Analyzing Mode Confusion via Model Checking*

Gerald Lüttgen¹ and Victor Carreño²

¹ Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23681-2199, USA, luettgem@icase.edu
² Assessment Technology Branch, NASA Langley Research Center, Hampton, Virginia 23681-2199, USA, v.a.carreno@larc.nasa.gov

Abstract. Mode confusion is a serious problem in aviation safety. Today’s complex avionics systems make it difficult for pilots to maintain awareness of the actual states, or modes, of the flight deck automation. NASA explores how formal methods, especially theorem proving, can be used to discover mode confusion. The present paper investigates whether state-exploration techniques, e.g., model checking, are better able to achieve this task than theorem proving and also to compare the verification tools Murő, SMV, and Spin for the specific application. While all tools can handle the task well, their strengths are complementary.

1 Introduction

Digital system automation in the flight deck of aircraft has significantly contributed to aviation efficiency and safety. Unfortunately, the aviation community is also starting to experience some undesirable side effects as a result of the high degree of automation. Incidents and accidents in aviation are increasingly attributed to pilot-automation interaction. Although automation has reduced the overall pilot workload, in some instances the workload has just been re-distributed, causing short periods of very high workloads. In these periods, pilots sometimes get confused about the actual states, or modes, of the flight deck automation. Mode confusion may cause pilots to inappropriately interact with the on-board automation, with possibly catastrophic consequences.

NASA Langley explores 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. The mode logic of a flight guidance system (FGS) was selected as a target system to develop this approach and to determine its feasibility. The FGS offers a realistic avionics system and has been specified in many notations and languages including CoRE [12], SCR [12], Z [6], ObjeCTime [13], and PVS [2]. In the PVS [16] effort, the FGS, which is characterized by its synchronous, reactive, and deterministic behavior, was encoded as a finite state machine. Properties, which were identified

* This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the first author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia 23681-2199, USA.
as possible sources of mode confusion by experts in human factors [9], were also
defined in the PVS language. These properties included inconsistent behavior,
ignored crew inputs, and indirect mode changes. Proofs in the PVS model were
undertaken to either show that a property holds or to discover conditions that
preclude the property from being true. The employed style of theorem proving
resembled a form of state exploration. Hence, the question arises whether state-
exploration and model-checking techniques [3, 5] are better suited for the study
of mode confusion.

In order to answer this question, we model and analyze the mode logic by
applying three popular and publicly available state-exploration/model-checking
tools, namely Murϕ [4, 14], SMV [11, 17], and Spin [7, 18]. Although all three tools
are appropriate for the task, each one has its own strengths and weaknesses. We
compare the tools regarding (1) the suitability of their languages for modeling the
mode logic, (2) their suitability for specifying and verifying the mode confusion
properties of interest, and (3) their ability to generate and animate diagnostic
information. The first aspect concerns the way in which we model the example
system. The second aspect refers to the adequacy of the language in which
properties are encoded and also to the degree of orthogonality between system
and property specifications. The third aspect is perhaps the most important one
for engineers since system designs are often incorrect in early design stages.

2 Flight Guidance Systems and Mode Logics

The FGS is a component of the flight control system (cf. Fig. 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 inputted via the crew interface and/or the flight management system. In response,
the FGS 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.

![Flight control system diagram]

Fig. 1. Flight control system

In the following we focus on the mode-logic part of the FGS. Especially,
we leave out the modeling of the control laws and, if no confusion arises, use
interchangeably the terms FGS and mode logic. A mode logic essentially acts as
a deterministic machine which is composed of several synchronous sub-machines.
It receives events from its environment and reacts to them by changing its state appropriately. This reaction may require several simultaneous mode changes. Fig. 2 shows a typical mode logic consisting of three interacting components: the lateral guidance, the vertical guidance, and the flight director. The mode of the flight director determines whether the FGS is used as a navigational aid. 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 NAV mode having additional sub-states in the active state.

![Mode Logic Diagram](image)

**Fig. 2. Architecture of the model logic of the FGS**

The properties of interest regarding the FGS can be classified as mandatory properties and mode confusion properties. Some of the mandatory 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) a default mode is activated when the flight director is on and other modes are cleared. Regarding mode confusion, several categories are identified in [9]. We have selected three categories to use in the analysis of our system: (1) inconsistent behaviors, i.e., a crew interface input 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 sources of mode confusion, we formulate the negation of each property 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. This process is interactive and labor intensive when using theorem proving [13]. Thus, we investigate whether state-exploration/model-checking tools can perform the analysis more efficiently.

3 Modeling the Mode Logic in Mur\(\phi\)

*Mur\(\phi\)* [4, 14] is a state-exploration tool developed by David Dill's group at Stanford University and consists of a compiler and a description language. The compiler takes a system description and generates a C++ special-purpose verifier for it, which can then be used for checking assertions and deadlock behavior.
The Murϕ description language borrows many constructs found in imperative programming languages, such as Pascal. System descriptions may include declarations of constants, finite data-types, global and local variables, and unnested procedures and functions. Moreover, they contain transition rules for describing system behavior, a definition of the initial states, and a set of state invariants and assertions. Each transition rule may consist of a guard – which is never needed here – and an action, i.e., a statement which modifies global variables. A state in Murϕ’s execution model is an assignment to all global variables in the considered description. A transition is determined by a rule which is chosen nondeterministically from those rules whose guards are true in the current state. The rule’s execution updates some global variables according to its action.

<table>
<thead>
<tr>
<th>Table 1. Specification of module simple_guidance in Murϕ</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE sg_modes : ENUM { cleared, active };</td>
</tr>
<tr>
<td>TYPE sg_events : ENUM { activate, deactivate, switch, clear };</td>
</tr>
<tr>
<td>TYPE sg_signals : ENUM { null, activated, deactivated };</td>
</tr>
<tr>
<td>PROCEDURE simple_guidance(VAR mode : sg_modes; event : sg_events;</td>
</tr>
<tr>
<td>VAR signal : sg_signals);</td>
</tr>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>IF mode=cleared THEN</td>
</tr>
<tr>
<td>SWITCH event CASE activate : signal:=activated; mode:=active;</td>
</tr>
<tr>
<td>CASE deactivate : signal:=null;</td>
</tr>
<tr>
<td>CASE switch : signal:=activated; mode:=active;</td>
</tr>
<tr>
<td>CASE clear : signal:=null;</td>
</tr>
<tr>
<td>END;</td>
</tr>
<tr>
<td>ELSE</td>
</tr>
<tr>
<td>SWITCH event CASE activate : signal:=null;</td>
</tr>
<tr>
<td>CASE deactivate : signal:=null; mode:=cleared;</td>
</tr>
<tr>
<td>CASE switch : signal:=deactivated; mode:=cleared;</td>
</tr>
<tr>
<td>CASE clear : signal:=deactivated; mode:=cleared;</td>
</tr>
<tr>
<td>END; END;</td>
</tr>
</tbody>
</table>

The heart of the Murϕ model of the FGS is the deterministic procedure fgs which encodes the system’s reaction to some event entering the mode logic. By declaring a transition rule for each event env.ev as RULE "rule for env.ev" 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. Due to space constraints we do not completely present fgs, but concentrate on modeling the vertical-guidance component of the FGS [10]. Let us define the modes of this component as instantiations of an abstract data-type module simple_guidance which encodes each mode’s behavior as a single Mealy automaton (cf. Table 1). The module is parameterized by the mode mode under consideration, the input event event, and the output event signal. The parameters are of enumeration types sg_mode, sg_events, and sg_signals, respectively. The body of simple_guidance specifies the reaction of a mode
to \texttt{event} with respect to \texttt{mode}. This reaction is described by an \texttt{if-statement}, two \texttt{case-reitions}, and \texttt{assignments} to \texttt{mode} and \texttt{signal}. The vertical-guidance component is specified as a procedure, called \texttt{vertical\_guidance} (cf. Table 2), and employs \texttt{simple\_guidance} for describing the modes \texttt{pitch}, \texttt{vs}, and \texttt{vga}, which are defined as global variables. The task of \texttt{vertical\_guidance} is firstly to recognize whether \texttt{env\_ev} refers to mode \texttt{Pitch}, \texttt{VS}, or \texttt{VGA}. This is done by functions \texttt{pitch\_event}, \texttt{vs\_event}, and \texttt{vga\_event}. Then \texttt{env\_ev} is translated to an event of type \texttt{sg\_events} via functions \texttt{pitch\_conv}, \texttt{vs\_conv}, and \texttt{vga\_conv}, respectively, and passed to the mode to which it belongs. If this mode is activated by the event, i.e., \texttt{simple\_guidance} returns value \texttt{activated} via local variable \texttt{sig} then the other two modes must instantly be deactivated by invoking \texttt{simple\_guidance} with the appropriate modes and \texttt{event deactivate}.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
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 \\
\hspace{1em} \texttt{simple\_guidance(pitch, pitch\_conv(env\_ev), sig);} \\
\hspace{1em} IF sig=activated THEN \texttt{simple\_guidance(vs, deactivate, sig);} \\
\hspace{2em} \texttt{simple\_guidance(vga, deactivate, sig);} \\
END; \\
ELSIF vs\_event(env\_ev) THEN \\
\hspace{1em} \texttt{simple\_guidance(vs, vs\_conv(env\_ev), sig);} \\
\hspace{1em} IF sig=activated THEN \texttt{simple\_guidance(pitch, deactivate, sig);} \\
\hspace{2em} \texttt{simple\_guidance(vga, deactivate, sig);} \\
\hspace{1em} ELSIF sig=deactivated THEN \texttt{simple\_guidance(pitch, activate, sig);} \\
END; \\
ELSIF vga\_event(env\_ev) THEN \\
\hspace{1em} \texttt{simple\_guidance(vga, vga\_conv(env\_ev), sig);} \\
\hspace{1em} IF sig=activated THEN \texttt{simple\_guidance(pitch, deactivate, sig);} \\
\hspace{2em} \texttt{simple\_guidance(vs, deactivate, sig);} \\
\hspace{1em} ELSIF sig=deactivated THEN \texttt{simple\_guidance(pitch, activate, sig);} \\
END; END; END; \\
\hline
\end{tabular}
\caption{Specification of module \texttt{vertical\_guidance} in Murφ}
\end{table}

We now turn our focus to specifying mode confusion properties. As states are generated by Murφ, \texttt{assert} statements, that were explicitly included in the action of a rule, are checked. If an assertion is violated – i.e., the \texttt{assert} statement is evaluated to false in some reachable system state – the Murφ verifier halts and 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. Next, we show how an exemplary property of each category of the mode confusion properties mentioned in Section 2 can be stated as assertions. We encapsulate these assertions in procedure \texttt{mode\_confusion\_properties} which is invoked as the last
Table 3. Specification of some mode confusion properties in Murφ

<table>
<thead>
<tr>
<th>VAR</th>
<th>old_pitch, old_vs, old_vga : sg_modes;</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE</td>
<td>mode_confusion_properties(env_ev: env_events);</td>
</tr>
<tr>
<td>BEGIN</td>
<td>ALIAS mode_change : pitch!=old_pitch</td>
</tr>
<tr>
<td></td>
<td>-- check for response to pressing VS button</td>
</tr>
<tr>
<td></td>
<td>IF env_ev=vs_switch_hit THEN</td>
</tr>
<tr>
<td></td>
<td>assert (old_vs=cleared -&gt; vs=active ) &quot;vs_toggle_1&quot;;</td>
</tr>
<tr>
<td></td>
<td>assert (old_vs=active -&gt; vs=cleared) &quot;vs_toggle_2&quot;;</td>
</tr>
<tr>
<td>END;</td>
<td>-- search for ignored crew inputs (property violated)</td>
</tr>
<tr>
<td></td>
<td>assert (crew_input(env_ev) -&gt; mode_change)</td>
</tr>
<tr>
<td></td>
<td>&quot;search_for_ignored_crew_inputs&quot;;</td>
</tr>
<tr>
<td></td>
<td>-- no unknown ignored crew inputs</td>
</tr>
<tr>
<td></td>
<td>assert (crew_input(env_ev) &amp; !ignored_crew_input(ev) -&gt; mode_change)</td>
</tr>
<tr>
<td></td>
<td>&quot;no_unknown_ignored&quot;;</td>
</tr>
<tr>
<td></td>
<td>-- search for indirect mode changes (property violated)</td>
</tr>
<tr>
<td></td>
<td>assert (!crew_input(env_ev) -&gt; !mode_change)</td>
</tr>
<tr>
<td></td>
<td>&quot;search_for_indirect_mode_changes&quot;;</td>
</tr>
<tr>
<td></td>
<td>-- no unknown indirect mode changes</td>
</tr>
<tr>
<td></td>
<td>assert (!crew_input(env_ev) &amp; !indirect_mode_change(env_ev) -&gt;</td>
</tr>
<tr>
<td></td>
<td>!mode_change) &quot;no_unknown_indirect_mode_changes&quot;;</td>
</tr>
<tr>
<td>END;</td>
<td>old_pitch:=pitch; old_vs:=vs; old_vga:=vga;</td>
</tr>
</tbody>
</table>

statement in fgs (cf. Table 3). Here, “--” introduces a comment line, != denotes inequality, and !, &!, and -> stand for logical disjunction, conjunction, negation, and implication, respectively. Since all 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 can only reason about simple state invariants and not about more general “state transition invariants.” Therefore, 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, 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., if VS was in state cleared and vs_switch_hit arrived, then it is now in state active, and vice versa. Regarding the second category, we verify that no crew inputs are ignored, i.e., whenever an event that originated from the crew enters the mode logic, then at least one mode changes its value. We specify this property as crew_input(env_ev) -> mode_change, where crew_input is a function determining whether env_ev originates from the crew and where mode_change is a shortcut introduced as an ALIAS statement. As expected, this property does not always hold. The error trace returned by Murφ helps us in identifying
the cause, as is our objective. We filter out the cause by including a predicate `ignored_crew_input`, stating the negation of the cause, in the premise of the assertion (cf. Table 3). We then re-run Murφ and iterate this process until the assertion becomes true, thereby gradually capturing all crew-input scenarios responsible for mode confusion. Similarly, 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 env.ev is not originated by the crew. Using Murφ, we discover the conditions that invalidate this property and, subsequently, weaken it via predicate `indirect_mode_change`. The mandatory properties mentioned in Section 2 are formalized as invariant statements and proved. The difference between an `assert` and an `invariant` statement is that the former appears in the system description part of the model, while the latter does not. The reason for specifying mode confusion properties in the system description is their reference to `old.pitch`, `old.vs`, and `old.vga`. In order to keep the state space small, these 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 convenient for our task since the PVS model of the FGS [13] could simply be carried over. Unfortunately, Murφ forces us to encode state transition invariants as state invariants, thereby doubling the number of global variables and Murφ’s memory requirements. The full Murφ model subsumes about 30 assertions and leads to a finite automaton with 242 states and 3388 transitions. In each state, any of the 14 environment events may enter the system (“242 × 14 = 3388”). The state-space exploration took less than 2 seconds on a SUN SPARCstation 20.

4 Modeling the Mode Logic in SMV

The SMV system [11, 17], originally developed by Ken McMillan at Carnegie-Mellon University, is a model-checking tool for verifying finite-state systems against specifications in the temporal logic CTL [3, 5]. SMV implements a symbolic model-checking algorithm based on Binary Decision Diagrams (BDDs) [1].

SMV’s description language is a very simple, yet elegant language for modularly specifying finite-state systems, 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. In contrast to Murφ, SMV descriptions are interpreted. The interpreter makes sure that the specified system is 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 [5], which allows one to express a rich class of temporal properties, including safety, liveness, and fairness properties. Here, we focus on safety properties to which invariants belong.
Table 4. Specification of module \textit{simple\_guidance} in SMV

\begin{verbatim}
MODULE simple\_guidance(activate, deactivate, switch, clear)
VAR mode : {cleared, active};
ASSIGN init(mode) := cleared;
next(mode) := case deactivated | deactivate : cleared;
actuated := active;
1 := mode;
esac;
DEFINE activated := (mode=cleared) \& (activate | switch);
deactivated := (mode=active) \& (clear | switch);
\end{verbatim}

A \textit{module description} in SMV consists of four parts: (1) the \texttt{MODULE} clause, stating the module’s name and its formal (call-by-reference) parameters, (2) the \texttt{VAR} clause, declaring variables needed for describing the module’s behavior, (3) the \texttt{ASSIGN} clause, which specifies the initial value of all variables (cf. \texttt{init}) and how each variable is updated from state to state (cf. \texttt{next}), and (4) the \texttt{DEFINE} clause, which allows one to introduce abbreviations for more complex terms. The main module \texttt{MAIN} of our SMV specification encodes the environment of the FGS, which nondeterministically sends events to the mode logic. This is done by defining variable \texttt{env\_ev} of enumeration type \texttt{env\_events}, which contains all environment events, and by adding “\texttt{init(env\_ev):=env\_events; next(env\_ev):=env\_events}” to the \texttt{ASSIGN} clause. Similar to the Mur\phi\ model, we specify a module \texttt{simple\_guidance} (cf. Table 4) and, thereby, show how Mealy machines may be encoded in SMV. Module \texttt{simple\_guidance} takes the input events \texttt{activate, deactivate, switch}, and \texttt{clear} — which can be either absent or present — as parameters. The state associated with \texttt{simple\_guidance} is variable \texttt{mode} which may adopt values \texttt{cleared} and \texttt{active}. The initial value \texttt{init(mode)} of \texttt{mode} is \texttt{cleared}. The behavioral part of \texttt{simple\_guidance} is described in the \texttt{next(mode)} statement consisting of a \texttt{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 \textit{equality}, \textit{logical conjunction}, and \textit{logical disjunction}, respectively. The terms \texttt{activated} and \texttt{deactivated}, whose values are accessible from outside the module, are defined in the \texttt{DEFINE} clause.

Before we model module \texttt{vertical\_guidance}, we comment on why we have encoded the input event of \texttt{simple\_guidance} using four different signal lines instead of a single event of some enumeration type subsuming all four values. When \texttt{activate, deactivate, switch}, and \texttt{clear} are combined in an enumeration type, we need to identify the value of the input event via a SMV \texttt{case} construct. This induces a circularity which would be detected by the SMV interpreter, i.e., our description of the mode logic would be rejected. One difference between \texttt{simple\_guidance} as a \texttt{module} in SMV and as an \textit{abstract data-type} in Mur\phi\ is that the mode variable is encapsulated within the SMV module and is not a call-by-reference parameter. The behavior of each mode of \texttt{vertical\_guidance}, Pitch, VS, and VGA, can now be described by instantiating \texttt{simple\_guidance},
as is done in the VAR clause in Table 5. 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 terms on the input parameters of vertical_guidance. Note that the functions pitch_event, vs_event, and vga_event used in the Murφ description are encoded here in the DEFINE clause. 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 FGS can be modeled by a sequential algorithm, it must be described by parallel assignments in SMV.

Table 6. Specification of some mode confusion properties in SMV

<table>
<thead>
<tr>
<th>DEFINE</th>
<th>mode_change ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>(!vertical.pitch.mode=cleared &lt;-&gt; AX vertical.pitch.mode=cleared)</td>
<td></td>
</tr>
<tr>
<td>(!vertical.pitch.mode=active &lt;-&gt; AX vertical.pitch.mode=active )</td>
<td>...</td>
</tr>
<tr>
<td>-- check for response to pressing VS button</td>
<td></td>
</tr>
<tr>
<td>SPEC AG (vertical.vs.mode=cleared &amp; env.ev=vs_switch_hit -&gt;</td>
<td></td>
</tr>
<tr>
<td>AX vertical.vs.mode=active)</td>
<td></td>
</tr>
<tr>
<td>SPEC AG (vertical.vs.mode=active &amp; env.ev=vs_switch_hit -&gt;</td>
<td></td>
</tr>
<tr>
<td>AX vertical.vs.mode=cleared)</td>
<td></td>
</tr>
<tr>
<td>-- search for ignored crew inputs (property violated)</td>
<td></td>
</tr>
<tr>
<td>SPEC AG (crew_input -&gt; mode_change)</td>
<td></td>
</tr>
<tr>
<td>-- no unknown ignored crew inputs</td>
<td></td>
</tr>
<tr>
<td>SPEC AG (crew_input &amp; !ignored_crew_input -&gt; mode_change)</td>
<td></td>
</tr>
<tr>
<td>-- search for indirect mode changes (property violated)</td>
<td></td>
</tr>
<tr>
<td>SPEC AG (!crew_input -&gt; mode_change)</td>
<td></td>
</tr>
<tr>
<td>-- no unknown indirect mode changes</td>
<td></td>
</tr>
<tr>
<td>SPEC AG (!crew_input &amp; !indirect_mode_change) -&gt; !mode_change</td>
<td></td>
</tr>
</tbody>
</table>
In SMV, temporal system properties are specified in the Computational Tree Logic (CTL) [3, 5] and may be introduced by the keyword SPEC within the same file as the system description. The properties of interest to us can be specified as CTL formulas of the form AGφ, where AG stands for “always generally,” i.e., every reachable state satisfies property φ. The formula AXφ expresses that all successor states of the current state satisfy φ. In this light, the first formula in Table 6 states: “every reachable state satisfies that, if mode VS in vertical guidance is currently cleared and event vs_switch_hit enters the system, then VS is active in every successor state of the current state.” The symbols -> and <-> used in Table 6 stand for logical implication and equivalence, respectively. The identifiers mode_change, crew_input, indirect_mode_change, and ignored_crew_input are abbreviations of expressions defined in a DEFINE clause, as exemplary 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, an orthogonal treatment of model and property specifications is achieved. The SMV system verified about thirty assertions in slightly more than half an second using 438 BDD nodes and allocated less than 1 MByte memory on a SUN SPARCstation 20. The properties “search for ignored crew inputs” and “search for indirect mode changes” were invalidated as in the Murϕ model. The returned error traces, including the assignments of each variable in every state of the traces, supported the identification of potential problems with the FGS 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 3388 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 × 14 = 3388”).

Summarizing, SMV performed well for our example. CTL supports the convenient specification of mode confusion properties. However, SMV’s system description language is not high-level when compared to Murϕ.

5 Modeling the Mode Logic in Spin

Last, but not least, we explore the utility of the verification tool Spin [7, 8, 18], 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 distributed systems, such as communication protocols. The system description language of Spin, called Promela, allows one to specify nondeterministic processes, message channels, and variables in a C-like syntax. Given a Promela description, whose semantics is again defined as a finite automaton, Spin can perform random or interactive simulations of the system’s execution. Similar to Murϕ, it can also generate a special-purpose verifier in form of a C-program. This program performs an exhaustive exploration of the system’s state space and may check for deadlocks and unreachable code, validate invariants, and verify properties specified in a linear temporal logic [5].
Table 7. Specification of the main process init in Spin

```
init( env_ev=null; do :: atomic{
    if /* body encodes 1 synchr. step */
    :: env_ev=vs_switch_hit /* nondet. choice of env. event */
    :: ... /* 14 cases, for each env. event */
    fi; fgs(env_ev); env_ev=null /* perform synchronous step */
    od }
```

Since our FGS 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 7 encodes the main process, referred to as init in Spin, which is the only process of our model. Here, the global variable env_ev is of type mtype which contains an enumeration of all event and signal names that 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, one may only introduce a single declaration of 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 prevents Spin from storing intermediate states which might arise when executing the body. Thus, we may use this construct for encoding the complex algorithm of the mode logic that performs a single synchronous step. The repetition statement do together with the choice statement if nondeterministically chooses which environment event to assign to env_ev. The reason that we have not simply spelled out fgs(vs_switch_hit), and so on for each environment event, is that fgs needs to be implemented as an inline. Expanding this long inline fourteen times turns out to be inefficient.

Table 8. 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 FGS, is `inline`. This construct may take parameters, such as `mode`, `event`, and `signal` for component `simple guidance` (cf. Table 8). When compiling a Promela description, each occurrence of `simple guidance` in `vertical guidance` is replaced with its body. The modes instantiating parameter `mode` are global variables of type `bit`, where `cleared` and `active` are defined as constants 0 and 1, respectively, using the preprocessor command `#define`. 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 and which returns the result of evaluating the expression on the right hand side. The symbols `==` and `=` denote the `equality` and the `assignment` operator, respectively. Using `simple guidance`, we can specify component `vertical guidance` as another `inline` (cf. Table 9). The body of `vertical guidance` is self-explanatory and similar to the one for `Murφ`. It should only be noted that `guard else` is always executable and that expression `skip` leaves the current system state unchanged. Moreover, functions `pitch_event`, `vs_event`, and `vga_event` are spelled out as `inlines` here.

The verification technique we employed in Spin for reasoning about the FGS, 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 depicted in Table 10, 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 `#defines`. In order to encode expression `mode_change`, we have to keep a copy of the ‘old’ values of all global variables of interest, as in the `Murφ` model. Stating the mode confusion properties in Spin’s version of linear-time logic requires the re-compilation of Spin with compiler option `-DNXT`, such that the next-state operator, which is desired for specifying `mode_change`, becomes available. Although this would have allowed us to proceed as described for SMV, we preferred not to do so. The reason is that Spin does not support the definition of temporal formulas in a modular fashion, i.e., by composing complex formulas from more simpler ones, as SMV does. Thereby, temporal formulas related to mode confusion would be very lengthy and difficult to read. The verification results returned by the Spin verifier are similar to the ones for `Murφ`. The Spin model of the FGS also possesses 242 states and 3,388 transitions (+1 “dummy” transition). Unfortunately, Spin crashes and core dumps when analyzing the invalid assertions `search for ig-`
Table 9. Specification of module \textit{vertical\_guidance} in Spin

\begin{verbatim}
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 || ... }
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 }
\end{verbatim}

nored 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 mentioned that a previous effort to analyze a FGS using Spin suffered from an intractably large state space [15]. That model was then checked for invariants using Spin's \textit{bitstate hashing algorithm}.

Summarizing, carrying out our case study in Spin was feasible but less elegant than in Mur\phi due to the lack of procedure and function constructs in Promela, which had to be encoded using inlines and \#defines. We would like to see a
Table 10. Specification of some mode confusion properties in Spin

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>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)</td>
</tr>
</tbody>
</table>

richer type system in Spin, which can handle more than one \texttt{type} definition. Especially useful to us were Spin’s capabilities to simulate Promela models and to feed back error traces into the simulator. Simulations helped us to quickly identify the causes of ignored crew inputs and indirect mode changes. Beside monitoring variables, we found it useful that Spin highlights the part of the Promela description corresponding to the system state under investigation.

6 Discussion

In this section we discuss the strengths and weaknesses of Murφ, SMV, and Spin regarding their system and property description languages and regarding their capabilities for generating and animating diagnostic information. We restrict our discussion to the observations we made on the FGS case study and refrain from a more general comparison or a comparison along the line of a “feature list.” Not only is our experience with the tools limited, but a single case study will inevitably give a biased picture since tools are developed with different objectives and one will be more tailored for an application than another.

The system description languages of all three verification tools allow us to model the deterministic, synchronous behavior of the FGS, as well as the non-deterministic behavior of the system’s environment. However, Murφ’s language stands out since it (i) implements numerous language constructs and has a rich type system, as found in imperative programming languages, (ii) supports a modular programming style via parameterized procedures and functions, and (iii) permits one to easily adapt the existing PVS specification of the mode logic [13]. However, SMV’s \texttt{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. A major difference between the tools’ languages is that Murφ and Spin allow model encoding using sequential algo-
rithms, whereas SMV requires an algorithm description by parallel assignments. Regarding Promela, one notices that it is designed to specify asynchronous systems. It only offers the process declaration construct proc type for encapsulating code fragments. We used inline declarations to work around this problem. However, an inline construct is no substitute for a procedure mechanism. Although, depending on the employed parameter mechanism, both constructs may semantically coincide, there is an important practical difference. Inlines may blow up system descriptions, thereby making, e.g., syntax checks inefficient. We experienced durations of syntax checks well exceeding ten minutes for some variations of our Spin model. Finally, all three tools are missing the ability to organize events in a taxonomy via sub-typing. Such a concept would help us to divide all events into lateral-mode and vertical-mode events, and further into Pitch events, HDG events, etc.

Regarding the second issue, we also identified important differences among the tools. Since all mandatory and mode confusion properties of interest are invariants, they can be stated as assertions and verified in state-exploration tools, such as Murφ, as well as more general model-checking tools, such as SMV and Spin. When specifying mode confusion properties, a temporal logic is most convenient since it allows one to implicitly refer to adjacent states in program paths using the next-state operator. This is important for describing property mode change which requires one to access the mode variables of adjacent states. In contrast to Murφ, 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. Spin can be employed both as an assertion checker, similar to Murφ, and as a model checker, similar to SMV. Especially, SMV's BDD-based model checker performed very well in our case study since mode logics have the characteristics of Boolean terms which can be represented efficiently using BDDs. However, the small state space of our example system precludes us from fairly comparing the run times of Murφ, SMV, and Spin.

Concerning the third issue, only Spin provides rich features for simulating and animating diagnostic information. Each tool returns an error trace whenever a desired system property is invalidated. Murφ and SMV output a textual description of the error trace, which displays the global variables assignments at all states of the trace. Spin, however, is able to animate error traces using message sequence charts, time sequence panels, and data value panels which are integrated in its graphical user interface, known as Xspin. In our case study involving a synchronous system only the data value panel was of use. This feature and the ability to highlight the source code line corresponding to the current state in the simulation enabled us to detect sources of mode confusion quickly.

7 Conclusions and Future Work

This paper advocates the use of state-exploration techniques for analyzing mode confusion. Compared to theorem provers, model-checking tools are able to verify the properties of interest automatically. When weighing the strengths of Murφ,
SMV, and Spin for our application, it turned out that these are complementary: Murph has the most pleasant system description language, including a rich type system; SMV’s way of integrating temporal logics supports the convenient specification of mode confusion properties; Spin’s capability of animating diagnostic information enables the fast detection of sources of mode confusion.

Regarding future work, our case study should be extended to include more components of today’s digital flight decks and, subsequently, to explore other interesting properties related to mode confusion.

Acknowledgments. We thank Ricky Butler and Steve Miller for many enlightening discussions about mode confusion, as well as Ben Di Vito and the anonymous referees for their valuable comments and suggestions.

References