Using model checking to help discover mode confusions and other automation surprises

John Rushby

Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA

Automation surprises occur when an automated system behaves differently than its operator expects. If the actual system behavior and the operator's "mental model" are both described as finite state transition systems, then mechanized techniques known as "model checking" can be used automatically to discover any scenarios that cause the behaviors of the two descriptions to diverge from one another. These scenarios identify potential surprises and pinpoint areas where design changes, or revisions to training materials or procedures, should be considered. The mental models can be suggested by human factors experts, or can be derived from training materials, or can express simple requirements for "consistent" behavior. The approach is demonstrated by applying the Murϕ state exploration system to a "kill-the-capture" surprise in the MD-88 autopilot.

This approach does not supplant the contributions of those working in human factors and aviation psychology, but rather provides them with a tool to examine properties of their models using mechanized calculation. These calculations can be used to explore the consequences of alternative designs and cues, and of systematic operator error, and to assess the cognitive complexity of designs.

The description of model checking is tutorial and is hoped to be accessible to those from the human factors community to whom this technology may be new.

Keywords: automation surprise, mode confusion, model checking, formal methods, mental model, human-computer interaction

1 INTRODUCTION

Automated systems sometimes behave in ways that surprise their operators [23]. These "automation surprises" are particularly well-documented in the cockpits of advanced commercial aircraft [7,18,22] and several fatal crashes and other incidents are attributed to problems in the "flightcrew-automation interface" [10, Appendix D].

Cognitive scientists have proposed that humans construct "mental models" of the world [14]; in particular, operators and users of an automated system develop such models of the system's behavior and use them to guide their interaction with it [17]. An automation surprise then occurs when the actual behavior of a system departs from that predicted by its operator's mental model.

Complex systems are often structured into "modes" (for example, an aircraft flight management system might have different modes for cruise, initial descent, landing, and so on), and their behavior can change significantly across different modes. "Mode confusion" arises when the system is in a different mode than that assumed by its operator; this is a rich source of automation surprises, since the operator may interact with the system according to a mental model that is inappropriate for its actual mode.

If we accept that automation surprises may be due to a mismatch between the actual behavior of a system and the operator's mental model of that behavior, then one way to look for potential surprises is to construct explicit descriptions of the actual system behavior, and of a postulated mental model, and to compare them.

The discrete behavior of complex control systems can be described in terms of "state machines," which are a formal, mathematical representation that is amenable to various kinds of automated analysis. It is becoming accepted that such formal descriptions can be useful in requirements analysis and other verification and validation activities for critical systems [6]. It is also becoming accepted that state machines provide a natural representation for mental models [13]. Now, if a state machine specification is available for the actual system, and if we can construct one for a plausible mental model, then we could, in principle, "run" the two machines in parallel to see if their behaviors ever diverge from one another. What is potentially valuable about this approach is that if the two state machines have finite state spaces, then a body of techniques from the branch of formal methods in computer science known as "model checking" [5] can be used to compare *all possible* behaviors of the two machines. If a discrepancy is discovered in the behaviors of the two descriptions, a scenario can be generated that gives the sequence of inputs and interactions that manifests the divergence. This provides the designer or analyst with information that can be used to bring the design of the actual system into closer alignment with the mental model (either by changing its behavior, or by improving the cues it provides to its operator), or that can be used to guide the formation of more appropriate mental models through improvements in documentation or operator training.

This approach can be compared with current practice, which relies on development of a simulation or rapid prototype of the proposed design that can be evaluated experimentally (e.g., by having test pilots fly simulated missions). A test pilot may approximate the behavior and expectations of a line pilot more accurately than a hypothesized mental model encoded as a state machine, but only a limited number of experiments can be performed in this way, and these will cover only a small fraction of the possible joint behaviors of the automation and the pilot. The model checking approach uses a simple model of the pilot, but then examines all possible interactions and joint behaviors. The evidence from other applications of model checking (e.g., in hardware and protocol design) is that *complete* exploration of somewhat simplified models generally detects more problems than *partial* exploration of the real thing (as in testing and simulation). This is not to suggest that the model checking approach should replace experiments with human operators, but that it is likely to be a useful adjunct that can identify some problems that may be missed by experiments, and it may also identify some problems earlier or more cheaply than experiments.

There are some obvious difficulties with the model checking approach: the state machine descriptions of real systems often are not finite-state, or have finite state spaces that are too large for a model checker to analyze exhaustively (this may be so, for example, if the state includes numeric quantities); also, there is no direct way to access an operator's mental model for the purpose of encoding it as a state machine. I am of the opinion that both these difficulties can be overcome by *abstraction* and *generalization*. Because we are performing refutation rather than verification (i.e., we are looking for potential bugs—automation surprises in this case—not trying to prove their absence) we do not need to model *all* the details of the actual system. For example, to examine mode confusion, we need model only the mode transitions of the system, not the details of its behavior within those modes; and to examine the mode transition at, say, a capture altitude, we need only to model whether or not the airplane is at or close to the capture altitude and not its exact altitude.

Similarly, because our goal is to discover error-prone designs, not to gain psychological insight into human behavior, we do not need to examine the mental model of any particular operator. We will be content to check whether the actual system behavior violates plausible models and natural expectations (e.g., as suggested by training materials). I believe that human factors specialists, in combination with those involved with developing, analyzing, documenting, and using the system concerned, should find it quite easy to suggest suitably generalized mental models.

2 AN EXAMPLE SCENARIO

I describe the proposed method using an example reported by Palmer [18, Case 2]. This example has also been analyzed by Leveson and Palmer [15]; I compare their approach with mine in Section 4.

The example is one of five altitude deviation scenarios observed during a NASA study in which twenty-two airline crews flew realistic two hour missions in DC-9 and MD-88 aircraft simulators. To follow the scenario, it is sufficient to understand that the autopilot can be instructed to cause the aircraft to climb or to hold a certain altitude through the setting of its "pitch mode." In VERT SPD (Vertical Speed) mode the aircraft climbs at the rate set by the corresponding dial (e.g., 2,000 feet per minute); in IAS (Indicated Air Speed) mode, it climbs at whatever rate is consistent with holding the air speed set by another dial (e.g., 256 knots); in ALT HLD (Altitude Hold) mode, it holds the current altitude. In addition, certain "capture modes" may be *armed*. If ALT (Altitude) capture is armed, the aircraft will only climb as far as the altitude set by the corresponding dial, at which point the pitch mode will change to ALT HLD; if the capture mode is not armed, however, and the pitch mode is VERT SPD or IAS, then the aircraft will continue climbing indefinitely. The behavior of this system is complicated by the existence of an ALT CAP (Altitude Capture) pitch mode, which is intended to provide smooth leveling off at the desired altitude. The ALT CAP pitch mode is entered automatically when the aircraft gets close to the desired altitude and the ALT capture mode is armed (do not confuse the ALT CAP *pitch* mode with the ALT *capture* mode). The ALT CAP pitch mode disarms the ALT capture mode and causes the plane to level off at the desired altitude, at which point it enters ALT HLD pitch mode.

The following scenario description is slightly reworded from Palmer's original in order to fit my terminology.

> The crew had just made a missed approach and had climbed to and leveled at 2,100 feet. They received the clearance to "... climb now and maintain 5,000 feet..." The Captain set the MCP (Master Control Panel) altitude window to 5,000 feet (causing ALT capture mode to become armed), set the autopilot pitch mode to VERT SPD with a value of approximately 2,000 ft. per minute and the autothrottle to SPD mode with a value of 256 knots. Climbing through 3,500 feet the Captain called for flaps up and at 4,000 feet he called for slats retract. Passing through 4000 feet, the Captain pushed the IAS button on the MCP. The pitch mode became IAS and the autothrottles went to CLAMP mode. The ALT capture mode was still armed. Three seconds later the autopilot automatically switched pitch mode to ALT CAP. The FMA (Flight Mode Annunciator) ARM window went from ALT to blank and the PITCH window showed ALT CAP. A tenth of a second later, the Captain adjusted the vertical speed wheel to a value of about 4,000 feet a minute. This caused the pitch autopilot to switch modes from ALT CAP to VERT SPD. As the altitude passed through 5,000 feet at a vertical velocity of about 4,000 feet per minute, the Captain remarked, "Five thousand. Oops, it didn't arm." He pushed the MCP ALT HLD button and switched off the autothrottle. The aircraft then leveled off at about 5,500 feet as the "*altitude—altitude*" voice warning sounded repeatedly.

An aircraft climbing through its assigned altitude (and potentially into the airspace assigned to another aircraft) is colloquially called a "bust," so Palmer refers to the scenario above as the "kill-the-capture bust." However, the basic problem is present whether or not it leads to a bust, so I prefer to speak of it as the "kill-the-capture surprise." The source of the surprise is the interaction of the pitch and capture modes and, in particular, with the way the ALT CAP pitch mode disarms the ALT capture mode. When the ALT capture mode is armed, changing the pitch mode between IAS and VERT SPD, or changing the values set by their corresponding dials, simply changes *how* the plane climbs to the desired altitude. When the aircraft gets close to the desired altitude, however, it autonomously enters ALT CAP pitch mode and disarms ALT capture mode. If the pitch mode is then changed to IAS or VERT SPD, the aircraft will climb without limit in the newly selected mode, since the ALT capture mode is now disarmed. The only indication to the pilot that the autopilot is in this vulnerable combination of modes is that the ARM window of the FMA changes from ALT to blank.

3 ANALYZING THE EXAMPLE

To see how model checking techniques could reveal the existence of the kill-the-capture surprise, we first need to construct a mental model that a pilot might plausibly employ. Different pilots might have different mental models, and we cannot know what they are, but a plausible basic tenet might be that the pitch mode controls *how* the aircraft climbs, and the capture mode controls whether there is a *limit* to the climb. Another plausible basic tenet is that once capture mode is armed, it becomes disarmed only when the aircraft reaches the desired altitude (unless the pilot manually disarms it). Since this mental model makes no mention of the ALT CAP pitch mode, it obviously differs from the real system. This does not necessarily mean that the system harbors a surprise, however, because a mental model *should* suppress details considered unnecessary to understanding how to operate the system. The pilot might well be aware of the ALT CAP pitch mode and of its role in leveling the plane off—and may even be aware that the ALT CAP pitch mode and the ALT capture mode interact in some way—but could believe this is merely the implementation of the ideal capture mode assumed in the mental model. To discover whether a surprise really does reside here, we need to "run" the state machines representing the actual system and the mental model on all possible sequences of inputs and compare their behavior.

I now present an automated analysis of this example using the Murϕ (pronounced "Murphy") state exploration system developed by David Dill's group at Stanford University [8]. Strictly speaking, Murϕ is not a model checker (that term is properly reserved for tools that test whether a transition system is a Kripke model for some temporal logic formula [4]), but the term "model checking" is loosely applied to any tool that uses (explicit or symbolic) state exploration techniques. Systems are described in Murϕ by specifying their *state variables*, and a series of *rules* that indicate the actions that can be performed by the system and the circumstances under which they can be performed. Properties that should hold in some or all states can be given as part of a Murϕ specification (as *assertions* and *invariants*, respectively), and the Murϕ system undertakes a search of all reachable states to ensure that the given properties do indeed hold. If they do not, Murϕ prints an error trace that describes the circumstances leading to the violation. Those who have some familiarity with computer programming should find it fairly easy to interpret Murϕ specifications and can think of Murϕ as performing exhaustive simulation of the specified system, so that *all possible* behaviors are examined; this is feasible because the number of states (i.e., combinations of values of the system variables) is finite (although it may be very large). In hardware and protocol applications, it is routine to apply Murϕ to specifications that are thousands of lines long and that have tens of millions of reachable states.

At the level of abstraction appropriate for our investigation, the actual behavior of the example system can be described in terms of two state variables, `pitch_mode` and `capture_armed`, which are specified in Murϕ as follows.

```
Type
  pitch_modes: enum
   {vert_speed, ias, alt_cap, alt_hold};
Var
  pitch_mode: pitch_modes;
  capture_armed: boolean;
```

These declarations specify that `pitch_mode` can take one of the four values from the enumerated type `pitch_modes`, and that `capture_armed` is a `boolean`. The `pitch_mode` state variable represents the autopilot's pitch mode in a direct way,[1] while the `capture_armed` variable encodes whether the ALT capture mode is armed. The initial state of the system is specified in the Murϕ `Startstate` declaration as follows.

```
Startstate
Begin
  clear pitch_mode;
  capture_armed := false;
End;
```

The `clear` construct chooses some arbitrary initial value.

Now we can specify the actions of the system by means of Murϕ rules as follows.

```
Rule "IAS"
Begin
  pitch_mode := ias;
End;
```

This rule corresponds to the pilot engaging the IAS pitch mode (whether by pushing its button, or entering a value in its dial is unimportant at this level of abstraction). It has no guards, meaning that it can "fire" at any time, and has the effect of setting the `pitch_mode` state variable to the value `ias`. The string `IAS` is simply the name used to identify the rule.

The `HLD` and `VSPD` rules are similar and correspond to the pilot engaging the ALT HLD and VERT SPD pitch modes, respectively.

```
Rule "HLD"
Begin
  pitch_mode := alt_hold;
End;

Rule "VSPD"
Begin
  pitch_mode := vert_speed;
End;
```

[1] I use slightly different names to distinguish the pitch modes of the Murϕ model from those used in the narrative description, but the intended correspondence should be obvious.

Notice that I do not model the parameters (e.g., speed, climb rate) used by the various pitch modes, nor the dials that are used to set these parameters. We are concerned only with the basic mode transitions, so it is appropriate to omit these details. I should also note that I have no idea whether the specification being developed here accurately represents the real DC-9 or MD-88 autopilots—my purpose is only to explain the approach, not to present an industrial application.

The following rule corresponds to the pilot pushing the ALT capture mode button. I have chosen to specify it as a toggle: initially the mode is not armed, pushing the button arms it, and pushing it again disarms it once more.

```
Rule "ALT CAPTURE"
Begin
  capture_armed := !capture_armed;
End;
```

The next rule corresponds to the aircraft approaching the selected altitude. I call the rule `near` and use lower case to distinguish it from the upper case names used for the rules associated with pilot actions that were presented above.

This rule only has an effect when `capture_armed` is `true`, in which case it sets `pitch_mode` to `alt_cap` and `capture_armed` to `false`. (Those familiar with Murϕ might wonder why I did not use `capture_armed` as a guard on the rule; the reason is that I will later need to modify the rule to incorporate the mental model and the present arrangement is more convenient for this purpose.)

```
Rule "near"
Begin
  If capture_armed Then
    pitch_mode := alt_cap;
    capture_armed := false;
  Endif;
End;
```

The next rule corresponds to the aircraft reaching the selected altitude when the pitch mode is ALT CAP, thereby causing a transition to ALT HLD. I originally specified this as follows,

```
Rule "arrived"
Begin
  If pitch_mode = alt_cap Then
    pitch_mode := alt_hold;
  Endif;
End;
```

However, we also need to account for the possibility that the pilot arms ALT capture mode when the aircraft is already at the selected altitude. This circumstance is dealt with by the second `If-Then` clause of the following revised rule, which disarms the ALT capture mode and bypasses ALT CAP to enter the ALT HLD pitch mode directly. In this and in later specification fragments, faint type is used for parts presented previously, and dark type for the new or changed material.

```
Rule "arrived"
Begin
  If pitch_mode = alt_cap Then
    pitch_mode := alt_hold;
  Endif;
  If capture_armed Then
    capture_armed := false;
    pitch_mode := alt_hold;
  Endif;
End;
```

Some readers may consider the specification of the last two rules to be excessively loose: for example, there is nothing in the specification that excludes physically impossible sequences of events, such as `arrived` followed by `near`, or several `near`s in succession. This looseness is typical in model checking: by omitting to specify constraints that are enforced by the physical world, or by other components of the system, we allow the specified system to have *more* behaviors than is actually possible. If this less constrained description does not exhibit the flaws we are concerned about, then certainly a more tightly specified system (having strictly fewer behaviors) will not exhibit them.[2] Only if we get "false drops" (i.e., apparent errors that would be excluded if the model was more detailed) will we need to refine the model.

We have now specified the behavior of the actual system and can turn to the specification of an idealization that constitutes a plausible mental model. A suitable model could be one where reaching the desired altitude causes ALT capture mode to be turned off and the pitch mode to change to ALT HLD; the `near` event is not significant to this mental model.

To specify this, I begin by adding a `boolean` state variable called `ideal_capture` that will record the state of the altitude capture mode in the mental model. This variable is initialized to `false` in the modified `Startstate` shown below.

[2]This is true for what are technically called *safety* properties; it is not true of *liveness* properties. All the properties considered here are safety properties.

```
Var
  pitch_mode: pitch_modes;
  capture_armed: boolean;
  ideal_capture: boolean;

Startstate
Begin
  clear pitch_mode;
  capture_armed := false;
  ideal_capture := false;
End;
```

The ideal capture mode is toggled by the ALT capture mode button in the same way as the arming of the real mode, so I add this to the specification of the `ALT CAPTURE` rule.

```
Rule "ALT CAPTURE"
Begin
  capture_armed := !capture_armed;
  ideal_capture := !ideal_capture;
End;
```

The ideal capture mode is unaffected by the `near` event, so that rule is left unchanged. If an `arrived` event occurs when the ideal capture mode is armed, then the mode is disarmed. This is specified by adding a third `If-Then` clause to the corresponding rule as follows.

```
Rule "arrived"
Begin
  If pitch_mode = alt_cap Then
    pitch_mode := alt_hold;
  Endif;
  If capture_armed Then
    pitch_mode := alt_hold;
    capture_armed := false;
  Endif;
  If ideal_capture Then
    ideal_capture := false;
  Endif;
End;
```

We now need to relate the ideal capture mode of the mental model to the modes of the actual system. The actual system is set to capture the desired altitude if *either* the pitch mode is ALT CAP *or* the capture mode is ALT. In terms of the Murϕ model this condition is given by the expression

```
(capture_armed | pitch_mode = alt_cap) .
```

The modes of the actual system and of the mental model are consistent with each other if this expression is `true` exactly when `ideal_capture` is also `true`. We can state this in a Murϕ *invariant* as follows.

```
Invariant ideal_capture =
    (capture_armed | pitch_mode = alt_cap);
```

At this point, we have constructed specifications for the mode transitions of the actual system and of the mental model and stated, as an invariant, the condition for these to be consistent with each other. We can now proceed to examine whether any sequence of events can violate the invariant by causing Murϕ to perform exhaustive exploration of all the reachable states of the specification. Murϕ does this by systematically firing the rules of the specification in different orders until either an error is found or all possible cases have been examined. In this example, we receive the error trace shown in Figure 1.

```
Error trace for the error:

     Invariant "Invariant 0" failed.

Startstate Startstate 0 fired.
pitch_mode:vert_speed
capture_armed:false
ideal_capture:false
----------
Rule ALT CAPTURE fired.
capture_armed:true
ideal_capture:true
----------
Rule near fired.
pitch_mode:alt_cap
capture_armed:false
----------
Rule VSPD fired.
The last state of the trace (in full) is:
pitch_mode:vert_speed
capture_armed:false
ideal_capture:true
----------

End of the error trace.
```

Figure 1: First Error Trace

This is exactly the scenario that manifested the automation surprise described in the previous section: the pilot engages the ALT capture mode, the aircraft approaches the desired altitude and automatically disarms the capture mode and engages the ALT CAP pitch mode, and then the pilot engages VERT SPD pitch mode. At this point the ideal capture mode is still armed, but that of the actual system is not. Murϕ found this scenario in 0.24 seconds (on a 400 MHz Pentium II with 256 MB of memory running Linux).

Leveson and Palmer also detected the potential for this surprise using their method [15] (I discuss the differences between their method and mine in the following section), and suggested that it could be eliminated by making two changes to the actual system. (My specification is organized differently to theirs, so the following translates the intent of their changes into the terms of my specification.)

- Cause the `arrived` event to engage ALT HLD pitch mode when the ALT capture mode is armed (as opposed to when the pitch mode is ALT CAP), and
- Cause disarming of ALT capture mode to occur when the pitch mode becomes ALT HLD rather than ALT CAP.

The intuition is that the ALT CAP pitch mode should be regarded as engaging a particular control law that determines *how* the aircraft flies the capture trajectory, but the ALT capture mode stays in effect until the desired altitude is achieved.

The first of the changes above is accomplished in our specification by deleting the first `If-Then` clause in the `arrived` rule, so that it becomes the following (I use a strikeout like ~~`this`~~ to indicate text that is removed).

```
Rule "arrived"
Begin
  If pitch_mode = alt_cap Then
    pitch_mode := alt_hold;
  Endif;
  If capture_armed Then
    pitch_mode := alt_hold;
    capture_armed := false;
  Endif;
  If ideal_capture Then
    ideal_capture := false;
  Endif;
End;
```

The second change requires `capture_armed := false` to be removed from all rules that contain the assignment `pitch_mode := alt_cap` and added to all rules that contain the assignment `pitch_mode := alt_hold`. The `arrived` rule as modified above already satisfies this condition, but the `HLD` rule must be changed as follows.

```
Rule "HLD"
Begin
  pitch_mode := alt_hold;
  capture_armed := false;
End;
```

And the `near` rule must be changed to the following.

```
Rule "near"
Begin
  If capture_armed Then
    pitch_mode := alt_cap;
    capture_armed := false;
  Endif;
End;
```

If we cause Murϕ to perform state exploration on this modified specification we obtain the error trace shown in Figure 2, which highlights a potential surprise introduced by the changes just made to the specification: if the pilot engages ALT HLD pitch mode while ALT capture mode is armed, the modified actual system will disarm the capture mode, while it remains armed in the mental model (and remained so in the actual system prior to the change). Inspection of Leveson and Palmer's specification indicates that this issue is present in their specification also, and is not just an artifact of my encoding. Several interpretations seem plausible and reasonable for the intended behavior (and I have no idea what happens in this circumstance on a real aircraft), so we could modify either the description of the actual system, or that of the mental model, or both. I choose to suppose that ALT HLD pitch mode causes the aircraft to hold the current altitude, but that it should mask rather than disarm ALT capture mode—which will become active again if the pitch mode is changed to IAS or VERT SPD. This is consistent with the current mental model, and the prior system model, so the description of the actual system should be changed

```
Error trace for the error:

     Invariant "Invariant 0" failed.

Startstate Startstate 0 fired.
pitch_mode:vert_speed
capture_armed:false
ideal_capture:false
----------
Rule ALT CAPTURE fired.
capture_armed:true
ideal_capture:true
----------
Rule HLD fired.
The last state of the trace (in full) is:
pitch_mode:alt_hold
capture_armed:false
ideal_capture:true
----------

End of the error trace.
```

Figure 2: Second Error Trace

by undoing the change just made to the HLD rule (the other changes remain in place). This revision to the specification produces yet another error trace, shown in Figure 3.

```
Error trace for the error:

     Invariant "Invariant 0" failed.

Startstate Startstate 0 fired.
pitch_mode:vert_speed
capture_armed:false
ideal_capture:false
----------
Rule ALT CAPTURE fired.
capture_armed:true
ideal_capture:true
----------
Rule near fired.
pitch_mode:alt_cap
----------
Rule ALT CAPTURE fired.
The last state of the trace (in full) is:
pitch_mode:alt_cap
capture_armed:false
ideal_capture:false
----------

End of the error trace.
```

Figure 3: Third Error Trace

This highlights one more potential surprise in our specification: if the pilot presses the ALT button to arm the ALT capture mode and later, but before the desired altitude has been achieved, presses it again, the mental model indicates that the capture mode will be disarmed. This will be true of the actual system if the second button press occurs before the aircraft is near enough to the desired altitude to engage the ALT CAP pitch mode. But if the second button press occurs after ALT CAP mode has been engaged, then the actual system does indeed disarm the ALT capture mode, but the aircraft will still be in the ALT CAP pitch mode, and hence still flying a capture trajectory.[3]

[3]This surprise is present, in a different form, in the original specification as well: if the ALT capture mode button is pressed after ALT CAP pitch mode has been engaged, then the original specification will arm the ALT capture mode (since it will have been disarmed when ALT CAP pitch mode was entered), but disarm the ideal capture mode.

Leveson and Palmer's specification uses a "push-pull," rather than a toggle arrangement for the ALT capture mode button, so this issue does not arise in their specification. However, I suspect that something like it must occur because their button seems to hold a state (i.e., "pushed in" or "pulled out") that is not synchronized with the internal system state.

The best resolution to this issue is not obvious, so for simplicity I simply add a guard to the ALT CAPTURE rule that will cause ALT button presses to be ignored when the pitch mode is ALT CAP.

```
Rule "ALT CAPTURE" pitch_mode != alt_cap ==>
Begin
  capture_armed := !capture_armed;
  ideal_capture := !ideal_capture;
End;
```

With this change, we finally bring the behaviors of the actual system and the mental model into alignment; Murϕ confirms this as shown in Figure 4.

```
Status:

     No error found.

State Space Explored:

     7 states, 41 rules fired in 0.23s.

Rules Information:

     Fired 7 times   - Rule "arrived"
     Fired 7 times   - Rule "near"
     Fired 7 times   - Rule "VSPD"
     Fired 7 times   - Rule "IAS"
     Fired 7 times   - Rule "HLD"
     Fired 6 times   - Rule "ALT CAPTURE"
```

Figure 4: Murϕ Reports Success

The output displays of the system have not been considered in the treatment presented so far. The quality of information presented to the operator is a critical factor in reducing automation surprises and mode confusion, and should certainly be examined in any comprehensive analysis. As a final illustration, I will indicate how this can be done using the model checking approach: the information displayed will be specified as part of the system description, the way it used by the operator will be part of mental model, and the interaction of these elements will be examined as part of the automated analysis.

An operator does not have access to all the data available to the actual system, and hence may not always know when a circumstance arises that calls for a mode change. Well-designed automation should keep the operator informed of these circumstances through its output displays—conversely, a badly designed system may make it impossible for the operator to accurately predict the consequences of certain actions. (Degani and Heymann describe such a situation in a current autopilot [7].) In addition, operators have limited memory and attention span and should not be expected to retain the internal state of their mental model infallibly. Good output displays should provide information that allows operators to "reload" their mental state.

We can model an occasionally forgetful operator by adding a "whoops" rule to our specification as follows.

```
Rule "whoops"
Begin
  ideal_capture := !ideal_capture;
End;
```

This rule flips the value of `ideal_capture` and is invoked nondeterministically to model an operator who not merely forgets the state of his mental model, but "misremembers" the wrong one. Obviously, Murϕ detects numerous errors when this rule is added to the model without further adjustments.

Let us suppose, however, that the actual system turns on a light exactly when ALT capture mode is armed. The pilot's method of operation is changed so that, before performing any operation, she sets the state of the ideal capture mode of her mental model to be that indicated by the light. This is specified by adding the assignment `ideal_capture := capture_armed` to the beginning of the rules that represent pilot actions—namely, `IAS`, `VSPD`, `HLD`, and `ALT CAPTURE`.[4] Murϕ will again find that the `Invariant` fails in numerous circumstances (e.g., following the `whoops` rule). However, the only time it is really important for the actual system and the mental model to be in agreement is *following* any action by the pilot (so that the pilot can accurately predict the consequences of her actions). This can be accomplished by replacing the `Invariant` (which is evaluated after *every* rule) by `Assert` statements in the bodies of the four "pilot action" rules, as shown in Figure 5.

Murϕ reports no errors in this modified specification. (It is not hard to see by inspection that this must be so.) Additional experimentation will reveal that the guard on the `ALT CAPTURE` rule is still required, and that the only time `ideal_capture` does depart from the actual system state is in the `near` event when this follows a `whoops`. We regard this as unimportant, because it does not lead to a surprise in any action performed by the pilot. Combining this analysis with earlier ones, we conclude that the current design does not harbor surprises for a forgetful operator who follows the display light, nor for a nonforgetful one (independently of the light).

```
Rule "IAS"
Begin
  ideal_capture := capture_armed;
  pitch_mode := IAS;
  Assert ideal_capture =
    (capture_armed | pitch_mode = alt_cap);
End;

Rule "VSPD"
Begin
  ideal_capture := capture_armed;
  pitch_mode := vert_speed;
  Assert ideal_capture =
    (capture_armed | pitch_mode = alt_cap);
End;

Rule "HLD"
Begin
  ideal_capture := capture_armed;
  pitch_mode := alt_hold;
  Assert ideal_capture =
    (capture_armed | pitch_mode = alt_cap);
End;

Rule "ALT CAPTURE" pitch_mode != alt_cap ==>
Begin
  ideal_capture := capture_armed;
  capture_armed := !capture_armed;
  ideal_capture := !ideal_capture;
  Assert ideal_capture =
    (capture_armed | pitch_mode = alt_cap);
End;
```

Figure 5: The "Pilot Action" Rules Modified to Use the Display Light

As noted earlier, in addition to global `invariants`, Murϕ also allows `assert` statements in the bodies of its rules; these provide a way of checking additional properties, such as those that should hold on mode *transitions* (as opposed to when the system is *in* a mode). For example, we can add an `assert` statement to the rule `arrived` to check that the pitch mode is indeed ALT HLD whenever the ideal capture mode is disarmed as a result of the `arrived` event.

[4] Because the light displays exactly the value of the state variable `capture_armed`, we do not need to introduce a new state variable or function to represent it.

```
Rule "arrived"
Begin
  If capture_armed Then
    pitch_mode := alt_hold;
    capture_armed := false;
  Endif;
  If ideal_capture Then
    ideal_capture := false;
    Assert pitch_mode = alt_hold;
  Endif;
End;
```

This check is satisfied (provided there are no `whoops` events) in the final specification presented above, but detects issues (that were also found by the `Invariant`) in earlier specifications.

A notable property of all the analyses performed here is their simplicity and efficiency. Once the initial investment has been made to formalize the actual system behavior (and this might already have been done for other requirements analysis purposes), making adjustments to the system or mental model, performing state exploration, and examining the results is the work of minutes (none of the analyses described here took more than 0.25 seconds to run). Of course, the specifications used here have almost trivially small state spaces (from 7 to 14 states depending on the specification) and require very few rules to be fired (from 14 to 96). However, the evidence from other fields of application is that state exploration and model checking techniques scale quite well: it is routine to examine tens of millions of states with explicit enumeration, and often vastly more using symbolic methods.

4 DISCUSSION

There is much excellent work in the fields of system design, aviation psychology, ergonomics and human factors that seeks to understand and reduce the sources of operator error in automated systems. The work described here is intended to complement these existing studies by providing a practical, mechanized means to examine system designs for features that may be error prone. Human factors and other studies provide an idea of what to look for, and the work described here provides a method to look for it. The method uses existing tools for model checking and state exploration that have, in other kinds of applications, scaled successfully to quite large systems.

Model checking is a member of the class of techniques known as "formal methods," and there is also prior work, principally by Leveson and her colleagues, in applying formal methods to the problems of automation surprises [16]. Leveson's work uses an evolving list of design features (currently there are about 15 items on the list) that are prone to cause operator mode awareness errors. These features provide criteria that can be applied to a formal system description in order to root out design elements that would repay additional consideration. Leveson and Palmer [15] apply this approach to the kill-the-capture surprise considered here. One of the error-prone design features identified by Leveson is use of "indirect" mode transitions which occur without explicit operator input. She and Palmer construct a formal specification of the relevant parts of the MD-88 autopilot and examine it (by hand) to detect such transitions. This approach successfully leads them to discover the indirect pitch mode transition to ALT CAP, and the confusing interaction between the pitch and capture modes.

Automation is not a replacement for careful manual review of perspicuous, carefully structured formal specifications, but it is a valuable adjunct whose value becomes greater as the specifications get larger and their analysis correspondingly more difficult. The example considered here is almost trivially small, yet its automated analysis raised an issue that was not reported in Leveson and Palmer's manual examination—namely, that the repaired specification causes selection of the ALT HLD pitch mode to disarm the ALT capture mode. To be fair, Leveson and Palmer explicitly note that their repair to the kill-the-capture surprise "may violate other goals or desired behaviors of the autoflight system—the designers would have to determine this when deciding what solution to use. In addition, a more sophisticated solution may be required, e.g., a hysteresis factor may need to be added to the mode transition logic to avoid too rapid 'ping-ponging' transitions between pitch modes." Nonetheless, the fact remains that the approach used here found the original kill-the-capture surprise, found this issue with the repaired specification, and found another issue (namely that pressing the ALT capture mode button after the pitch mode has changed to ALT CAP does not disarm the altitude capture)—all with essentially no effort. It also allowed rapid and inexpensive exploration of an occasionally forgetful operator and of the efficacy of displays in mitigating this problem. The ability to use formal analysis in this manner for active design exploration is an underappreciated attribute of formal methods—and one that depends critically on efficiently mechanized methods of analysis.

Many authors have observed that model checking and other forms of automated formal analysis can usefully be applied to requirements specifications. Indeed, Leveson and Palmer propose that "the pilot's mental model includes a cause and effect relationship between arming the altitude capture and eventually... acquiring that altitude and holding it" and this phraseology almost immediately invites formulation in temporal logic (such logics provide an *eventually* modality), which is the classical application of model checking. A little thought and experimentation, however, reveals that it is generally difficult or impossible to formulate a mental model, or the expectations it engenders, within the limited expressivity of a temporal logic. In the example just quoted, it would be necessary to add the caveat "provided the pilot does not explicitly disarm altitude capture" and this is not easily stated in temporal logic. Furthermore, the suggested formulation relates a mode control issue ("arming the altitude capture") to an external event ("acquiring that altitude"). In order to examine this relationship, our formal model would need to include some treatment (e.g., qualitative physics) for the notion of an aircraft "climbing" and its relation to "altitude" that would add greatly to its complexity.

The novelty and utility in the approach used here is that it moves specification of the desired behavior from the property/assertion language of the model checker into its system specification language. That is to say, the desired property is conceived as a mental model that is specified as a state machine running in parallel with the state machine that specifies the actual system. This seems consistent with representations already employed in the human factors community [13], and provides the expressiveness needed to accommodate possibilities such as the pilot explicitly disarming altitude capture, while allowing the correctness criterion to be stated in terms of (idealized) modes rather than external physical realities (such as reaching a desired height). The property/assertion language of the model checker or state exploration system is used simply to state (as an invariant) the desired correspondence between actual and idealized modes.

In more technical terms, we are really checking a simulation or conformance relation between two system descriptions (the mental model and the actual system). This is a basic capability of "model checkers" for process algebras, such as the FDR tool for CSP [19], but must be achieved somewhat indirectly in tools based on state transition relations such as Murϕ. The approach used here works in simple cases; in more complicated cases, it may be necessary to use superposition and an explicit abstraction (or, dually, refinement) relation to connect the two system descriptions (see [20] for a tutorial explanation).

Other ways to apply formal methods to examination of human factors issues are exemplified in the work of Sage and Johnson [21] and of Butler *et al* [3]. These specify the behavior of the actual automation as a state machine in much the same way as here, but specify user expectations as logical formulas. These are examined by model checking in the case of Sage and Johnson and by theorem proving in the case of Butler *et al*. The disadvantage to these approaches is that they support only a simple model of the user; the concept of mental model arose in psychology (and in artificial intelligence [11]) precisely because deductive rules were found to be inadequate models of human cognition.

The expressiveness provided by the mental models approach opens a number of interesting possibilities for modeling and analysis in addition to those already illustrated.

- We can examine the consequences of a faulty operator: simply endow the operator model with selected faulty behaviors and observe their consequences. The effectiveness of remedies such as lockins and lockouts, or improved displays, can be evaluated similarly.

- We can examine the load placed on an operator: if the simplest mental model that can adequately track the actual system requires many states, or a moderately complicated data structure such as a stack, then we may consider the system too complex for reliable human operation. We can use the same method to evaluate any improvement achieved by additional or modified output displays, or by redesigning the system behavior. This could provide a formal way to evaluate the methods proposed by Vakil and Hansman for mitigating the complexity of interfaces [24].

- We can examine the accuracy of an operator instruction manual by formulating it as a transition system and comparing it to a similar formulation of its actual system—just as we formulated and compared a mental model with its actual system in the example.

- Javaux and Polson [13] suggest that mental models have a predictable structure that is due to unconscious simplifications that elide parts that are rarely employed. We could take the model suggested by training material, apply the simplifying processes of Javaux and Polson, and check whether the resulting model (which may be a fairly good approximation to the mental models of real operators) is an adequate representation of the real system.

- We could take a mental model from one system (e.g., an A320) and check it against a different actual system (e.g., an A340). Discrepancies could highlight areas that should be given special attention in training programs to convert operators from one system to the other.

- We could extend the approach to multi-operator systems: for example, the air traffic control system, where the controller and the pilot may act according to different mental models of the same situation.

A limitation to all these analyses, and to our whole approach, is that we are modeling only a small fragment of the cognitive processes involved in human-computer interaction. Our approach is silent, for example, on problems that might be due to an operator's difficulty in recalling the right mental model, or to excessive demands on an operator's attention. There is very interesting work by Bowman and Faconti [2] and by Duke and Duce [9] that applies formal methods to deeper models of cognition and this allows them to detect different kinds of issues than the automation surprises described here. I consider all these approaches to be complementary and representative of a very promising general direction: the detection of potential human factors problems by explicitly comparing the design of a system against a model of some aspect of human cognition using mechanized formal methods. Models of different aspects of cognition are likely to reveal different kinds of problems. The approach described here uses simple mental models to find design flaws that lead to automation surprises, and it seems very effective for that purpose.

In the future, I hope that this approach will be developed and documented further, and extended in the directions listed above. I also look forward to evaluating it on a more realistic example.

Acknowledgments

I am grateful for help and encouragement received from several people. Comments from Judy Crow improved the presentation of this material. David Dill suggested the forgetful operator and a number of the other extensions listed above. I received very helpful feedback from talks at NASA Ames and Langley Research Centers, and at Rockwell Collins. Bill Rogers and Trish Ververs provided several useful suggestions, and Denis Javaux been an inspiring source of information and ideas.

This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044, by the National Science Foundation under contract CCR-9509931, and by NASA Langley Research Center under contract NAS1-20334.

REFERENCES

1. Abbott, K., Speyer, J.-J., and Boy, G., editors. *International Conference on Human-Computer Interaction in Aeronautics: HCI-Aero 2000*, Toulouse, France, September 2000.

2. Bowman, H. and Faconti, G. Analysing cognitive behaviour using LOTOS and Mexitl. *Formal Aspects of Computing*, 11(2):132–159, 1999.

3. Butler, R. W., Miller, S. P., Potts, J. N., and Carreño, V. A. A formal methods approach to the analysis of mode confusion. In *17th AIAA/IEEE Digital Avionics Systems Conference*, Bellevue, WA, October 1998.

4. Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

5. Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, 1999.

6. Crow, J. and Di Vito, B. L. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.

7. Degani, A. and Heymann, M. Pilot-autopilot interaction: A formal perspective. In Abbott et al. [1], pages 157–168.

8. Dill, D. L. The Murϕ verification system. In Alur, R. and Henzinger, T. A., editors, *Computer-Aided Verification, CAV '96*, Volume 1102 of Springer-Verlag *Lecture Notes in Computer Science*, pages 390–393, Springer-Verlag, New Brunswick, NJ, July/August 1996.

9. Duke, D. and Duce, D. The formalization of a cognitive architecture and its application to reasoning about human computer interaction. *Formal Aspects of Computing*, 11(6):665–689, 1999.

10. The interfaces between flightcrews and modern flight deck systems. Report of the FAA human factors team, Federal Aviation Administration, 1995. Available at `http://www.faa.gov/avr/afs/interfac.pdf`.

11. Halpern, J. Y. and Vardi, M. Y. Model checking vs. theorem proving—a manifesto. In *2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 325–334, Association for Computing Machinery, Cambridge, MA, April 1991.

12. Javaux, D. and Keyser, V. D., editors. *Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD'99)*, University of Liege, Belgium, June 1999.

13. Javaux, D. and Polson, P. G. A method for predicting errors when interacting with finite state machines. In Javaux and Keyser [12]. Also to appear in *Reliability Engineering and System Safety*.

14. Johnson-Laird, P. N. *The Computer and the Mind : An Introduction to Cognitive Science*. Harvard University Press, 1989.

15. Leveson, N. G. and Palmer, E. Designing automation to reduce operator errors. In *Proceedings of the IEEE Systems, Man, and Cybernetics Conference*, October 1997. Available at `http://www.cs.washington.edu/research/projects/safety/www/papers/smc.ps`.

16. Leveson, N. G., Pinnel, L. D., Sandys, S. D., Koga, S., and Rees, J. D. Analyzing software specifications for mode confusion potential. In Johnson, C. W., editor, *Proceedings of a Workshop on Human Error and System Development*, pages 132–146, Glasgow Accident Analysis Group, technical report GAAG-TR-97-2, Glasgow, Scotland, March 1997. Paper available at `http://www.cs.washington.edu/research/projects/safety/www/papers/glascow.ps` (sic).

17. Norman, D. A. *The Psychology of Everyday Things*. Basic Books, New York, NY, 1988. Also available in paperback under the title "The Design of Everyday Things".

18. Palmer, E. "Oops, it didn't arm." A case study of two automation surprises. In Jensen, R. S. and Rakovan, L. A., editors, *Proceedings of the Eighth International Symposium on Aviation Psychology*, pages 227–232, The Aviation Psychology Laboratory, Department of Aerospace Engineering, Ohio State University, Columbus, OH, April 1995. Paper available at `http://olias.arc.nasa.gov/~ev/OSU95_Oops/PalmerOops.html`.

19. Roscoe, A. W. Model-checking CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1994.

20. Rushby, J. Combining system properties: A cautionary example and formal examination. Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Unpublished project report; included in the Murϕ distribution, and also available at `http://www.csl.sri.com/~rushby/combined.html`.

21. Sage, M. and Johnson, C. Formally verified, rapid prototyping for air traffic control. In Javaux and Keyser [12]. Also to appear in *Reliability Engineering and System Safety*.

22. Sarter, N. B. and Woods, D. D. How in the world did we ever get into that mode? Mode error and awareness in supervisory control. *Human Factors*, 37(1):5–19, 1995.

23. Sarter, N. B., Woods, D. D., and Billings, C. E. Automation surprises. In Salvendy, G., editor, *Handbook of Human Factors and Ergonomics*. John Wiley and Sons, second edition, 1997.

24. Vakil, S. S. and Hansman, R. J. Approaches to mitigating complexity-driven issues in commercial autoflight systems. In Javaux and Keyser [12]. Also to appear in *Reliability Engineering and System Safety*.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.