xspin`, distinguishes Spin from other verification tools.

6. Discussion and Related Work. In this section we discuss the most important strengths and weaknesses of each of the verification tools Murϕ, SMV, and Spin regarding our case study. We structure our discussion by separating the issues related to the tools' (i) system description languages, (ii) property description languages, and (iii) capabilities for system simulation and for animating diagnostic information.

The system description languages of all three verification tools allow us to model the deterministic, synchronous behavior of the flight guidance system, as well as the nondeterministic behavior of the system's environment. Especially, Murϕ's system description language proved to be very useful for the following reasons. First, Murϕ implements numerous language constructs and a rich type system, as found in many standard high-level imperative programming languages, such as Pascal. Second, it supports a modular programming style via parameterized procedures and functions. Third, it allows us to adapt the existing PVS specification of the mode logic in a straightforward manner [22]. One major difference between the languages is that Murϕ and Spin allow model encoding using a sequential algorithm, whereas SMV requires an algorithm description by parallel assignments. As a consequence, SMV has the feel of a low-level or hardware description language. However, SMV's `module` concept is slightly more elegant than Murϕ's procedure concept for our application, since mode variables can be declared within the module to which they belong and need not be declared outside. Regarding Spin's system description language Promela, one notices that it is actually designed to specify asynchronous systems, especially communication protocols. This is evident by the fact that it only offers the process declaration construct `proctype` for encapsulating code fragments. By using `inline` declarations we were able to circumvent this problem for our purposes. Finally, we want to mention one desired feature that the system description languages of all three tools are missing, namely the ability to organize the events of the flight guidance system in a *taxonomy*, e.g., by including *subtyping* in the description languages. The presence of such a concept would help us to naturally divide all events into lateral-mode and vertical-mode events, and further into Pitch events, HDG events, etc. This taxonomy was encoded in Murϕ and SMV using functions and in Spin using inlines.

Regarding the second issue concerning the property description languages of the three verification tools, we also identified several important differences. We first note that all of the mandatory and mode confusion properties of interest to us are invariants. Therefore, they can be stated as assertions and verified in reachability analysis tools, such as Murϕ, as well as more general model-checking tools, such as SMV and Spin. When specifying mode confusion properties, SMV's adaptation of the temporal logic CTL is most convenient, not because of its expressiveness which we hardly use, but since it allows one to implicitly refer to adjacent states in program paths using the 'next-state' operator `AX`. This is important for describing property `mode_change` which requires one to access the mode variables of adjacent states. In contrast to Murϕ and Spin, the encoding of mode confusion properties in SMV does not require the storage of old values of mode variables. Thereby, the size of the associated state vector is cut in half. Unfortunately, the 'next-state' operator is left out in Spin's version of linear-time logic. Therefore, we could employ Spin only as an assertion checker, similar to Murϕ. In addition to its suitable property description language, SMV's BDD-based model checker performed very well in our case study. Its high efficiency is due to the fact that mode logics have the characteristics of Boolean terms which can be represented in a very compact way using BDDs. However, the small state space of our example system precludes us from fairly comparing the run times of the Murϕ, SMV, and Spin verifiers. Finally, we remark again that Murϕ and Spin compile system and property descriptions into C++ and C-code, respectively, which may be considered as building special-purpose verifiers. This compilation process, however, is considerably slower than SMV's interpreter.

Regarding the third issue, only Spin provides rich features related to system simulation and to animation of diagnostic information. System simulation is especially useful when being combined with diagnostic information. Each tool returns an error trace whenever a desired system property is invalidated in the model under consideration. More precisely, Murϕ and SMV output a textual description of an error trace, which displays the global variables' assignments at all states of this trace, and allow for textual, interactive simulations. Spin, however, is able to animate error traces using *message sequence charts*, *time sequence panels*, and *data value panels* which are integrated in its nice *graphical user interface*, known as `Xspin`. In our case study dealing with a synchronous, single-process system, only the data value panel was of use. However, this feature, together with the ability to highlight the source code line corresponding to the current state in the simulation, enabled us to detect sources of mode confusion in a very time-efficient manner compared to SMV and Murϕ, and especially when compared to the studies of failed proof subgoals in PVS.

Finally, related work other than the PVS case study regarding the flight guidance system [5] should be mentioned. The CoRE [9] and SCR [14] specifications of the flight guidance system [20, 21] were intended for illustrating the utility of the methods for specifying new generations of systems in a more rigorous, consistent, and structured way. Especially, they should replace the traditional custom of specifying such systems in plain English. In contrast to this paper, the SCR and CoRE specifications were not subject to any automated analysis tools, although some tool support for them exists [14]. The well-known Z specification standard [31] was applied to the flight guidance system in order to formally express concepts that appear rather informally in CoRE [10], such as the semantics of *continuous variables*. Recently, tools supporting the analysis of Z specifications emerged, e.g., Z/EVES [27] which interfaces Z to the theorem prover EVES. This tool was applied to the Z specification of the flight guidance system for validating some of the mandatory properties mentioned also in this paper, as well as for proving disjointness and completeness of table entries and for determinism checks. The gained experiences with Z/EVES are very similar to the ones made with PVS [25]. ObjecTime [28] is an environment for testing and simulation and was used as the driving engine of a partial flight deck visualization of the flight guidance system's behavior [22].

7. Conclusions and Future Work. This paper advocates the use of state-exploration and model-checking techniques for analyzing flight guidance systems with respect to causes of mode confusion. Compared to theorem provers, model-checking tools are able to verify invariants *automatically.* When weighting the strengths of the verification tools Murϕ, SMV, and Spin with respect to our application, it turned out that these are complementary. Murϕ has the most pleasant system description language, including a rich type system and allows for high-level specifications. SMV's adaptation of the temporal logic CTL as property description language supports the convenient specification of mode confusion properties. Spin's capability of animating diagnostic information, which is returned from unsuccessful verification attempts, is very useful. We hope that our experiences might give tool developers some useful ideas for combining the strengths of Murϕ, SMV, and Spin in a single tool.

Regarding future work, our case study should be extended to include more components of today's digital flight decks and to explore other interesting properties related to mode confusion. Also, the integration of verification tools with state-of-the-art specification languages, such as UML [28], must be a primary goal in order to make formal verification techniques accessible to engineers in industry or at applied research labs.

We thank Ricky Butler and Steve Miller for many enlightening discussions about mode confusion, as well as Ben Di Vito, Michael Mendler, and César Muñoz for carefully proofreading a draft of this paper.

REFERENCES

[1] G. Berry and G. Gonthier, *The ESTEREL synchronous programming language: Design, semantics, implementation*, Science of Computer Programming, 19 (1992), pp. 87–152.

[2] C. Billings, *Aviation Automation: The Search for a Human Centered Approach*, Lawrence Erlbaum Associates, Mahwah, NJ, USA, 1996.

[3] R. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers, C-35 (1986).

[4] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation, 98 (1992), pp. 142–170.

[5] R. Butler, S. Miller, J. Potts, and V. Carreño, *A formal methods approach to the analysis of mode confusion*, in Seventh Digital Avionics Systems Conference (DASC '98), Bellevue, WA, USA, November 1998, IEEE. Proccedings available on CD-ROM.

[6] E. Clarke, E. Emerson, and A. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems, 8 (1986), pp. 244–263.

[7] D. Dill, *The Murphi verification system*, in Computer Aided Verification (CAV '96), R. Alur and T. Henzinger, eds., vol. 1102 of Lecture Notes in Computer Science, New Brunswick, NJ, USA, July 1996, Springer-Verlag, pp. 390–393.

[8] E. Emerson, *Temporal and modal logic*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, North-Holland, 1990, pp. 995–1072.

[9] S. Faulk, L. Finneran, J. Kirby, and A. Moini, *Consortium requirements engineering guidebook*, Tech. Report SPC-92060-CMC, Software Productivity Consortium, Herndon, VA, USA, December 1993.

[10] F. Fung and D. Jamsek, *Formal specification of a flight guidance system*, NASA Contractor Report NASA/CR-1998-206915, Odyssey Research Associates, Ithaca, NY, USA, January 1998.

[11] R. Gerth, D. Peled, M. Vardi, and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, in Protocol Specification Testing and Verification (PSTV '95), Warsaw, Poland, 1995, Chapman & Hall, pp. 3–18.

[12] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, vol. 1032 of Lecture Notes in Computer Science, Springer-Verlag, 1996.

[13] D. Harel, *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, 8 (1987), pp. 231–274.

[14] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw, *SCR*: A toolset for specifying and analyzing requirements*, in COMPASS '95: Tenth Annual Conference on Computer Assurance, Gaithersburg, MD, USA, 1995, National Institute of Standards and Technology, pp. 109–122.

[15] G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, 1991.

[16] ——, *The model checker Spin*, IEEE Transactions on Software Engineering, 23 (1997), pp. 279–295. Special issue on Formal Methods in Software Practice.

[17] ——, *An analysis of bitstate hashing*, Formal Methods in System Design, 13 (1998), pp. 287–305.

[18] N. Leveson, L. Pinnel, S. Sandys, S. Koga, and J. Reese, *Analyzing software specifications for mode confusion potential*, in Workshop on Human Error and System Development, Glasgow, UK, March 1997.

[19] K. McMillan, *Symbolic Model Checking: An Approach to the State-Explosion Problem*, PhD thesis, Carnegie-Mellon University, 1992.

[20] S. Miller, *Specifying the mode logic of a flight guidance system in CoRE and SCR*, in Second Workshop on Formal Methods in Software Practice (FMSP '98), M. Ardis, ed., Clearwater Beach, FL, USA, March 1998, ACM Press, pp. 44–53.

[21] S. Miller and K. Hoech, *Specifying the mode logic of a flight guidance system in CoRE*, Tech. Report WP-97-2011, Rockwell Collins, Inc., November 1997.

[22] S. Miller and J.N.Potts, *Detecting mode confusion through formal modeling and analysis*, NASA Contractor Report NASA/CR-1999-208971, Advanced Technology Center, Rockwell Collins, Inc., January 1999.

[23] *Murphi*. Project Page at http://sprout.stanford.edu/dill/murphi.html.

[24] D. Naydich and J. Nowakowski, *Flight guidance system validation using Spin*, NASA Contractor Report NASA/CR-1998-208434, Odyssey Research Associates, Ithaca, NY, USA, June 1998.

[25] S. Owre, J. Rushby, N. Shankar, and F. von Henke, *Formal verification for fault-tolerant systems: Prolegomena to the design of PVS*, IEEE Transactions on Software Engineering, 21 (1995), pp. 107–125.

[26] A. Pnueli, *Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends*, in Current Trends in Concurrency, vol. 224 of Lecture Notes in Computer Science, Springer-Verlag, 1986, pp. 510–584.

[27] M. Saaltink, *The Z/EVES system*, in ZUM '97, the Z Formal Specification Notation: 10th International Conference of Z Users, J. Bowen, M. Hinchey, and D. Till, eds., vol. 1212 of Lecture Notes in Computer Science, Reading, UK, April 1997, Springer-Verlag, pp. 72–85.

[28] B. Selic and J. Rumbaugh, *Using UML for modeling complex real-time systems*, tech. report, ObjecTime Limited, 1998.

[29] *SMV*. Project Page at http://www.cs.cmu.edu/˜modelcheck/smv.html.

[30] *Spin*. Project Page at http://netlib.bell-labs.com/netlib/spin/whatispin.html.

[31] J. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge Tracts in Theoretical Computer Science 3, Cambridge University Press, Cambridge, UK, 1988.

Appendix A. Specification and Verification Using Murϕ.

A.1. Full Model of the Mode Logic.

```
TYPE env_events : ENUM { hdg_switch_hit,             nav_switch_hit,
                         nav_armed_long_enough_event, nav_track_cond_met_event,
                         ga_switch_hit,              vs_pitch_wheel_changed,
                         vs_switch_hit,              fd_switch_hit,
                         overspeed_start,            overspeed_stop,
                         ap_engaged_event,           ap_disengaged_event,
                         sync_switch_pressed,        sync_switch_released
                       };

TYPE sg_mode    : ENUM { sg_cleared, sg_active };
TYPE sg_signals : ENUM { sg_null,    sg_activated, sg_deactivated };
TYPE sg_events  : ENUM { sg_nil,     sg_activate,  sg_deactivate, sg_switch,  sg_clear };

TYPE ag_mode    : ENUM { ag_cleared, ag_track, ag_armed_initial, ag_armed_long_enough };
TYPE ag_signals : ENUM { ag_null, ag_activated,   ag_deactivated };
TYPE ag_events  : ENUM { ag_nil,  ag_activate,    ag_deactivate, ag_switch, ag_clear,
                         ag_armed_long_enough_ev, ag_track_cond_met };

TYPE fd_mode    : ENUM { fd_off,    fd_cues,       fd_no_cues };
TYPE fd_signals : ENUM { fd_null,   fd_turned_on,  fd_turned_off };
TYPE fd_events  : ENUM { fd_nil,    fd_force_cues, fd_turn_on,   fd_switch, fd_turn_off };

TYPE ag_state   : RECORD  mode : ag_mode;  track_cond_met : boolean;  END;

-- variables controled by the environment ---------------------------------------------------

VAR  overspeed : boolean;  ap_engaged : boolean;

-- mode variables ---------------------------------------------------------------------------

VAR  pitch, old_pitch : sg_mode;   vs,  old_vs  : sg_mode;  vga, old_vga : sg_mode;
     roll,  old_roll  : sg_mode;   hdg, old_hdg : sg_mode;  lga, old_lga   : sg_mode;
     nav,   old_nav   : ag_state;  fd,  old_fd  : fd_mode;

-- auxiliary functions, building a taxonomy on events ----------------------------------------

FUNCTION hdg_event(env_ev:env_events) : boolean;
BEGIN
  IF env_ev=hdg_switch_hit
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;
```

```
FUNCTION nav_event(env_ev:env_events) : boolean;
BEGIN
  IF (env_ev=nav_switch_hit)           | (env_ev=nav_armed_long_enough_event) |
     (env_ev=nav_track_cond_met_event)
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;


FUNCTION lga_event(env_ev:env_events) : boolean;
BEGIN
  IF (env_ev=ga_switch_hit) | (env_ev=ap_engaged_event) | (env_ev=sync_switch_pressed)
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;


FUNCTION pitch_event(env_ev:env_events) : boolean;
BEGIN
  IF env_ev=vs_pitch_wheel_changed
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;


FUNCTION vs_event(env_ev:env_events) : boolean;
BEGIN
  IF env_ev=vs_switch_hit
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;


FUNCTION vga_event(env_ev:env_events) : boolean;
BEGIN
  IF (env_ev=ga_switch_hit) | (env_ev=ap_engaged_event) | (env_ev=sync_switch_pressed)
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;
```

```
FUNCTION lateral_mode_requested(env_ev:env_events) : boolean;
BEGIN
  IF (env_ev=hdg_switch_hit) | (env_ev=nav_switch_hit) | (env_ev=ga_switch_hit)
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;

FUNCTION vertical_mode_requested(env_ev:env_events) : boolean;
BEGIN
  IF (env_ev=vs_switch_hit) | (env_ev=ga_switch_hit)
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;

FUNCTION flight_director_event(env_ev:env_events) : boolean;
BEGIN
  IF (env_ev=ap_engaged_event) | (env_ev=fd_switch_hit) | (env_ev=overspeed_start) |
     lateral_mode_requested(env_ev) | vertical_mode_requested(env_ev)
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;

-- abstract data type module simple guidance -----------------------------------------------

PROCEDURE simple_guidance(VAR mode:sg_mode; event:sg_events; VAR signal:sg_signals);
BEGIN
  IF mode=sg_cleared THEN
    SWITCH event
      CASE sg_nil        :                     signal := sg_null;
      CASE sg_activate   : mode := sg_active;  signal := sg_activated;
      CASE sg_deactivate :                     signal := sg_null;
      CASE sg_switch     : mode := sg_active;  signal := sg_activated;
      CASE sg_clear      :                     signal := sg_null;
     END;
  ELSE
    SWITCH event
      CASE sg_nil        :                     signal := sg_null;
      CASE sg_activate   :                     signal := sg_null;
      CASE sg_deactivate : mode := sg_cleared; signal := sg_null;
      CASE sg_switch     : mode := sg_cleared; signal := sg_deactivated;
      CASE sg_clear      : mode := sg_cleared; signal := sg_deactivated;
    END;
  END;
END;
```

```
-- abstract data object module arming guidance --------------------------------------------

PROCEDURE arming_guidance(event:ag_events; VAR signal:ag_signals);
BEGIN
  IF nav.mode=ag_cleared THEN
    SWITCH event
      CASE ag_nil                 :                                 signal := ag_null;
      CASE ag_activate            : nav.mode := ag_armed_initial;  signal := ag_activated;
      CASE ag_deactivate          :                                 signal := ag_null;
      CASE ag_switch              : nav.mode := ag_armed_initial;  signal := ag_activated;
      CASE ag_clear               :                                 signal := ag_null;
      CASE ag_armed_long_enough_ev :                                signal := ag_null;
      CASE ag_track_cond_met      : nav.track_cond_met := true;    signal := ag_null;
    END;
  ELSE
    SWITCH event
      CASE ag_nil                 :                           signal := ag_null;
      CASE ag_activate            :                           signal := ag_null;
      CASE ag_deactivate          : nav.mode := ag_cleared;  signal := ag_null;
      CASE ag_switch              : nav.mode := ag_cleared;  signal := ag_deactivated;
      CASE ag_clear               : nav.mode := ag_cleared;  signal := ag_deactivated;
      CASE ag_armed_long_enough_ev : IF (nav.mode=ag_armed_initial) & nav.track_cond_met
                                   THEN
                                     nav.mode := ag_track;  signal := ag_null;
                                   ELSIF (nav.mode=ag_armed_initial) & !nav.track_cond_met
                                   THEN
                                     nav.mode := ag_armed_long_enough;  signal := ag_null;
                                   ELSE
                                                                        signal := ag_null;
                                   END;
      CASE ag_track_cond_met      : IF nav.mode=ag_armed_long_enough THEN
                                     nav.mode           := ag_track;    signal := ag_null;
                                     nav.track_cond_met := true;
                                   ELSE
                                     nav.track_cond_met := true;        signal := ag_null;
                                   END;
END; END; END;

-- function module lateral guidance & auxiliary functions -----------------------------

FUNCTION hdg_conv(env_ev:env_events) : sg_events;
BEGIN
  SWITCH env_ev
    CASE hdg_switch_hit : RETURN sg_switch;
    ELSE                  RETURN sg_nil;
  END;
END;
```

```
FUNCTION nav_conv(env_ev:env_events) : ag_events;
BEGIN
  SWITCH env_ev
    CASE nav_switch_hit              : RETURN ag_switch;
    CASE nav_track_cond_met_event    : RETURN ag_track_cond_met;
    CASE nav_armed_long_enough_event : RETURN ag_armed_long_enough_ev;
    ELSE                               RETURN ag_nil;
  END;
END;

FUNCTION lga_conv(env_ev:env_events) : sg_events;
BEGIN
  SWITCH env_ev
    CASE ga_switch_hit       : RETURN sg_activate;
    CASE ap_engaged_event    : RETURN sg_clear;
    CASE sync_switch_pressed : RETURN sg_clear;
    ELSE                       RETURN sg_nil;
  END;
END;

PROCEDURE lateral_guidance(env_ev:env_events);
VAR roll_sig, hdg_sig, lga_sig : sg_signals; nav_sig : ag_signals;
BEGIN  CLEAR roll_sig;  CLEAR hdg_sig;  CLEAR lga_sig;  CLEAR nav_sig;
  IF hdg_event(env_ev) THEN
    simple_guidance(hdg, hdg_conv(env_ev), hdg_sig);
    IF hdg_sig=sg_activated THEN      simple_guidance(roll, sg_deactivate, roll_sig);
                                      simple_guidance(lga,  sg_deactivate, lga_sig );
                                      arming_guidance(      ag_deactivate, nav_sig );
    ELSIF hdg_sig=sg_deactivated THEN simple_guidance(roll, sg_activate, roll_sig  );
    END;
  ELSIF nav_event(env_ev) THEN
    arming_guidance(     nav_conv(env_ev), nav_sig);
    IF nav_sig=ag_activated THEN      simple_guidance(roll, sg_deactivate, roll_sig);
                                      simple_guidance(hdg,  sg_deactivate, hdg_sig );
                                      simple_guidance(lga,  sg_deactivate, lga_sig );
    ELSIF nav_sig=ag_deactivated THEN simple_guidance(roll, sg_activate, roll_sig  );
    END;
  ELSIF lga_event(env_ev) THEN
    simple_guidance(lga, lga_conv(env_ev), lga_sig);
    IF lga_sig=sg_activated THEN      simple_guidance(roll, sg_deactivate, roll_sig);
                                      simple_guidance(hdg,  sg_deactivate, hdg_sig );
                                      arming_guidance(      ag_deactivate, nav_sig );
    ELSIF lga_sig=sg_deactivated THEN simple_guidance(roll, sg_activate, roll_sig  );
    END;
  END;
END;
```

```
-- function module vertical guidance & auxiliary functions-----------------------------------

FUNCTION pitch_conv(env_ev:env_events) : sg_events;
BEGIN
  SWITCH env_ev
    CASE vs_pitch_wheel_changed : RETURN sg_activate;
    ELSE                          RETURN sg_nil;
END; END;

FUNCTION vs_conv(env_ev:env_events) : sg_events;
BEGIN
  SWITCH env_ev
    CASE vs_switch_hit : RETURN sg_switch;
    ELSE                 RETURN sg_nil;
END; END;

FUNCTION vga_conv(env_ev:env_events) : sg_events;
BEGIN
  SWITCH env_ev
    CASE ga_switch_hit       : RETURN sg_switch;
    CASE ap_engaged_event    : RETURN sg_clear;
    CASE sync_switch_pressed : RETURN sg_clear;
    ELSE                       RETURN sg_nil;
END; END;

PROCEDURE vertical_guidance(env_ev:env_events);
VAR pitch_sig, vs_sig, vga_sig : sg_signals;
BEGIN  CLEAR pitch_sig;  CLEAR vs_sig;  CLEAR vga_sig;
  IF pitch_event(env_ev) THEN
    simple_guidance(pitch, pitch_conv(env_ev), pitch_sig);
    IF pitch_sig=sg_activated THEN    simple_guidance(vs,  sg_deactivate, vs_sig    );
                                      simple_guidance(vga, sg_deactivate, vga_sig   );
    END;
  ELSIF vs_event(env_ev) THEN
    simple_guidance(vs, vs_conv(env_ev), vs_sig);
    IF vs_sig=sg_activated THEN       simple_guidance(pitch, sg_deactivate, pitch_sig);
                                      simple_guidance(vga,   sg_deactivate, vga_sig  );
    ELSIF vs_sig=sg_deactivated THEN  simple_guidance(pitch, sg_activate, pitch_sig  );
    END;
  ELSIF vga_event(env_ev) THEN
    simple_guidance(vga, vga_conv(env_ev), vga_sig);
    IF vga_sig=sg_activated THEN      simple_guidance(pitch, sg_deactivate, pitch_sig);
                                      simple_guidance(vs,    sg_deactivate, vs_sig   );
    ELSIF vga_sig=sg_deactivated THEN simple_guidance(pitch, sg_activate, pitch_sig  );
    END;
  END;
END;
```

```
-- abstract data object module flight director ---------------------------------------------

PROCEDURE flight_director(event:fd_events; VAR signal:fd_signals);
BEGIN
  IF event=fd_nil THEN
    signal := fd_null;
  ELSIF fd=fd_off THEN
    SWITCH event
      CASE fd_force_cues : fd := fd_cues; signal := fd_turned_on;
      CASE fd_turn_on    : fd := fd_cues; signal := fd_turned_on;
      CASE fd_switch     : fd := fd_cues; signal := fd_turned_on;
      CASE fd_turn_off   :                signal := fd_null;
    END;
  ELSIF fd=fd_cues THEN
    SWITCH event
      CASE fd_force_cues : signal := fd_null;
      CASE fd_turn_on    : signal := fd_null;
      CASE fd_switch     : IF overspeed | ap_engaged THEN
                             fd := fd_no_cues; signal := fd_null;
                           ELSE
                             fd := fd_off;     signal := fd_turned_off;
                           END;
      CASE fd_turn_off   : IF overspeed | ap_engaged THEN
                             fd := fd_no_cues; signal := fd_null;
                           ELSE
                             fd := fd_off;     signal := fd_turned_off;
                           END;
    END;
  ELSE
    SWITCH event
      CASE fd_force_cues : fd := fd_cues; signal := fd_null;
      CASE fd_turn_on    : fd := fd_cues; signal := fd_null;
      CASE fd_switch     : IF overspeed | ap_engaged THEN
                             fd := fd_cues;    signal := fd_null;
                           ELSE
                             fd := fd_off;     signal := fd_turned_off;
                           END;
      CASE fd_turn_off   : IF overspeed | ap_engaged THEN
                                               signal := fd_null;
                           ELSE
                             fd := fd_off;     signal := fd_turned_off;
                           END;
    END;
  END;
END;
```

```
-- mode confusion properties as assertions & auxiliary functions -----------------------------

FUNCTION crew_input(env_ev:env_events) : boolean;
BEGIN
  IF (env_ev=ap_engaged_event)       | (env_ev=sync_switch_pressed)    |
     (env_ev=sync_switch_released)   | (env_ev=fd_switch_hit)          |
     lateral_mode_requested(env_ev)  | vertical_mode_requested(env_ev) |
     (env_ev=vs_pitch_wheel_changed)
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;


FUNCTION ignored_crew_input(env_ev:env_events) : boolean;
BEGIN
  IF ((env_ev=ap_engaged_event)       &
        !((old_lga=sg_active) | (old_vga=sg_active))) |
--   ((env_ev=ga_switch_hit)          &                 -- PVS model too strong
--        (old_lga=sg_active) & (old_vga=sg_active))  | -- (may be left out)
     ((env_ev=sync_switch_pressed)    &
        !((old_lga=sg_active) | (old_vga=sg_active))) |
--   ((env_ev=sync_switch_pressed)    &                 -- PVS model too strong
--     (old_fd=fd_off))                               | -- (may be left out)
     (env_ev=sync_switch_released)                    |
     ((env_ev=vs_pitch_wheel_changed) &
       (old_fd=fd_off))                               |
     ((env_ev=vs_pitch_wheel_changed) &
       (old_pitch=sg_active))
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;


FUNCTION indirect_mode_change(env_ev:env_events) : boolean;
BEGIN
  IF ((env_ev=overspeed_start)             & !(old_fd=fd_cues))                   |
     ((env_ev=nav_armed_long_enough_event) & (old_nav.mode=ag_armed_initial    )) |
     ((env_ev=nav_track_cond_met_event)    & (old_nav.mode=ag_armed_long_enough))
  THEN RETURN true;
  ELSE RETURN false;
  END;
END;
```

```
PROCEDURE mode_confusion_properties(env_ev:env_events);
BEGIN
  ALIAS
    mode_change : fd   != old_fd   | pitch != old_pitch |  vs != old_vs  | vga != old_vga |
                  roll != old_roll | hdg   != old_hdg   | lga != old_lga |
                  nav.mode != old_nav.mode;
  DO
    SWITCH env_ev
      CASE hdg_switch_hit              :
        -- check for response to pressing HDG button
        assert (old_hdg=sg_cleared -> hdg =sg_active) "hdg_selected and hdg_toggle_1";
        assert (old_hdg=sg_active  -> roll=sg_active) "hdg_deselected";
        assert (old_hdg=sg_active  -> hdg=sg_cleared) "hdg_toggle_2";

      CASE nav_switch_hit              :
        -- check for response to pressing NAV button
        assert (old_nav.mode=ag_cleared -> (nav.mode=ag_armed_initial) |
                 (nav.mode=ag_armed_long_enough) | (nav.mode=ag_track))
               "nav_selected and nav_toggle_1";
        assert (((old_nav.mode=ag_armed_initial) | (old_nav.mode=ag_armed_long_enough) |
                 (old_nav.mode=ag_track)) -> roll=sg_active)      "nav_deselected";
        assert (((old_nav.mode=ag_armed_initial) | (old_nav.mode=ag_armed_long_enough) |
                 (old_nav.mode=ag_track)) -> nav.mode=ag_cleared) "nav_toggle_2";

      CASE vs_switch_hit               :
        -- check for response to pressing VS button
        assert (old_vs=sg_cleared -> vs   =sg_active) "vs_selected and vs_toggle_1";
        assert (old_vs=sg_active  -> pitch=sg_active) "vs_deselected";
        assert (old_vs=sg_active  -> vs  =sg_cleared) "vs_toggle_2";

      CASE fd_switch_hit               :
        -- check for response to pressing the FD button
        assert (old_fd=fd_off -> fd=fd_cues)
               "fd_off";
        assert ((!(old_fd=fd_off)    & !(ap_engaged | overspeed)) -> fd=fd_off)     "fd_on";
        assert (((old_fd=fd_cues)    &  (ap_engaged | overspeed)) -> fd=fd_no_cues) "fd_cues";
        assert (((old_fd=fd_no_cues) &  (ap_engaged | overspeed)) -> fd=fd_cues)    "fd_no_cues";

    END;

    -- search for ignored crew inputs
    -- assert (crew_input(env_ev) -> mode_change) "search_for_ignored_crew_inputs";
    -- property violated

    -- no unknown ignored crew inputs
    assert ((crew_input(env_ev) & !(ignored_crew_input(env_ev))) -> mode_change)
           "no_unknown_ignored_crew_inputs";
    -- search for indirect mode changes
    -- assert (!(crew_input(env_ev)) -> !mode_change) "search_for_indirect_mode_changes";
    -- property violated

    -- no unknown indirect mode changes
    assert ((!(crew_input(env_ev)) & !(indirect_mode_change(env_ev))) -> !mode_change)
           "no_unknown_indirect_mode_changes";
  END;

  -- update old state variables
  old_pitch := pitch; old_vs  := vs;  old_vga := vga;  old_roll := roll;
  old_hdg   := hdg;   old_lga := lga; old_fd  := fd;   old_nav  := nav;
END;

-----------------------------------------------------------------------------------------------

PROCEDURE clear_all_modes();
BEGIN
  pitch := sg_cleared;  vs  := sg_cleared;  vga := sg_cleared;       roll := sg_cleared;
  hdg   := sg_cleared;  lga := sg_cleared;  nav.mode := ag_cleared;
END;

PROCEDURE select_default_mode();
BEGIN
  pitch := sg_active;  roll  := sg_active;
END;

PROCEDURE process_external_event(env_ev:env_events);
BEGIN
  SWITCH env_ev
    CASE ap_engaged_event    : ap_engaged := true;
    CASE ap_disengaged_event : ap_engaged := false;
    CASE overspeed_start     : overspeed  := true;
    CASE overspeed_stop      : overspeed  := false;
  END;
END;

FUNCTION fd_event(env_ev:env_events) : fd_events;
BEGIN
  IF env_ev=ap_engaged_event            THEN RETURN fd_turn_on;
  ELSIF lateral_mode_requested(env_ev)  THEN RETURN fd_turn_on;
  ELSIF vertical_mode_requested(env_ev) THEN RETURN fd_turn_on;
  ELSIF env_ev=fd_switch_hit            THEN RETURN fd_switch;
  ELSIF env_ev=overspeed_start          THEN RETURN fd_force_cues;
  ELSE                                       RETURN fd_nil;
  END;
END;
```

```
PROCEDURE process_fd_event(env_ev:env_events);
VAR fd_sig : fd_signals;
BEGIN
  IF flight_director_event(env_ev) THEN
    flight_director(fd_event(env_ev),fd_sig);
    IF fd_sig=fd_turned_off   THEN clear_all_modes();
    ELSIF fd_sig=fd_turned_on THEN select_default_mode();
    END;
  END;
END;

PROCEDURE process_flight_mode_event(env_ev:env_events);
BEGIN
  IF !(fd=fd_off) THEN lateral_guidance(env_ev); vertical_guidance(env_ev); END;
END;

PROCEDURE fgs(env_ev:env_events);
BEGIN
  process_external_event(env_ev);    process_fd_event(env_ev);
  process_flight_mode_event(env_ev); mode_confusion_properties(env_ev);
END;

-- model of the environment using rules -----------------------------------------------------

RULE "hdg_switch_hit"              BEGIN fgs(hdg_switch_hit);              END;
RULE "nav_switch_hit"              BEGIN fgs(nav_switch_hit);              END;
RULE "nav_armed_long_enough_event" BEGIN fgs(nav_armed_long_enough_event); END;
RULE "nav_track_cond_met_event"    BEGIN fgs(nav_track_cond_met_event);    END;
RULE "ga_switch_hit"               BEGIN fgs(ga_switch_hit);               END;
RULE "vs_pitch_wheel_changed"      BEGIN fgs(vs_pitch_wheel_changed);      END;
RULE "vs_switch_hit"               BEGIN fgs(vs_switch_hit);               END;
RULE "fd_switch_hit"               BEGIN fgs(fd_switch_hit);               END;
RULE "overspeed_start"             BEGIN fgs(overspeed_start);             END;
RULE "overspeed_stop"              BEGIN fgs(overspeed_stop);              END;
RULE "ap_engaged_event"            BEGIN fgs(ap_engaged_event);            END;
RULE "ap_disengaged_event"         BEGIN fgs(ap_disengaged_event);         END;
RULE "sync_switch_pressed"         BEGIN fgs(sync_switch_pressed);         END;
RULE "sync_switch_released"        BEGIN fgs(sync_switch_released);        END;

-- start state -------------------------------------------------------------------------------

STARTSTATE
BEGIN
  overspeed := false;  ap_engaged := false;

  CLEAR pitch;  CLEAR vs;  CLEAR vga;  CLEAR roll;  CLEAR hdg;  CLEAR lga;
  CLEAR nav.mode; nav.track_cond_met := false;      CLEAR fd;

  CLEAR old_pitch;  CLEAR old_vs;  CLEAR old_vga;  CLEAR old_roll;        CLEAR old_hdg;
  CLEAR old_lga;    CLEAR old_nav.mode; old_nav.track_cond_met := false;  CLEAR old_fd;
END;

-- mandatory properties as invariants -------------------------------------------------------

ALIAS
  nav_active : (nav.mode=ag_armed_initial) | (nav.mode=ag_armed_long_enough) |
               (nav.mode=ag_track);
DO

-- the flight director is on if the autopilot is engaged

INVARIANT "fd_on_if_ap_engaged"
  ap_engaged -> !(fd=fd_off);

-- at least one lateral mode is active iff the flight director is on

INVARIANT "at_least_one_lateral_mode_active"
  (!(fd=fd_off) -> (roll=sg_active | hdg=sg_active | lga=sg_active | nav_active)) &
  ((roll=sg_active | hdg=sg_active | lga=sg_active | nav_active ) -> !(fd=fd_off));

-- there is never more than one lateral mode active

INVARIANT "at_most_one_lateral_mode_active"
  ((lga=sg_active)  -> (roll=sg_cleared & hdg=sg_cleared & nav.mode=ag_cleared)) &
  ((roll=sg_active) -> (lga =sg_cleared & hdg=sg_cleared & nav.mode=ag_cleared)) &
  ((hdg=sg_active)  -> (roll=sg_cleared & lga=sg_cleared & nav.mode=ag_cleared)) &
  (nav_active       -> (roll=sg_cleared & hdg=sg_cleared & lga=sg_cleared));

-- at least one vertical mode is active iff the flight director is on

INVARIANT "at_least_one_vertical_mode_active"
  (!(fd=fd_off) -> (vga=sg_active | vs=sg_active | pitch=sg_active)) &
  ((vga=sg_active | vs=sg_active | pitch=sg_active) -> !(fd=fd_off));

-- at most one vertical mode is active

INVARIANT "at_most_one_vertical_mode_active"
  (vga  =sg_active -> (pitch=sg_cleared & vs =sg_cleared)) &
  (vs   =sg_active -> (pitch=sg_cleared & vga=sg_cleared)) &
  (pitch=sg_active -> (vga  =sg_cleared & vs =sg_cleared));
```

```
-- if the flight director is off, all modes must be cleared

INVARIANT "fd_off_implies_all_modes_cleared"
  (fd=fd_off -> (pitch=sg_cleared &  vs=sg_cleared & vga=sg_cleared &
                 roll =sg_cleared & hdg=sg_cleared & lga=sg_cleared & nav.mode=ag_cleared)
  );

-- the default modes are active if the flight director is on and all other modes are cleared

INVARIANT "default_modes"
  ((!(fd=fd_off) &  vs=sg_cleared & vga=sg_cleared & hdg=sg_cleared &
                  lga=sg_cleared & nav.mode=ag_cleared
   ) -> (pitch=sg_active & roll=sg_active));

END;

-- -----------------------------------------------------------------------------------------------
```

A.2. Output of the Murϕ verifier.

```
This program should be regarded as a DEBUGGING aid, not as a
certifier of correctness.
Call with the -l flag or read the license file for terms
and conditions of use.
Run this program with "-h" for the list of options.

Bugs, questions, and comments should be directed to
"murphi@verify.stanford.edu".

Murphi compiler last modified date: Jan 29 1999
Include files   last modified date: Jan 29 1999
==========================================================================

==========================================================================
Murphi Release 3.1
Finite-state Concurrent System Verifier.

Copyright (C) 1992 - 1999 by the Board of Trustees of
Leland Stanford Junior University.

==========================================================================

Protocol: fgs

Algorithm:
Verification by breadth first search.
with symmetry algorithm 3 -- Heuristic Small Memory Normalization
with permutation trial limit 10.
```

```
Memory usage:

* The size of each state is 160 bits (rounded up to 20 bytes).
* The memory allocated for the hash table and state queue is
  8 Mbytes.
  With two words of overhead per state, the maximum size of
  the state space is 327869 states.
   * Use option "-k" or "-m" to increase this, if necessary.
* Capacity in queue for breadth-first search: 32786 states.
   * Change the constant gPercentActiveStates in mu_prolog.inc
     to increase this, if necessary.

Warning: No trace will not be printed in the case of protocol errors!
         Check the options if you want to have error traces.

==========================================================================

Status:

No error found.

State Space Explored:

242 states, 3388 rules fired in 1.80s.

Rules Information:

Fired 242 times - Rule "sync_switch_released"
Fired 242 times - Rule "sync_switch_pressed"
Fired 242 times - Rule "ap_disengaged_event"
Fired 242 times - Rule "ap_engaged_event"
Fired 242 times - Rule "overspeed_stop"
Fired 242 times - Rule "overspeed_start"
Fired 242 times - Rule "fd_switch_hit"
Fired 242 times - Rule "vs_switch_hit"
Fired 242 times - Rule "vs_pitch_wheel_changed"
Fired 242 times - Rule "ga_switch_hit"
Fired 242 times - Rule "nav_track_cond_met_event"
Fired 242 times - Rule "nav_armed_long_enough_event"
Fired 242 times - Rule "nav_switch_hit"
Fired 242 times - Rule "hdg_switch_hit"
```

Appendix B. Specification and Verification Using SMV.

B.1. Full Model of the Mode Logic.

```
MODULE simple_guidance(activate, deactivate, switch, clear, fd_is_on)

VAR
  mode : {cleared, active};

ASSIGN
  init(mode) := cleared;
  next(mode) := case
                  !fd_is_on                : mode;
                  deactivated | deactivate : cleared;
                  activated                : active;
                  1                        : mode;
                esac;

DEFINE
  activated   := (mode=cleared) & (activate | switch);
  deactivated := (mode=active ) & (clear    | switch);

-- ------------------------------------------------------------------------------------------

MODULE arming_guidance(activate, deactivate, switch, clear, track_cond_met_event,
                       armed_long_enough_event, fd_is_on)

VAR
  mode           : {cleared, track, armed_initial, armed_long_enough};
  track_cond_met : boolean;

ASSIGN
  init(track_cond_met) := 0;
  next(track_cond_met) := track_cond_met | track_cond_met_event;

  init(mode) := cleared;
  next(mode) := case
                  !fd_is_on                                : mode;
                  deactivated | deactivate                 : cleared;
                  (mode=armed_long_enough) & track_cond    : track;
                  (mode=armed_initial)     & track_cond  &
                    armed_long_enough_event                : track;
                  activated                                : armed_initial;
                  (mode=armed_initial)     & !track_cond &
                    armed_long_enough_event                : armed_long_enough;
                  1                                        : mode;
                esac;
```

```
DEFINE
  track_cond  := track_cond_met | track_cond_met_event;

  mode_armed  := (mode=armed_initial) | (mode=armed_long_enough);
  mode_active := (mode_armed)         | (mode=track);

  activated   := (mode=cleared) & (activate | switch);
  deactivated := mode_active    & (clear    | switch);

-- ----------------------------------------------------------------------------------------------

MODULE lateral_guidance(_hdg_switch_hit,      _ga_switch_hit,  _ap_engaged_event,
                        _sync_switch_pressed, _nav_switch_hit, _nav_armed_long_enough_event,
                        _nav_track_cond_met_event,      clear, select_default,      fd_is_on)

VAR
  roll : simple_guidance(roll_activate,      roll_deactivate,
                         roll_switch,        roll_clear           , fd_is_on);
  hdg  : simple_guidance(hdg_activate,       hdg_deactivate,
                         hdg_switch,         hdg_clear            , fd_is_on);
  ga   : simple_guidance(ga_activate,        ga_deactivate,
                         ga_switch,          ga_clear             , fd_is_on);
  nav  : arming_guidance(nav_activate,       nav_deactivate,
                         nav_switch,         nav_clear,
                         nav_track_cond_met, nav_armed_long_enough, fd_is_on);

DEFINE
  roll_activate         := (hdg_event & hdg.deactivated) | (nav_event & nav.deactivated) |
                           (lga_event & ga.deactivated ) | select_default;
  roll_deactivate       := (hdg_event & hdg.activated) | (nav_event & nav.activated) |
                           (lga_event & ga.activated );
  roll_switch           := 0;
  roll_clear            := (clear & !select_default);

  hdg_activate          := 0;
  hdg_deactivate        := (nav_event & nav.activated) | (lga_event & ga.activated );
  hdg_switch            := _hdg_switch_hit;
  hdg_clear             := clear;

  ga_activate           := _ga_switch_hit;
  ga_deactivate         := (hdg_event & hdg.activated) | (nav_event & nav.activated);
  ga_switch             := 0;
  ga_clear              := _ap_engaged_event | _sync_switch_pressed | clear;

  nav_activate          := 0;
  nav_deactivate        := (hdg_event & hdg.activated) | (lga_event & ga.activated );
  nav_switch            := _nav_switch_hit;
  nav_clear             := hdg.activated | ga.activated | clear;
  nav_track_cond_met    := _nav_track_cond_met_event;
  nav_armed_long_enough := _nav_armed_long_enough_event;

  hdg_event             := _hdg_switch_hit;
  lga_event             := _ga_switch_hit | _ap_engaged_event | _sync_switch_pressed;
  nav_event             := _nav_switch_hit | _nav_armed_long_enough_event |
                             _nav_track_cond_met_event;

-- ---------------------------------------------------------------------------------------

MODULE vertical_guidance(_vs_pitch_wheel_changed, _vs_switch_hit, _ga_switch_hit,
                         _ap_engaged_event,       _sync_switch_pressed,
                         clear,                   select_default,       fd_is_on)

VAR
  pitch : simple_guidance(pitch_activate, pitch_deactivate,
                          pitch_switch,   pitch_clear,      fd_is_on);
  vs    : simple_guidance(vs_activate,    vs_deactivate,
                          vs_switch,      vs_clear,         fd_is_on);
  ga    : simple_guidance(ga_activate,    ga_deactivate,
                          ga_switch,      ga_clear,         fd_is_on);

DEFINE
  pitch_activate   := (vs_event & vs.deactivated) | (vga_event & ga.deactivated) |
                      _vs_pitch_wheel_changed        |
                      select_default;
  pitch_deactivate := (vs_event  & vs.activated) | (vga_event & ga.activated);
  pitch_switch     := 0;
  pitch_clear      := (clear & !select_default);

  vs_activate      := 0;
  vs_deactivate    := (pitch_event & pitch.activated) | (vga_event & ga.activated   );
  vs_switch        := _vs_switch_hit;
  vs_clear         := clear;

  ga_activate      := 0;
  ga_deactivate    := (pitch_event & pitch.activated) | (vs_event & vs.activated   );
  ga_switch        := _ga_switch_hit;
  ga_clear         := _ap_engaged_event | _sync_switch_pressed | clear;

  pitch_event      := _vs_pitch_wheel_changed;
  vs_event         := _vs_switch_hit;
  vga_event        := _ga_switch_hit | _ap_engaged_event | _sync_switch_pressed;

-- ---------------------------------------------------------------------------------------
```

```
MODULE flight_director(force_cues, turn_on, switch, turn_off,
                       _ap_engaged, _overspeed)

VAR
  mode : {off, cues, no_cues};

ASSIGN
  init(mode) := off;
  next(mode) := case
                  turned_off                                              : off;
                  turned_on                                               : cues;
                  (mode=no_cues) &
                    (force_cues | turn_on | (switch & (_overspeed | _ap_engaged))) : cues;
                  (mode=cues) & switch & (_overspeed | _ap_engaged)             : no_cues;
                  1                                                             : mode;
                esac;

DEFINE
  mode_on    := (mode=cues) | (mode=no_cues);
  turned_on  := (mode=off) & (turn_on | force_cues | switch);
  turned_off := (mode_on)  & (switch  | turn_off) & !_overspeed & !_ap_engaged;

-- ---------------------------------------------------------------------------------------------

MODULE main

VAR
  env_ev     : {hdg_switch_hit,              nav_switch_hit,
                nav_armed_long_enough_event, nav_track_cond_met_event,
                ga_switch_hit,               vs_pitch_wheel_changed,
                vs_switch_hit,               fd_switch_hit,
                overspeed_start,             overspeed_end,
                ap_engaged_event,            ap_disengaged_event,
                sync_switch_pressed,         sync_switch_released     };

  overspeed  : boolean;
  ap_engaged : boolean;

  fd         : flight_director(fd_force_cues, fd_turn_on,    fd_switch,
                               fd_turn_off,   fd_ap_engaged, fd_overspeed);

  lateral    : lateral_guidance(lg_hdg_switch_hit, lg_ga_switch_hit,
                                lg_ap_engaged,     lg_sync_switch_pressed,
                                lg_nav_switch_hit, lg_nav_armed_long_enough,
                                lg_nav_track_cond_met,
                                lg_clear,          lg_select_default,
                                fd_is_on);
  vertical   : vertical_guidance(vg_vs_pitch_wheel_changed, vg_vs_switch_hit,
                                 vg_ga_switch_hit,          vg_ap_engaged,
                                 vg_sync_switch_pressed,
                                 vg_clear,                  vg_select_default,
                                 fd_is_on);

ASSIGN
  init(env_ev)     := all_events;
  next(env_ev)     := all_events;

  init(overspeed)  := 0;
  next(overspeed)  := new_overspeed;

  init(ap_engaged) := 0;
  next(ap_engaged) := new_ap_engaged;

DEFINE
  all_events :=  {hdg_switch_hit,              nav_switch_hit,
                  nav_armed_long_enough_event, nav_track_cond_met_event,
                  ga_switch_hit,               vs_pitch_wheel_changed,
                  vs_switch_hit,               fd_switch_hit,
                  overspeed_start,             overspeed_end,
                  ap_engaged_event,            ap_disengaged_event,
                  sync_switch_pressed,         sync_switch_released     };

  new_overspeed              := case
                                  (env_ev=overspeed_start)     : 1;
                                  (env_ev=overspeed_end)       : 0;
                                  1                            : overspeed;
                                esac;

  new_ap_engaged             := case
                                  (env_ev=ap_engaged_event)    : 1;
                                  (env_ev=ap_disengaged_event) : 0;
                                  1                            : ap_engaged;
                                esac;

  fd_is_on                   := !(fd.mode=off) | fd.turned_on;

  lateral_mode_requested     := (env_ev=hdg_switch_hit) | (env_ev=nav_switch_hit) |
                                (env_ev=ga_switch_hit);
  vertical_mode_requested    := (env_ev=vs_switch_hit)  | (env_ev=ga_switch_hit);

  flight_director_event      := (env_ev=ap_engaged_event) | (env_ev=fd_switch_hit) |
                                (env_ev=overspeed_start)  |
                                lateral_mode_requested    | vertical_mode_requested;
  fd_force_cues               := (env_ev=overspeed_start);
  fd_turn_on                  := (env_ev=ap_engaged_event) |
                                 lateral_mode_requested    | vertical_mode_requested;
  fd_switch                   := (env_ev=fd_switch_hit);
  fd_turn_off                 := 0;
  fd_ap_engaged               := new_ap_engaged;
  fd_overspeed                := new_overspeed;

  lg_hdg_switch_hit           := (env_ev=hdg_switch_hit);
  lg_ga_switch_hit            := (env_ev=ga_switch_hit);
  lg_ap_engaged               := (env_ev=ap_engaged_event);
  lg_sync_switch_pressed      := (env_ev=sync_switch_pressed);
  lg_nav_switch_hit           := (env_ev=nav_switch_hit);
  lg_nav_armed_long_enough    := (env_ev=nav_armed_long_enough_event);
  lg_nav_track_cond_met       := (env_ev=nav_track_cond_met_event);
  lg_clear                    := flight_director_event & fd.turned_off;
  lg_select_default           := flight_director_event & fd.turned_on;

  vg_vs_pitch_wheel_changed   := (env_ev=vs_pitch_wheel_changed);
  vg_vs_switch_hit            := (env_ev=vs_switch_hit);
  vg_ga_switch_hit            := (env_ev=ga_switch_hit);
  vg_ap_engaged               := (env_ev=ap_engaged_event);
  vg_sync_switch_pressed      := (env_ev=sync_switch_pressed);
  vg_clear                    := flight_director_event & fd.turned_off;
  vg_select_default           := flight_director_event & fd.turned_on;

-- mandatory properties ----------------------------------------------------------------------


-- the flight director is on if the autopilot is engaged

DEFINE fd_on_if_ap_engaged := AG (ap_engaged -> !(fd.mode=off));

SPEC fd_on_if_ap_engaged



-- at least one lateral mode is active iff the flight director is on

DEFINE at_least_one_lateral_mode_active :=
  AG (!(fd.mode=off) <-> (lateral.roll.mode=active | lateral.hdg.mode=active |
                          lateral.ga.mode=active   | lateral.nav.mode_active )
     );

SPEC at_least_one_lateral_mode_active
```

```
-- there is never more than one lateral mode active

DEFINE at_most_one_lateral_mode_active :=
         AG((lateral.ga.mode=active ->  (lateral.roll.mode=cleared & lateral.hdg.mode=cleared  &
                                         lateral.nav.mode=cleared  )
            ) &
            (lateral.roll.mode=active -> (lateral.ga.mode=cleared  & lateral.hdg.mode=cleared &
                                          lateral.nav.mode=cleared )
            ) &
            (lateral.hdg.mode=active -> (lateral.roll.mode=cleared & lateral.nav.mode=cleared  &
                                         lateral.ga.mode=cleared   )
            ) &
            (lateral.nav.mode_active -> (lateral.roll.mode=cleared & lateral.hdg.mode=cleared  &
                                         lateral.ga.mode=cleared   )
            )
           );

SPEC at_most_one_lateral_mode_active

-- at least one vertical mode is active iff the flight director is on

DEFINE at_least_one_vertical_mode_active :=
         AG (!(fd.mode=off) <-> (vertical.ga.mode=active    | vertical.vs.mode=active |
                                 vertical.pitch.mode=active )
            );

SPEC at_least_one_vertical_mode_active

-- at most one vertical mode is active

DEFINE at_most_one_vertical_mode_active :=
  AG ((vertical.ga.mode=active -> (vertical.pitch.mode=cleared & vertical.vs.mode=cleared)) &
      (vertical.vs.mode=active -> (vertical.pitch.mode=cleared & vertical.ga.mode=cleared)) &
      (vertical.pitch.mode=active -> (vertical.ga.mode=cleared & vertical.vs.mode=cleared))
     );

SPEC at_most_one_vertical_mode_active

-- if the flight director is off, all modes must be cleared

DEFINE fd_off_implies_all_modes_cleared :=
  AG (fd.mode=off -> (vertical.pitch.mode=cleared & vertical.vs.mode   =cleared &
                      vertical.ga.mode   =cleared & lateral.roll.mode  =cleared &
                      lateral.hdg.mode   =cleared & lateral.ga.mode    =cleared &
                      lateral.nav.mode   =cleared
                     )
     );
```

```
SPEC fd_off_implies_all_modes_cleared

-- the default modes are active if the flight director is on and all other modes are cleared

DEFINE default_modes :=
  AG ((!(fd.mode=off) & vertical.vs.mode =cleared & vertical.ga.mode =cleared &
                        lateral.hdg.mode =cleared & lateral.ga.mode  =cleared &
                        lateral.nav.mode =cleared
       ) ->
       (vertical.pitch.mode=active & lateral.roll.mode=active)
      );

SPEC default_modes

-- mode confusion properties -----------------------------------------------------------------

-- check for response to pressing HDG button

DEFINE
  hdg_selected_and_hdg_toggle_1 :=
    AG (lateral.hdg.mode=cleared & env_ev=hdg_switch_hit -> AX lateral.hdg.mode=active);

  hdg_deselected :=
    AG (lateral.hdg.mode=active  & env_ev=hdg_switch_hit -> AX lateral.roll.mode=active);

  hdg_toggle_2 :=
    AG (lateral.hdg.mode=active  & env_ev=hdg_switch_hit -> AX lateral.hdg.mode=cleared);

SPEC hdg_selected_and_hdg_toggle_1
SPEC hdg_deselected
SPEC hdg_toggle_2

-- check for response to pressing NAV button

DEFINE
  nav_selected_and_nav_toggle_1 :=
    AG (lateral.nav.mode=cleared & env_ev=nav_switch_hit -> AX lateral.nav.mode_active);

  nav_deselected :=
    AG (lateral.nav.mode_active  & env_ev=nav_switch_hit -> AX lateral.roll.mode=active);

  nav_toggle_2 :=
    AG (lateral.nav.mode=active  & env_ev=nav_switch_hit -> AX lateral.nav.mode=cleared);

SPEC nav_selected_and_nav_toggle_1
SPEC nav_deselected
SPEC nav_toggle_2
```

```
-- check for response to pressing VS button

DEFINE
  vs_selected_and_vs_toggle_1 :=
    AG (vertical.vs.mode=cleared & env_ev=vs_switch_hit -> AX vertical.vs.mode=active);

  vs_deselected :=
    AG (vertical.vs.mode=active  & env_ev=vs_switch_hit -> AX vertical.pitch.mode=active);

  vs_toggle_2 :=
    AG (vertical.vs.mode=active  & env_ev=vs_switch_hit -> AX vertical.vs.mode=cleared);

SPEC vs_selected_and_vs_toggle_1
SPEC vs_deselected
SPEC vs_toggle_2

-- check for response to pressing the FD button

DEFINE
  fd_off     := AG (fd.mode=off     & env_ev=fd_switch_hit -> AX fd.mode=cues);
  fd_on      := AG (!(fd.mode=off)  & env_ev=fd_switch_hit &
                     !(ap_engaged | overspeed) ->
                    AX fd.mode=off
                    );
  fd_cues    := AG (fd.mode=cues    & env_ev=fd_switch_hit &
                      (ap_engaged | overspeed) ->
                    AX fd.mode=no_cues
                    );
  fd_no_cues := AG (fd.mode=no_cues & env_ev=fd_switch_hit &
                      (ap_engaged | overspeed) ->
                    AX fd.mode=cues
                    );

SPEC fd_off
SPEC fd_on
SPEC fd_cues
SPEC fd_no_cues

-- search for ignored crew inputs

DEFINE
  crew_input  := env_ev=ap_engaged_event       | env_ev=fd_switch_hit        |
                 env_ev=sync_switch_pressed    | env_ev=sync_switch_released |
                 lateral_mode_requested        | vertical_mode_requested |
                 env_ev=vs_pitch_wheel_changed;
  mode_change :=
    !(fd.mode=off      <-> AX fd.mode=off)      |
    !(fd.mode=cues     <-> AX fd.mode=cues)     |
    !(fd.mode=no_cues <-> AX fd.mode=no_cues) |

    !(lateral.roll.mode = cleared <-> AX lateral.roll.mode = cleared) |
    !(lateral.roll.mode = active  <-> AX lateral.roll.mode = active ) |
    !(lateral.hdg.mode  = cleared <-> AX lateral.hdg.mode  = cleared) |
    !(lateral.hdg.mode  = active  <-> AX lateral.hdg.mode  = active ) |
    !(lateral.ga.mode   = cleared <-> AX lateral.ga.mode   = cleared) |
    !(lateral.ga.mode   = active  <-> AX lateral.ga.mode   = active ) |
    !(lateral.nav.mode  = cleared <-> AX lateral.nav.mode  = cleared) |
    !(lateral.nav.mode  = track   <-> AX lateral.nav.mode  = track  ) |
    !(lateral.nav.mode  = armed_initial <->
        AX lateral.nav.mode = armed_initial)                          |
    !(lateral.nav.mode  = armed_long_enough   <->
        AX lateral.nav.mode = armed_long_enough)                      |

    !(vertical.pitch.mode = cleared <-> AX vertical.pitch.mode = cleared) |
    !(vertical.pitch.mode = active  <-> AX vertical.pitch.mode = active ) |
    !(vertical.vs.mode    = cleared <-> AX vertical.vs.mode    = cleared) |
    !(vertical.vs.mode    = active  <-> AX vertical.vs.mode    = active ) |
    !(vertical.ga.mode    = cleared <-> AX vertical.ga.mode    = cleared) |
    !(vertical.ga.mode    = active  <-> AX vertical.ga.mode    = active );

  search_for_ignored_crew_inputs := AG (crew_input -> mode_change);

SPEC search_for_ignored_crew_inputs                        -- property violated

-- no unknown ignored crew inputs

DEFINE
  ignored_crew_input :=
    (env_ev=ap_engaged_event       & !(lateral.ga.mode=active | vertical.ga.mode=active)) |
--  (ev=ga_switch_hit & (lateral.ga.mode=active & vertical.ga.mode=active)) |
--                                                PVS model too strong (may be left out)
    (env_ev=sync_switch_pressed    & !(lateral.ga.mode=active | vertical.ga.mode=active)) |
--  (ev=sync_switch_pressed & fd.mode=off) |
--                                                PVS model too strong (may be left out)
    (env_ev=sync_switch_released)                 |
    (env_ev=vs_pitch_wheel_changed & fd.mode=off) |
    (env_ev=vs_pitch_wheel_changed & vertical.pitch.mode=active);

  no_known_ignored_crew_inputs :=
    AG (crew_input & !ignored_crew_input -> mode_change);

SPEC no_known_ignored_crew_inputs
```

```
-- search for indirect mode changes

DEFINE
  search_for_indirect_mode_changes := AG (!crew_input -> !mode_change);

SPEC search_for_indirect_mode_changes                     -- property violated

-- no unknown indirect mode changes

DEFINE
  indirect_mode_change :=
    (env_ev=overspeed_start & !(fd.mode=cues))                             |
    (env_ev=nav_armed_long_enough_event & lateral.nav.mode=armed_initial)  |
    (env_ev=nav_track_cond_met_event & lateral.nav.mode=armed_long_enough);

  no_unknown_indirect_mode_change :=
    AG ((!crew_input & !indirect_mode_change) -> !mode_change);

SPEC no_unknown_indirect_mode_change

-- ---------------------------------------------------------------------------------------------
```

B.2. Output of the SMV verifier.

```
-- specification fd_on_if_ap_engaged is true
-- specification at_least_one_lateral_mode_active is true
-- specification at_most_one_lateral_mode_active is true
-- specification at_least_one_vertical_mode_active is true
-- specification at_most_one_vertical_mode_active is true
-- specification fd_off_implies_all_modes_cleared is true
-- specification default_modes is true
-- specification hdg_selected_and_hdg_toggle_1 is true
-- specification hdg_deselected is true
-- specification hdg_toggle_2 is true
-- specification nav_selected_and_nav_toggle_1 is true
-- specification nav_deselected is true
-- specification nav_toggle_2 is true
-- specification vs_selected_and_vs_toggle_1 is true
-- specification vs_deselected is true
-- specification vs_toggle_2 is true
-- specification fd_off is true
-- specification fd_on is true
-- specification fd_cues is true
-- specification fd_no_cues is true
-- specification search_for_ignored_crew_inputs is false
-- as demonstrated by the following execution sequence
-- loop starts here --
```

```
state 1.1:
vg_select_default = 0
vg_clear = 0
vg_sync_switch_pressed = 0
vg_ap_engaged = 0
vg_ga_switch_hit = 0
vg_vs_switch_hit = 0
vg_vs_pitch_wheel_changed = 0
lg_select_default = 0
lg_clear = 0
lg_nav_track_cond_met = 0
lg_nav_armed_long_enough = 0
lg_nav_switch_hit = 0
lg_sync_switch_pressed = 0
lg_ap_engaged = 0
lg_ga_switch_hit = 0
lg_hdg_switch_hit = 0
fd_overspeed = 0
fd_ap_engaged = 0
fd_turn_off = 0
fd_switch = 0
fd_turn_on = 0
fd_force_cues = 0
flight_director_event = 0
vertical_mode_requested = 0
lateral_mode_requested = 0
fd_is_on = 0
new_ap_engaged = 0
new_overspeed = 0
all_events = hdg_switch_hit,nav_switch_hit,nav_armed_...
fd_on_if_ap_engaged = 1
at_least_one_lateral_mode_active = 1
at_most_one_lateral_mode_active = 1
at_least_one_vertical_mode_active = 1
at_most_one_vertical_mode_active = 1
fd_off_implies_all_modes_cleared = 1
default_modes = 1
hdg_toggle_2 = 1
hdg_deselected = 1
hdg_selected_and_hdg_toggle_1 = 1
nav_toggle_2 = 1
nav_deselected = 1
nav_selected_and_nav_toggle_1 = 1
vs_toggle_2 = 1
vs_deselected = 1
vs_selected_and_vs_toggle_1 = 1
fd_no_cues = 1
```

```
fd_cues = 1
fd_on = 1
fd_off = 1
search_for_ignored_crew_inputs = 0
mode_change = 0
crew_input = 1
no_known_ignored_crew_inputs = 1
ignored_crew_input = 1
search_for_indirect_mode_changes = 0
no_unknown_indirect_mode_change = 1
indirect_mode_change = 0
env_ev = sync_switch_released
overspeed = 0
ap_engaged = 0
fd.turned_off = 0
fd.turned_on = 0
fd.mode_on = 0
fd.mode = off
lateral.nav_event = 0
lateral.lga_event = 0
lateral.hdg_event = 0
lateral.nav_armed_long_enough = 0
lateral.nav_track_cond_met = 0
lateral.nav_clear = 0
lateral.nav_switch = 0
lateral.nav_deactivate = 0
lateral.nav_activate = 0
lateral.ga_clear = 0
lateral.ga_switch = 0
lateral.ga_deactivate = 0
lateral.ga_activate = 0
lateral.hdg_clear = 0
lateral.hdg_switch = 0
lateral.hdg_deactivate = 0
lateral.hdg_activate = 0
lateral.roll_clear = 0
lateral.roll_switch = 0
lateral.roll_deactivate = 0
lateral.roll_activate = 0
lateral.roll.deactivated = 0
lateral.roll.activated = 0
lateral.roll.mode = cleared
lateral.hdg.deactivated = 0
lateral.hdg.activated = 0
lateral.hdg.mode = cleared
lateral.ga.deactivated = 0
lateral.ga.activated = 0
lateral.ga.mode = cleared
lateral.nav.deactivated = 0
lateral.nav.activated = 0
lateral.nav.mode_active = 0
lateral.nav.mode_armed = 0
lateral.nav.track_cond = 0
lateral.nav.mode = cleared
lateral.nav.track_cond_met = 0
vertical.vga_event = 0
vertical.vs_event = 0
vertical.pitch_event = 0
vertical.ga_clear = 0
vertical.ga_switch = 0
vertical.ga_deactivate = 0
vertical.ga_activate = 0
vertical.vs_clear = 0
vertical.vs_switch = 0
vertical.vs_deactivate = 0
vertical.vs_activate = 0
vertical.pitch_clear = 0
vertical.pitch_switch = 0
vertical.pitch_deactivate = 0
vertical.pitch_activate = 0
vertical.pitch.deactivated = 0
vertical.pitch.activated = 0
vertical.pitch.mode = cleared
vertical.vs.deactivated = 0
vertical.vs.activated = 0
vertical.vs.mode = cleared
vertical.ga.deactivated = 0
vertical.ga.activated = 0
vertical.ga.mode = cleared

state 1.2:

-- specification no_known_ignored_crew_inputs is true
-- specification search_for_indirect_mode_changes is false
-- as demonstrated by the following execution sequence
state 2.1:
vg_select_default = 0
vg_clear = 0
vg_sync_switch_pressed = 0
vg_ap_engaged = 0
vg_ga_switch_hit = 0
vg_vs_switch_hit = 0
vg_vs_pitch_wheel_changed = 0
lg_select_default = 0
```

```
lg_clear = 0
lg_nav_track_cond_met = 0
lg_nav_armed_long_enough = 0
lg_nav_switch_hit = 0
lg_sync_switch_pressed = 0
lg_ap_engaged = 0
lg_ga_switch_hit = 0
lg_hdg_switch_hit = 0
fd_overspeed = 0
fd_ap_engaged = 0
fd_turn_off = 0
fd_switch = 0
fd_turn_on = 0
fd_force_cues = 0
flight_director_event = 0
vertical_mode_requested = 0
lateral_mode_requested = 0
fd_is_on = 0
new_ap_engaged = 0
new_overspeed = 0
all_events = hdg_switch_hit,nav_switch_hit,nav_armed_...
fd_on_if_ap_engaged = 1
at_least_one_lateral_mode_active = 1
at_most_one_lateral_mode_active = 1
at_least_one_vertical_mode_active = 1
at_most_one_vertical_mode_active = 1
fd_off_implies_all_modes_cleared = 1
default_modes = 1
hdg_toggle_2 = 1
hdg_deselected = 1
hdg_selected_and_hdg_toggle_1 = 1
nav_toggle_2 = 1
nav_deselected = 1
nav_selected_and_nav_toggle_1 = 1
vs_toggle_2 = 1
vs_deselected = 1
vs_selected_and_vs_toggle_1 = 1
fd_no_cues = 1
fd_cues = 1
fd_on = 1
fd_off = 1
search_for_ignored_crew_inputs = 0
mode_change = 0
crew_input = 1
no_known_ignored_crew_inputs = 1
ignored_crew_input = 1
search_for_indirect_mode_changes = 0
```

```
no_unknown_indirect_mode_change = 1
indirect_mode_change = 0
env_ev = sync_switch_released
overspeed = 0
ap_engaged = 0
fd.turned_off = 0
fd.turned_on = 0
fd.mode_on = 0
fd.mode = off
lateral.nav_event = 0
lateral.lga_event = 0
lateral.hdg_event = 0
lateral.nav_armed_long_enough = 0
lateral.nav_track_cond_met = 0
lateral.nav_clear = 0
lateral.nav_switch = 0
lateral.nav_deactivate = 0
lateral.nav_activate = 0
lateral.ga_clear = 0
lateral.ga_switch = 0
lateral.ga_deactivate = 0
lateral.ga_activate = 0
lateral.hdg_clear = 0
lateral.hdg_switch = 0
lateral.hdg_deactivate = 0
lateral.hdg_activate = 0
lateral.roll_clear = 0
lateral.roll_switch = 0
lateral.roll_deactivate = 0
lateral.roll_activate = 0
lateral.roll.deactivated = 0
lateral.roll.activated = 0
lateral.roll.mode = cleared
lateral.hdg.deactivated = 0
lateral.hdg.activated = 0
lateral.hdg.mode = cleared
lateral.ga.deactivated = 0
lateral.ga.activated = 0
lateral.ga.mode = cleared
lateral.nav.deactivated = 0
lateral.nav.activated = 0
lateral.nav.mode_active = 0
lateral.nav.mode_armed = 0
lateral.nav.track_cond = 0
lateral.nav.mode = cleared
lateral.nav.track_cond_met = 0
vertical.vga_event = 0
vertical.vs_event = 0
vertical.pitch_event = 0
vertical.ga_clear = 0
vertical.ga_switch = 0
vertical.ga_deactivate = 0
vertical.ga_activate = 0
vertical.vs_clear = 0
vertical.vs_switch = 0
vertical.vs_deactivate = 0
vertical.vs_activate = 0
vertical.pitch_clear = 0
vertical.pitch_switch = 0
vertical.pitch_deactivate = 0
vertical.pitch_activate = 0
vertical.pitch.deactivated = 0
vertical.pitch.activated = 0
vertical.pitch.mode = cleared
vertical.vs.deactivated = 0
vertical.vs.activated = 0
vertical.vs.mode = cleared
vertical.ga.deactivated = 0
vertical.ga.activated = 0
vertical.ga.mode = cleared

state 2.2:
vg_select_default = 1
lg_select_default = 1
fd_overspeed = 1
fd_force_cues = 1
flight_director_event = 1
fd_is_on = 1
new_overspeed = 1
mode_change = 1
crew_input = 0
ignored_crew_input = 0
indirect_mode_change = 1
env_ev = overspeed_start
fd.turned_on = 1
lateral.roll_activate = 1
lateral.roll.activated = 1
vertical.pitch_activate = 1
vertical.pitch.activated = 1

state 2.3:
vg_select_default = 0
lg_select_default = 0
fd_force_cues = 0
```

```
flight_director_event = 0
mode_change = 0
crew_input = 1
ignored_crew_input = 1
indirect_mode_change = 0
env_ev = sync_switch_released
overspeed = 1
fd.turned_on = 0
fd.mode_on = 1
fd.mode = cues
lateral.roll_activate = 0
lateral.roll.activated = 0
lateral.roll.mode = active
vertical.pitch_activate = 0
vertical.pitch.activated = 0
vertical.pitch.mode = active

-- specification no_unknown_indirect_mode_change is true

resources used:
user time: 0.58 s, system time: 0.17 s
BDD nodes allocated: 5940
Bytes allocated: 983040
BDD nodes representing transition relation: 438 + 1
reachable states: 3388 (2^11.7262) out of 86016 (2^16.3923)
```

Appendix C. Specification and Verification Using **Spin**.

C.1. Full Model of the Mode Logic.

```
/** values of modes ***********************************************************************/

#define cleared            0
#define active             1
#define track              2
#define armed_initial      3
#define armed_long_enough  4
#define off                5
#define cues               6
#define no_cues            7

/** events and signals ********************************************************************/

mtype = { /* environment events */
          hdg_switch_hit,              nav_switch_hit,
          nav_armed_long_enough_event, nav_track_cond_met_event,
          ga_switch_hit,               vs_pitch_wheel_changed,
          vs_switch_hit,               fd_switch_hit,
          overspeed_start,             overspeed_stop,
          ap_engaged_event,            ap_disengaged_event,
          sync_switch_pressed,         sync_switch_released,

          /* simple guidance, arming guidance and flight director events */
          clear,                       activate,
          deactivate,                  switch,
          turn_on,                     turn_off,
          force_cues,                  armed_long_enough_event,
          track_cond_met_event,

          /* signals */
          activated,                   deactivated,
          turned_on,                   turned_off,
          null
        }

typedef ag_state { byte mode           = cleared;
                   bool track_cond_met = false
                 };

/** variables controled by the environment ************************************************/

bool overspeed      = false;
bool ap_engaged     = false;
```

```
/** mode and signal variables & variable for env. event ************************************/

bit  pitch = cleared;       bit  old_pitch = cleared;
bit  vs    = cleared;       bit  old_vs    = cleared;
bit  vga   = cleared;       bit  old_vga   = cleared;
bit  roll  = cleared;       bit  old_roll  = cleared;
bit  hdg   = cleared;       bit  old_hdg   = cleared;
bit  lga   = cleared;       bit  old_lga   = cleared;
byte fd    = off;           byte old_fd    = off;

ag_state nav;               ag_state old_nav;

mtype pitch_signal = null;  mtype vs_signal   = null;
mtype vga_signal   = null;  mtype roll_signal = null;
mtype hdg_signal   = null;  mtype lga_signal  = null;
mtype nav_signal   = null;  mtype fd_signal   = null;

mtype env_ev = null;

/** useful abbreviations ********************************************************************/

#define lateral_mode_requested
        ((env_ev==hdg_switch_hit) || (env_ev==nav_switch_hit) || (env_ev==ga_switch_hit))

#define vertical_mode_requested
        ((env_ev==vs_switch_hit) || (env_ev==ga_switch_hit))

inline flight_director_event(env_ev) { (env_ev==ap_engaged_event) ||
                                       (env_ev==fd_switch_hit)    ||
                                       (env_ev==overspeed_start)  ||
                                        lateral_mode_requested    ||
                                       vertical_mode_requested
                                     }

/** auxiliary "functions" *****************************************************************/

inline hdg_event(env_ev)   { env_ev==hdg_switch_hit }

inline nav_event(env_ev)   { (env_ev==nav_switch_hit)              ||
                             (env_ev==nav_armed_long_enough_event) ||
                             (env_ev==nav_track_cond_met_event)
                           }

inline lga_event(env_ev)   { (env_ev==ga_switch_hit)       ||
                             (env_ev==ap_engaged_event)    ||
                             (env_ev==sync_switch_pressed)
                           }
```

```
inline pitch_event(env_ev) { env_ev==vs_pitch_wheel_changed }

inline vs_event(env_ev)    { env_ev==vs_switch_hit }

inline vga_event(env_ev)   { (env_ev==ga_switch_hit)       || (env_ev==ap_engaged_event) ||
                             (env_ev==sync_switch_pressed)
                           }

/** abstract data type module simple guidance *********************************************/

inline simple_guidance(mode, event, signal)
{
  if
  :: mode==cleared ->
       if
       :: event==activate   -> mode=active;  signal=activated
       :: event==deactivate ->               signal=null
       :: event==switch     -> mode=active;  signal=activated
       :: event==clear      ->               signal=null
       fi
  :: mode==active  ->
       if
       :: event==activate   ->               signal=null
       :: event==deactivate -> mode=cleared; signal=null
       :: event==switch     -> mode=cleared; signal=deactivated
       :: event==clear      -> mode=cleared; signal=deactivated
       fi
  fi
}

/** abstract data object module arming guidance ********************************************/

inline arming_guidance(event, signal)
{
  if
  :: nav.mode==cleared ->
       if
       :: event==activate                -> nav.mode          =armed_initial;
                                            signal            =activated
       :: event==deactivate              -> signal            =null
       :: event==switch                  -> nav.mode          =armed_initial;
                                            signal            =activated
       :: event==clear                   -> signal            =null
       :: event==armed_long_enough_event -> signal            =null
       :: event==track_cond_met_event    -> nav.track_cond_met=true;
                                            signal            =null
       fi
  :: else         ->
       if
       :: event==activate                -> signal  =null
       :: event==deactivate              -> nav.mode=cleared;
                                            signal  =null
       :: event==switch                  -> nav.mode=cleared;
                                            signal  =deactivated
       :: event==clear                   -> nav.mode=cleared;
                                            signal  =deactivated
       :: event==armed_long_enough_event ->
            if
            :: (nav.mode==armed_initial) &&
               nav.track_cond_met        -> nav.mode=track;
                                            signal  =null
            :: (nav.mode==armed_initial) &&
               !nav.track_cond_met       -> nav.mode=armed_long_enough;
                                            signal  =null
            :: else                      -> signal  =null
            fi
       :: event==track_cond_met_event    ->
            if
            :: nav.mode==armed_long_enough -> nav.mode           =track;
                                              nav.track_cond_met=true;
                                              signal            =null
            :: else                        -> nav.track_cond_met=true;
                                              signal            =null
            fi
       fi
  fi
}


/** function module lateral guidance *******************************************************/

inline lateral_guidance(env_ev)
{
  if
  :: hdg_event(env_ev) ->
       simple_guidance(hdg, switch, hdg_signal);
       if
       :: hdg_signal==activated   -> simple_guidance(roll, deactivate, roll_signal);
                                     simple_guidance(lga,  deactivate, lga_signal );
                                     arming_guidance(      deactivate, nav_signal )
       :: hdg_signal==deactivated -> simple_guidance(roll, activate,   roll_signal)
       :: else                    -> skip
       fi
  :: nav_event(env_ev) ->
       if
       :: env_ev==nav_switch_hit              ->
            arming_guidance(switch,                  nav_signal)
       :: env_ev==nav_track_cond_met_event    ->
            arming_guidance(track_cond_met_event,    nav_signal)
       :: env_ev==nav_armed_long_enough_event ->
            arming_guidance(armed_long_enough_event, nav_signal)
       fi;
       if
       :: nav_signal==activated   -> simple_guidance(roll, deactivate, roll_signal);
                                     simple_guidance(hdg,  deactivate, hdg_signal );
                                     simple_guidance(lga,  deactivate, lga_signal )
       :: nav_signal==deactivated -> simple_guidance(roll, activate,   roll_signal)
       :: else                    -> skip
       fi

  :: lga_event(env_ev) ->
       if
       :: env_ev==ga_switch_hit   -> simple_guidance(lga, activate, lga_signal)
       :: else                    -> simple_guidance(lga, clear,    lga_signal)
       fi;
       if
       :: lga_signal==activated   -> simple_guidance(roll, deactivate, roll_signal);
                                     simple_guidance(hdg,  deactivate, hdg_signal );
                                     arming_guidance(      deactivate, nav_signal )
       :: lga_signal==deactivated -> simple_guidance(roll,  activate,   roll_signal)
       :: else                    -> skip
       fi

  :: else              ->
       skip
  fi
}

/** function module vertical guidance ********************************************************/

inline vertical_guidance(env_ev)
{
  if
  :: pitch_event(env_ev) ->
       simple_guidance(pitch, activate, pitch_signal);
       if
       :: pitch_signal==activated -> simple_guidance(vs,  deactivate, vs_signal );
                                     simple_guidance(vga, deactivate, vga_signal)
       :: else                    -> skip
       fi
  :: vs_event(env_ev)    ->
       simple_guidance(vs, switch, vs_signal);
       if
       :: vs_signal==activated    -> simple_guidance(pitch, deactivate, pitch_signal);
                                     simple_guidance(vga,   deactivate, vga_signal  )
       :: vs_signal==deactivated  -> simple_guidance(pitch, activate,   pitch_signal)
       :: else                    -> skip
       fi

  :: vga_event(env_ev)   ->
       if
       :: env_ev==ga_switch_hit   -> simple_guidance(vga, switch,vga_signal)
       :: else                    -> simple_guidance(vga, clear, vga_signal)
       fi;
       if
       :: vga_signal==activated   -> simple_guidance(pitch, deactivate, pitch_signal);
                                     simple_guidance(vs,    deactivate, vs_signal   )
       :: vga_signal==deactivated -> simple_guidance(pitch, activate,   pitch_signal)
       :: else                    -> skip
       fi

  :: else                ->
       skip
  fi
}

/** abstract data object module flight director *******************************************/

inline flight_director(event, signal)
{
  if
  :: fd==off     ->
       if
       :: event==force_cues -> fd=cues; signal=turned_on;
       :: event==turn_on    -> fd=cues; signal=turned_on
       :: event==switch     -> fd=cues; signal=turned_on
       :: event==turn_off   ->          signal=null
       fi
  :: fd==cues    ->
       if
       :: event==force_cues -> signal=null
       :: event==turn_on    -> signal=null
       :: event==switch     ->
            if
            :: overspeed || ap_engaged -> fd=no_cues; signal=null
            :: else                    -> fd=off;     signal=turned_off
            fi
       :: event==turn_off   ->
            if
            :: overspeed || ap_engaged -> fd=no_cues; signal=null
            :: else                    -> fd=off;     signal=turned_off
            fi
       fi
  :: else        ->
       if
       :: event==force_cues -> fd=cues; signal=null
       :: event==turn_on    -> fd=cues; signal=null
       :: event==switch     ->
            if
            :: overspeed || ap_engaged -> fd=cues; signal=null
            :: else                    -> fd=off;  signal=turned_off
            fi
       :: event==turn_off   ->
            if
            :: overspeed || ap_engaged ->          signal=null
            :: else                    -> fd=off;  signal=turned_off
            fi
       fi
  fi
}

/** mandatory and mode confusion properties as assertions ***********************************/

#define nav_active
        ((nav.mode==armed_initial) || (nav.mode==armed_long_enough) || (nav.mode==track))

#define crew_input
        ((env_ev==ap_engaged_event)        || (env_ev==sync_switch_pressed) ||
         (env_ev==sync_switch_released)    || (env_ev==fd_switch_hit)       ||
         lateral_mode_requested            || vertical_mode_requested       ||
         (env_ev==vs_pitch_wheel_changed))

#define ignored_crew_input
        (((env_ev==ap_engaged_event)       && !((old_lga==active)||(old_vga==active))) ||
         ((env_ev==sync_switch_pressed)    && !((old_lga==active)||(old_vga==active))) ||
         (env_ev==sync_switch_released)                                                ||
         ((env_ev==vs_pitch_wheel_changed) && (old_fd==off))                           ||
         ((env_ev==vs_pitch_wheel_changed) && (old_pitch==active)))

#define indirect_mode_change
        (((env_ev==overspeed_start)             && !(old_fd==cues))                    ||
         ((env_ev==nav_armed_long_enough_event) && (old_nav.mode==armed_initial))      ||
         ((env_ev==nav_track_cond_met_event)    && (old_nav.mode==armed_long_enough)))
```

```
#define mode_change
        ((fd   != old_fd)   || (pitch != old_pitch) || (vs  != old_vs)  || (vga != old_vga) ||
         (roll != old_roll) || (hdg   != old_hdg)   || (lga != old_lga) ||
         (nav.mode != old_nav.mode))

inline mandatory_and_mode_confusion_properties(env_ev)
{
  /** the flight director is on if the autopilot is engaged **/
  assert(!ap_engaged || !(fd==off));

  /** at least one lateral mode is active iff the flight director is on **/
  assert(((fd==off) || (roll==active || hdg==active || lga==active || nav_active)) &&
         (!(roll==active || hdg==active || lga==active || nav_active) || !(fd==off))
        );

  /** there is never more than one lateral mode active **/
  assert((!(lga ==active) || (roll==cleared && hdg==cleared && nav.mode==cleared)) &&
         (!(roll==active) ||  (lga==cleared && hdg==cleared && nav.mode==cleared)) &&
         (!(hdg ==active) || (roll==cleared && lga==cleared && nav.mode==cleared)) &&
         (!(nav_active)   || (roll==cleared && hdg==cleared && lga     ==cleared))
        );

  /** at least one vertical mode is active iff the flight director is on **/
  assert(((fd==off) || (vga==active || vs==active || pitch==active))   &&
         (!(vga==active || vs==active || pitch==active) || !(fd==off))
        );

  /** at most one vertical mode is active **/
  assert((!(vga  ==active) || (pitch==cleared &&  vs==cleared)) &&
         (!(vs   ==active) || (pitch==cleared && vga==cleared)) &&
         (!(pitch==active) || (  vga==cleared &&  vs==cleared))
        );

  /** if the flight director is off, all modes must be cleared **/

  assert(!(fd==off) || (pitch==cleared &&  vs==cleared && vga==cleared && roll==cleared &&
                        hdg==cleared && lga==cleared && nav.mode==cleared)
        );

  /** the default modes are active if the flight director is on and **/
  /** all other modes are cleared                                   **/

  assert(!(!(fd==off) &&  vs==cleared && vga==cleared &&
                       hdg==cleared && lga==cleared && nav.mode==cleared
           ) || (pitch==active && roll==active)
        );
```

```
/** mandatory properties **/
if
:: env_ev==hdg_switch_hit ->
   /** check for response to pressing HDG button **/
   assert(!(old_hdg==cleared) || ( hdg==active ));
   assert(!(old_hdg==active ) || (roll==active ));
   assert(!(old_hdg==active ) || ( hdg==cleared))

:: env_ev==nav_switch_hit ->
   /** check for response to pressing NAV button **/
   assert(!(old_nav.mode==cleared) || ((nav.mode==armed_initial)     ||
                                        (nav.mode==armed_long_enough) || (nav.mode==track)
                                       )
         );
   assert(!((old_nav.mode==armed_initial) || (old_nav.mode==armed_long_enough) ||
            (old_nav.mode==track)
           ) || (roll==active)
         );
   assert(!((old_nav.mode==armed_initial) || (old_nav.mode==armed_long_enough) ||
            (old_nav.mode==track)
           ) || (nav.mode==cleared)
         )

:: env_ev==vs_switch_hit ->
   /** check for response to pressing VS button **/
   assert(!(old_vs==cleared) || (vs==active));
   assert(!(old_vs==active ) || (pitch==active));
   assert(!(old_vs==active ) || (vs==cleared))

:: env_ev==fd_switch_hit ->
   /** check for response to pressing the FD button **/
   assert(!(old_fd==off) || (fd==cues));
   assert((!(!(old_fd==off) && !(ap_engaged || overspeed))) || (fd==off));
   assert(!((old_fd==cues) && (ap_engaged || overspeed)) || (fd==no_cues));
   assert(!((old_fd==no_cues) && (ap_engaged || overspeed)) || (fd==cues));

:: else ->
     skip

fi;

/** search for ignored crew inputs        **/
/** assert(!(crew_input) || mode_change); **/
/** property violated                     **/

/** no unknown ignored crew inputs **/
assert(!(crew_input && !(ignored_crew_input)) || mode_change);
```

```
  /** search for indirect mode changes            **/
  /** assert(!(!(crew_input)) || !(mode_change)); **/
  /** property violated                           **/

  /** no unknown indirect mode changes **/
  assert(!(!(crew_input) && !(indirect_mode_change)) || !(mode_change));

  /** save the current mode values **/
  old_pitch = pitch;  old_vs    = vs;  old_vga   = vga;  old_roll  = roll;
  old_hdg   = hdg;    old_lga   = lga; old_fd    = fd;

  old_nav.mode = nav.mode;  old_nav.track_cond_met = nav.track_cond_met
}


/*********************************************************************************************/

inline clear_all_modes()
{
  pitch=cleared;   vs=cleared;       vga=cleared;  roll=cleared;
    hdg=cleared;  lga=cleared;  nav.mode=cleared
}

/*********************************************************************************************/

inline select_default_mode()
{
  pitch=active;  roll=active
}


/*********************************************************************************************/

inline process_external_event(env_ev)
{
  if
  :: env_ev==ap_engaged_event    -> ap_engaged=true
  :: env_ev==ap_disengaged_event -> ap_engaged=false
  :: env_ev==overspeed_start     -> overspeed =true
  :: env_ev==overspeed_stop      -> overspeed =false
  :: else                        -> skip
  fi
}

/*********************************************************************************************/
```

```
inline process_fd_event(env_ev)
{
  if
  :: flight_director_event(env_ev) ->
       if
       :: env_ev==fd_switch_hit    -> flight_director(switch, fd_signal)
       :: env_ev==overspeed_start  -> flight_director(force_cues, fd_signal)
       :: else                     -> flight_director(turn_on, fd_signal)
       fi;
       if
       :: fd_signal==turned_off    -> clear_all_modes()
       :: fd_signal==turned_on     -> select_default_mode()
       :: else                     -> skip
       fi
  :: else                          ->
       skip
  fi
}

/***********************************************************************************************/

inline process_flight_mode_event(env_ev)
{
  if
  :: !(fd==off) -> lateral_guidance(env_ev);  vertical_guidance(env_ev)
  :: else       -> skip
  fi
}

/***********************************************************************************************/

inline clear_signals()
{
  pitch_signal = null;  vs_signal  = null;  vga_signal = null;  roll_signal = null;
  hdg_signal   = null;  lga_signal = null;  nav_signal = null;  fd_signal   = null
}

/** main module performing modeling one synchronous step of the system *********************/

inline fgs(env_ev)
{
  process_external_event(env_ev);
  process_fd_event(env_ev);
  process_flight_mode_event(env_ev);
  clear_signals();                                         /** signals are no longer needed **/
  mandatory_and_mode_confusion_properties(env_ev)
}
```

```
/** init process, including model of the environment ***************************************/

init{ end_main: do :: atomic{ if /* nondeterministically choose env. event   */
                              :: env_ev=hdg_switch_hit
                              :: env_ev=nav_switch_hit
                              :: env_ev=nav_armed_long_enough_event
                              :: env_ev=nav_track_cond_met_event
                              :: env_ev=ga_switch_hit
                              :: env_ev=vs_pitch_wheel_changed
                              :: env_ev=vs_switch_hit
                              :: env_ev=fd_switch_hit
                              :: env_ev=overspeed_start
                              :: env_ev=overspeed_stop
                              :: env_ev=ap_engaged_event
                              :: env_ev=ap_disengaged_event
                              :: env_ev=sync_switch_pressed
                              :: env_ev=sync_switch_released
                              fi;
                              fgs(env_ev); /* perform synchronous step       */
                              env_ev=null  /* env. event is no longer needed */
                            }
                od }

/*********************************************************************************************/
```

C.2. Output of the Spin verifier.

```
(Spin Version 3.2.4 -- 10 January 1999)

Full statespace search for:
never-claim          - (none specified)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid endstates +

State-vector 32 byte, depth reached 4151, errors: 0
     242 states, stored
    3147 states, matched
    3389 transitions (= stored+matched)
  165976 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.604  memory usage (Mbyte)

real        1.9
user        1.7
sys         0.2
```