Electronic Communications of the EASST
Volume 45 (2011)

Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems (FMIS 2011)

Formal Modeling and Analysis for Interactive Hybrid Systems

Ellen J. Bass, Karen M. Feigh, Elsa Gunter, and John Rushby

16 pages

Guest Editors: Judy Bowen, Steve Reeves
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Formal Modeling and Analysis for Interactive Hybrid Systems

Ellen J. Bass[1], Karen M. Feigh[2], Elsa Gunter[3], and John Rushby[4]*

[1]Systems and Information Engineering, University of Virginia
[2]School of Aerospace Engineering, Georgia Institute of Technology
[3]Department of Computer Science, University of Illinois, Urbana-Champaign
[4]Computer Science Laboratory, SRI International, Menlo Park, California

Abstract: An effective strategy for discovering certain kinds of automation surprise and other problems in interactive systems is to build models of the participating (automated and human) agents and then explore all reachable states of the composed system looking for divergences between mental states and those of the automation. Various kinds of model checking provide ways to automate this approach when the agents can be modeled as discrete automata. But when some of the agents are continuous dynamical systems (e.g., airplanes), the composed model is a hybrid (i.e., mixed continuous and discrete) system and these are notoriously hard to analyze.

We describe an approach for very abstract modeling of hybrid systems using relational approximations and their automated analysis using infinite bounded model checking supported by an SMT solver. When counterexamples are found, we describe how additional constraints can be supplied to direct counterexamples toward plausible scenarios that can be confirmed in high-fidelity simulation. The approach is illustrated though application to a known (and now corrected) human-automation interaction problem in Airbus aircraft.

Keywords: Hybrid systems, infinite bounded model checking, SMT solvers, mental models, automation surprise

1 Introduction

New air traffic control procedures (collectively known as NextGen) involve automated exchange of information, intentions, and instructions among aircraft and between aircraft and the ground. These procedures raise new issues in situational awareness, autonomy, authority, and control among pilots, ground controllers, and automated systems in the cockpit and on the ground.

We participate in a project called "NextGen Authority and Autonomy" (NextGenAA) to explore human-automation interaction issues in these procedures using formal methods and simulation [BBF+11]. The ultimate goal is to provide assurance that a given procedure harbors no potential for an "automation surprise" [SWB97] or other anomaly, but along the way we will be interested to discover scenarios that may indicate such potential and to subject them to particular scrutiny.

* Supported by NSF grant CNS-0720908 and by NASA contract NNA10DE79C. The content is solely the responsibility of the authors and does not necessarily represent the official views of NSF or NASA.

Our approach to assurance is to develop models of the human and automated actors in these procedures, together with the aircraft they control or the air traffic automation they are using, and to search for anomalous scenarios, such as those that exhibit an automation surprise—which will be manifested as a divergence between the state of the mental model of one of the human actors and the real state of the physical system (or the mental state of another actor) [Rus02]. If such anomalies are found, then our models should enable us to find the root cause and to revise the associated procedure—or its supporting automation or training, as appropriate. If no anomalies are discovered, and we can show that our search is exhaustive, then we have delivered assurance for absence of anomalies in the procedures examined, subject to caveats about the soundness of our models and methods of analysis.

Many of our models will be state machines, for which several effective methods of automated analysis (i.e., search) are available: these methods use various techniques from automated deduction, such as model checking and theorem proving. State machines are suitable models for automated systems, and for some aspects of human behavior. However, we must also model the behavior of physical "plant," such as aircraft in flight. High fidelity aircraft models use differential equations that accurately represent the flight dynamics of a given aircraft. An aircraft may operate in different "modes" (e.g., flaps retracted or extended), and there will be different sets of differential equations for each mode: aircraft models are therefore hybrid systems (i.e., they combine discrete and continuous elements) and these pose challenges for analysis, especially when composed with models for the other actors.

A straightforward method for analyzing hybrid systems is simulation (e.g., using Matlab Simulink/Stateflow). Simulations can be very accurate, but they are computationally expensive and therefore of limited utility in a search for anomalous scenarios: the computational cost will restrict the search to a small number of scenarios and some anomalous ones may be overlooked. There are formal methods for exploring the reachable states of hybrid systems, and for verifying invariants, but these also are computationally challenging and can seldom handle models with more than five continuous variables.

The properties of some hybrid systems do depend crucially on the differential equations involved; control systems are like this, because the controller is highly tuned to the behavior (modeled by differential equations) of the controlled plant. But properties of other kinds of hybrid system are less dependent on the differential equations: human-automation interaction must surely be of this latter kind, because the human can employ only a fairly crude mental simulation of the dynamics involved. What we seek is a method of modeling and analysis for hybrid systems that is commensurate with this second kind of system.

Our approach is to use a range of models: we start with very abstract (i.e., highly approximate) models for aircraft dynamics that can be analyzed efficiently using a method known as "infinite bounded model checking." If anomalies are discovered, then we must decide whether these are due to the approximations employed, or are real. We attempt to do this by refining and manipulating the approximate model so that it delivers an anomalous scenario that appears credible; we then use this scenario to guide a limited search in a simulator to see if a similar anomalous scenario can be created in high fidelity.

The purpose of this paper is to describe and explain the models and the analysis techniques that we employ at the most abstracted end of our range of methods. We illustrate our techniques using a known automation surprise found in certain Airbus aircraft. Although this example does

not concern air traffic operations, it does employ similar components: human mental models, automated systems, and aircraft dynamics. We use this example because we have analyzed it previously [CJR00] and can illustrate the difference between earlier methods and those developed here.

The automation surprise that is the basis for our example is described in the following section; our method for abstract modeling and analysis is described in Section 3 and its application to the example is described in detail in Section 4; we present our conclusions in Section 5.

2 An Example Scenario: A320 Speed Protection

Our example focuses on a form of "speed protection" built in to various Airbus aircraft, and the potential it provides for an automation surprise. The specific protection that we model is an older form that was installed by default in A320, A330, and A340 aircraft and, in a somewhat similar form, in A310s. Due to automation surprise incidents, this protection is modified in the "global speed protection package" offered today. To see why, we recommend viewing the following video of an incident that occurred on 24 September 1994 to an Airbus A310, registration YR-LCC, operating as Tarom Flight 381 from Bucharest to Paris Orly: http://www.youtube.com/watch?v=VqmrRFeYzBI. The first part of the video is a reconstruction of the incident, based on information from the flight data recorder; the second part is actual video taken from the ground. The sound track from the voice data recorder is synchronized to both parts. The official incident report is available from the French authorities [BEA94]. In the following paragraphs, we adumbrate relevant parts of the automation employed in the A320 and describe a scenario that could provoke an automation surprise and an incident such as that on Tarom Flight 381.

Our automation model is based on the implementation used in A320 aircraft, as described in [CJR00]. The A320 autopilot has several vertical modes and submodes, of which we are interested in `V/S FPA` (the Flight Path Angle submode of Vertical Speed mode), `OP DES` (the Descent submode of Open mode), and `OP CLB` (the Climb submode of Open mode). `V/S FPA` flies at a specific flight path angle whose value is set in the Flight Control Unit (FCU). The autothrottle also has several modes, of which we focus on the behavior in `SPD` (Speed mode), which tries to maintain a specific airspeed whose value also is set in the FCU. If the aircraft is descending steeply, it is possible that its airspeed will exceed that requested, even when the autothrottle has selected idle thrust. In this case, FPA is prioritized over airspeed and the latter is allowed to exceed the speed set in the FCU. However, the airspeed may become sufficiently large that it exceeds the maximum safe speed, which depends on the aircraft configuration (specifically, whether the flaps are extended or not).

In this case, automated speed protection changes the vertical mode from `V/S FPA` to `OP DES` or `OP CLB`, which prioritize airspeed over vertical speed. The selected mode depends on whether the "target altitude" set in the FCU is above or below the aircraft's current altitude. During descent, the target altitude is generally set by the crew to the missed-approach altitude, in case a go-around is needed. If speed protection causes mode reversion when the plane is below this altitude, the new mode will be `OP CLB` (open climb), which will cause the plane to climb towards the altitude set in the FCU. The climb is performed at maximum thrust, using a flight path angle that will maintain the set airspeed.

This mode change is likely to occasion an automation surprise, as the mental state of the pilots will be "descent and landing" and they will not be expecting the aircraft suddenly to reverse direction from gradual descent to a strong climb. The response to such a surprise will depend on the training and performance of the crew. In the case of Flight 381, the pilots disconnected the autopilot but left the autothrottle engaged, which continued to command high thrust; they then (apparently inadvertently) commanded maximum upward pitch trim, which they countered with down elevator. When they relaxed the down elevator, the aircraft was massively out of trim and pitched violently upwards and stalled. Due to this and several similar incidents, Airbus modified speed protection so that the aircraft stays in `V/S FPA` mode, but adjusts its flight path angle to remain below the maximum permitted speed.

3 Formal Modeling Issues

One approach to formal analysis of human-automation interaction issues builds on the proposition that human interaction with machines is guided by a mental model of the device concerned [JL83]. The nature of the model is subject to debate: some authors posit stimulus-response rules, while others argue for a state machine or a similar representation that supports mental simulation of the device. Our view is that a device or system will not be useable unless interaction with it can be guided by some simple representation. We use state machines as our representation and study the interaction of such a "mental model" with a state machine model of the actual system; the hypothesis of [Rus02] is that significant divergences between the models indicate potential automation surprises, and a model checker can be used to detect such divergences and to construct scenarios that manifest them.

Now, how can we develop the state machine for a mental model? We are not seeking to develop or validate psychological insight, so we do not attempt to discover the models employed by individual human operators; rather, we suppose that the system developer explicitly or implicitly designs the system so that it can be operated with the aid of a relatively simple model, and part of the purpose of the system training manuals is to communicate that model. Javaux [Jav98] suggests that training induces fairly detailed and precise state machine models, which are then simplified through experience and forgetfulness. He proposes (and has validated) two specific processes: *frequential simplification* causes rarely taken transitions, or rarely encountered guards on transitions, to be forgotten, while *inferential simplification* causes transition rules that are "similar" to one another to be merged into a single prototypical rule that blurs their differences.

Plausible mental models could conceivably be constructed mechanically using Javaux' insights: extract the state machine implied by the training manuals, then iteratively apply the two simplification rules until a fixed point is obtained. However, our previous examples, [Rus02] (MD-88), [CJR00] (A320), and [Rus00] (737) use very crude mental models that represent little more than plausible expectations about vertical direction (i.e., whether the pilot expects the aircraft to climb or to descend), and the representation of the aircraft automation is also highly abstracted. A divergence is considered to occur when the vertical direction of the mental model is opposite to that commanded by the automation.

A valid objection to the modeling used in these examples is that no aircraft is present! An automation surprise is modeled as a divergence between the state of the mental model and that of

the automation but, in reality, the state of the automation is not always known to the pilot (that is often the root cause of the automation surprise); a true automation surprise is surely a divergence between the state of the mental model and the actual behavior of the aircraft—as observed via its instruments or through the seat of the pants.

To model and analyze this more realistic interpretation, we need to introduce the dynamics of the aircraft, and the ways in which these are controlled by the automation. As noted earlier, this takes us into the realm of hybrid systems where, despite much impressive research, the problems of model checking and verification remain computationally formidable. Our approach is to remain very abstract (i.e., approximate) but to introduce "just enough" modeling of the dynamics that we are able to detect anomalies very efficiently and can generate scenarios that may be used to guide high-fidelity simulations.

For the A320 speed protection example, we need to model how the thrust and pitch values computed by the control laws of the autothrottle and autopilot, and the flap setting commanded by the pilots, determine the trajectory of the airplane. We are interested only in the vertical dimension, so we focus on altitude and airspeed and the derivatives of these. Now it might seem that we need differential equations to model these attributes—and indeed, we do for full fidelity. But for our purposes, something much cruder suffices: we can ignore the metric element of time and simply assert relationships among the continuous state variables "now" and those in *any* future state, provided the airplane stays in the same discrete mode. For example, if the pitch angle is positive, we can assert that the altitude any time in the future will be greater than it is now, provided there is no change in discrete parameters (such as the flaps setting, or the pitch angle itself).

There is, in fact, a very strong and sound new method for analysis of hybrid systems based on relational abstractions of this type [ST11]. The difference between that approach and ours is that sound relational abstractions are calculated from a hybrid system model by specialized invariant generation, whereas we simply assert a relation as our model. We claim that this is sufficient for our purposes: provided our relations are conservative (i.e., admit *more* behaviors than would an accurate model), we are certain to discover anomalies if they exist. Of course, we may discover spurious anomalies if our modeling is too approximate. What we will do is construct very approximate models to begin with, then, if we discover an interestingly anomalous scenario (e.g., one in which the pilot's mental mode is "descend" but airplane is climbing), we can refine the model until the scenario becomes realistic or is found to dissolve as an artifact of excessive approximation. Once we have developed a realistic scenario with the model checker, we will attempt to reproduce it in a high-fidelity simulation.

Relational approximations allow us to eliminate derivatives and differential equations but we must still consider how to represent the relations in a form that a model checker can accept and analyze. Model checkers differ greatly in the ways models may be specified, and in the kinds of analysis they support. Some model checkers employ a modeling language similar, or even identical, to a programming language, while others provide a more abstract notation. Among the latter, "guarded commands" are a popular choice. These consist of a series of commands of the following form

```
guard --> var = expression
```

where the `guard` is some predicate over the state variables, `var` is one of the state variables,

and `expression` is some expression over the state variables. The interpretation is that in a state where the `guard` evaluates to true, the state variable `var` can be updated to the result of evaluating `expression` in that state. It usually is possible to have multiple assignments, so that several state variables can be updated atomically. If more than one guard evaluates to true, then one is chosen nondeterministically. Most model checkers also allow nondeterministic assignment so that a set of expressions may appear on the right, and one is chosen nondeterministically. It is sometimes useful if the value newly assigned to one state variable can be used in the expression assigned to a second. Primes are often used for this purpose, so that the example above would become

```
guard --> var' = expression
```

and primed variables may then appear in the `expression` (some analysis may be necessary to eliminate circular dependencies). Some model checkers further allow primed variables to appear in the `guard`.

With this background, we now describe three methods for specifying relational models. To make it concrete, we suppose we are trying to specify a model in which altitude must increase when pitch angle is positive, and we will use the concrete syntax of the SAL suite of model checkers from SRI [SAL]. Our first method is the following.

```
pitch > 0 --> altitude' IN {x | x > altitude}
```
1

Here `pitch` and `altitude` are state variables of some numeric type; the `IN` construct is SAL's notation for nondeterministic assignment, and the relation appears directly within this construct: it says that the new value of `altitude` is chosen nondeterministically from those values greater than its current value.

A second method is the following.

```
pitch > 0 AND (altitude' > altitude) --> altitude' IN {x | TRUE}
```
2

Here, the assignment is totally nondeterministic and the relation appears in the guard. Many model checkers do not allow primed variables in the guards and so they cannot support this method.

The third method employs a *synchronous observer*. This is a separate model, synchronously composed with the first, that observes the state of the system and sets a new Boolean state variable `ok` to `FALSE` when it observes a violation of the relational constraint. For our simple example, the basic model will use a totally nondeterministic assignment

```
pitch > 0 --> altitude' IN {x | TRUE}
```
3

and the observer will enforce the relation as follows.

```
NOT (altitude' > altitude) --> ok' = FALSE
```
4

When we run the model checker, we will instruct it to consider only those runs where `ok` is true. For example, when model checking for some invariant property `prop`, we will use a specification of the following form, where `G` (sometimes written $\Box$) is the *always* modality of Linear Temporal Logic (LTL).[1]

[1] A model checker using Computation Tree Logic (CTL) would use `AG` in place of `G`.

```
G(ok IMPLIES prop)
```

While the first method seems the most attractive in this simple example, it does not extend to the case where assignments are made to several state variables and the new values must satisfy some joint relation (e.g., assign to `x` and `y` such that `x*x + y*y < 1`). The second method can deal with this case but, as already noted, many model checkers do not allow primed variables in the guards. Hence, for widest applicability, we will employ the third method; we will see later that use of a synchronous observer also makes it easy to refine the relation (by simply adding more constraints). Primed variables in the guards of a synchronous observer can be eliminated, if necessary, by introducing additional variables to store the previous values of the state variables concerned (so the model operates on values delayed by one time unit: "previous" and "now," rather than "now" and "next.")

We now turn to the representation of numerical values for quantities such as `altitude` and `pitch`. A crude approximation simply uses a discrete enumeration to represent ranges of numeric values (e.g., `low`, `medium`, and `high` for `altitude`). However, most modern model checkers directly support the use of bounded integers (e.g., in the range -32,786 to 32,767) by encoding them as bitvectors. Model checking is accomplished by translating the model to a purely Boolean representation that is analyzed by a BDD or SAT engine, so operations such as addition are performed by compiling the representation for a binary adder into the translation sent to the backend engine.[2]

The bitvector representation is adequate for many examples, but it is computationally expensive so that models with many numerical state variables can become difficult to analyze. Furthermore, the "natural" representation for quantities such as pitch and altitude is surely as real numbers, by which we mean the mathematical notion of real numbers, not approximations such as floating point.[3] Fortunately, there is a technology that supports this representation, and also mathematical (i.e., unbounded) integers; this is the technology of "infinite bounded model checking" [MRS02]. In its basic form, *Bounded Model Checking* (BMC) takes a finite state model of a system, a putative invariant for the system (i.e., a property that should be true in all its reachable states), and a natural number k, and determines whether there is a counterexample to the property of length k steps or less. The finite state model is translated to a Boolean representation and "unrolled" k times, the property is likewise represented in Boolean form, and these two Boolean representations are combined to pose a problem that can be solved by a propositional satisfiability (SAT) solver: BMC can be extended from refutation (i.e., bug-finding) to verification by slightly reformulating the underlying SAT problem so that it performs k-induction [MRS03].

SAT solving can be generalized from the purely propositional case to Satisfiability Modulo Theories (SMT), which supports several useful theories, including the real numbers, mathematical (i.e., unbounded) integers, and uninterpreted functions [Rus06]. BMC and k-induction can then be reformulated to target the capabilities of SMT solvers; this is referred to as *infinite* BMC, because it operates over potentially infinite state spaces, such as those involving real numbers. In additional to supporting more realistic models, infinite BMC over reals and integers is often

[2] The different traditions and technologies that contribute to model checking use different terms for the same notions—bitvectors, and binary, Boolean or propositional representations—they are all equivalent for our purposes.

[3] Cockpit instruments may display altitude in terms of integer feet, or flight level, but the underlying physical parameter—the actual height of the airplane above the ground—is a real-valued quantity.

faster than ordinary BMC using bitvector representations for bounded integers.

SMT solvers, and hence infinite BMC, allow the use of *uninterpreted* functions: these are functions about which nothing is known, save what is supplied via axioms. The attraction here is that the totally nondeterministic assignment in 3 can be replaced by one that indicates the state variables on which this assignment should depend. This does not change the reachable states of the model (those are determined by the relational constraint enforced by the synchronous observer of 4) but it conveys more of the intuition behind the model, and this can help in communicating with those who develop the more detailed simulation models. Specifically, we can introduce an uninterpreted relation (which can be thought of as an uninterpreted *higher-order* function that returns a *set* of values) `ad` (standing for altitude dynamics) that takes an altitude and a pitch angle and returns the set of possible new altitudes. This relation would be defined in SAL like this

```
ad(alt: REAL, pitch: REAL): [REAL -> BOOLEAN]
```

(so that `ad(x, y)(z)` is `TRUE` if `z` is among the possible new altitudes when the current altitude and pitch are `x` and `y`, respectively) and the guarded command of 3 can then be written as follows.

```
pitch > 0 --> altitude' IN ad(altitude, pitch)
```

Now that we have introduced our approach to relational modeling of interactive hybrid systems, and methods for representing and analyzing these with a model checker, we are ready to illustrate the approach using the A320 speed protection example introduced previously; this is the topic of the following section.

4 Example: Formal Analysis for A320 Speed Protection

The behavior of the A320 and its speed protection system, as described in section 2, emerges from the interaction of several separate components: the pilots and the devices they manipulate (FCU, sidestick etc.), the autopilot and autothrottle, the engines, and the dynamics of the aircraft. Since the purpose of this example is illustration, we will omit many details and lump some of the components together: in particular, we will combine the autopilot and autothrottle into one component called `automation`, we will combine all aspects of human-automation interaction into a component called `pilots`, and we will combine the dynamics of the aircraft with its engines into another component called `airplane`. As we are interested only in the behavior of the aircraft in the vertical direction, we model just its altitude and airspeed, and ignore its heading and horizontal position.

We first outline each of the three lumped component models, then present their actual specification. We begin with the `automation`. This is a fairly conventional state machine: it takes as inputs various controls and values set by the `pilots` (desired vertical mode, FCU altitude and flight path angle, flap setting) and the current state of the airplane (its airspeed and altitude), determines the actual vertical mode to be used (which may be different than that desired by the `pilots` if a protection is being applied), and applies control laws to determine the thrust and

pitch settings to be used by the airplane. Notice that thrust and pitch are modeled as parameters to internal communications from automation to airplane; they are not observed by the pilots directly.

The pilots take the state of the airplane (airspeed and altitude) as inputs (given by instruments and, presumably, their own kinesthetics), and perform various actions such as dialing values for altitude and flight path angle into the FCU, setting the desired vertical mode, and extending or retracting the flaps. They perform these actions in the context of a "mental mode" (descending, climbing, level flight) that provides coherence: for example, they will not extend the flaps when the mental mode is "climbing." This behavior can be modeled by a conventional state machine.

The airplane takes as input the thrust and pitch values computed by the control laws of the automation, and the flap setting commanded by the pilots, and simulates the aircraft dynamics to calculate its trajectory through space, of which we model only altitude and airspeed.

Having described the general approach, we now present the example concretely using the notation of SAL.

```
a320sp: CONTEXT =
BEGIN

flap_config:   TYPE = {retracted, extended};
vertical_mode: TYPE = {vs_fpa, op_clb, op_des, other};
mental_modes:  TYPE = {climb, descend, level};

speedvals:  TYPE = {x: REAL | x >= 0 AND x < 700};
altvals:    TYPE = {x: REAL | x >= 0 AND x < 43000};
thrustvals: TYPE = {x: REAL | x >= 0 AND x <= 100};
pitchvals:  TYPE = {x: REAL | x >= -9 AND x <= 30};

VMAX: speedvals = 400;    Vfe: speedvals = 180;
```

The specification begins with its name a320sp ("sp" for speed protection) and is kept in a file named a320sp.sal. Next, we introduce types for some of the state variables. First, flap_config is an enumerated type used to specify flap configurations (we abstract all degrees of extension into the single value extended), then vertical_mode enumerates the various modes of the automation: we focus on vs_fpa (V/S FPA), op_clb (OP CLB) and op_des (OP DES), and abstract all others into other. The mental_modes of the pilots are also enumerated here. Next, we introduce the types that will represent airspeeds, altitudes, thrust settings, and pitch: these are modeled as suitable subranges of the real numbers. We also specify constants for the maximum speeds permitted for the airplane: Vfe is the maximum when the flaps are extended, and VMAX when they are retracted. These and other numeric constants appearing in the specification were chosen somewhat arbitrarily: we do not know the true values.

Then we introduce the uninterpreted functions (they are actually relations but, as noted before, SAL models these as higher-order functions whose range type is a predicate) that describe the dynamics of the airplane: there are two pairs of functions, one giving the airspeed dynamics, the other the altitude, each in two variants, depending on whether the flaps are extended or retracted (i.e., the wing is "clean"). These functions take the current airplane airspeed, altitude, engine thrust, and pitch angle and deliver sets (modeled as predicates—i.e., functions with

range type BOOLEAN) of airspeed or altitude as appropriate. To avoid cluttering the page, we replace function arguments with ... after the first.

```
speed_dynamics_clean(airspeed: speedvals, altitude: altvals,
   thrust: thrustvals,  pitch: pitchvals): [speedvals -> BOOLEAN];
alt_dynamics_clean(...):    [altvals -> BOOLEAN];
speed_dynamics_flaps(...): [speedvals -> BOOLEAN];
alt_dynamics_flaps(...):    [altvals -> BOOLEAN];
```

Next, we introduce uninterpreted functions that represent the control laws applied by the automation. These also are specified in pairs: one member of each pair computes the engine thrust to be applied, the other the pitch angle to be flown. The functions take as arguments the current airspeed, the flight path angle set in the FCU (fcu_fpa), the current altitude, the target altitude set in the FCU (fcu_alt), the current pitch angle and the flaps setting. There is a separate pair of control law functions for each mode of the automation: vs_fpa, op_clb, and op_des. To save space, we omit the thrust control law functions.

```
vs_fpa_pitch_law(airspeed: speedvals, fcu_fpa: pitchvals,
   altitude: altvals, fcu_alt: altvals, pitch: pitchvals,
   flaps: flap_config): [pitchvals -> BOOLEAN];
op_clb_pitch_law(...):  [pitchvals -> BOOLEAN];
op_des_pitch_law(...):  [pitchvals -> BOOLEAN];
```

Now we can specify the automation as a state machine. This takes the current flaps setting, fcu_alt, fcu_fpa, and fcu_mode (all set by the pilots), and the current airspeed and altitude (set by the airplane), and outputs the thrust and pitch angle settings, and also the actual_mode whose control laws it is applying.

```
automation: MODULE =
BEGIN
 INPUT
  flaps:    flap_config,      fcu_alt:   altvals,     fcu_fpa:  pitchvals,
  fcu_mode: vertical_mode,    airspeed:  speedvals,   altitude: altvals
 OUTPUT
  thrust:      thrustvals,    pitch:     pitchvals,
  actual_mode: vertical_mode, max_speed: speedvals
 INITIALIZATION
  actual_mode = fcu_mode
 DEFINITION
  max_speed = IF flaps = retracted THEN VMAX ELSE Vfe ENDIF;
 TRANSITION
[  track-fcu-mode: fcu_mode' /= fcu_mode --> actual_mode' = fcu_mode'
[] mode_reversion: actual_mode = vs_fpa AND airspeed > max_speed -->
    actual_mode' = IF fcu_alt > altitude THEN op_clb ELSE op_des ENDIF;
[] vs_fpa_mode: actual_mode = vs_fpa AND airspeed <= max_speed -->
    pitch' IN vs_fpa_pitch_law(...)
[] op_clb_mode: actual_mode = op_clb --> pitch' IN op_clb_pitch_law(...)
[] op_des_mode: actual_mode = op_des --> pitch' IN op_des_pitch_law(...)
[] automation_idles: ELSE -->
] END;
```

The guarded commands appearing in the TRANSITION section and separated by [] symbols specify the behavior of the state machine: each command has a label (whose only purpose is to provide identifying information in counterexamples) followed by a colon, then the guard (a Boolean expression), then an arrow (-->) followed by a series of assignments. At each step, some true guard is selected (the ELSE guard will be true if none of the others are) and the corresponding assignments are executed atomically.

Recall that primed names represent the value of the state variable in the "new" state while unprimed values represent the "old" value, and notice that primed names can appear in guards and on the right side of assignments. Thus, the guard of the command track_fcu_mode is true when the pilots have *changed* the setting of fcu_mode, and the result of the assignment is to set the actual_mode equal to the new value of the fcu_mode.

The guard of the command mode_reversion is true when the actual_mode is vs_fpa and the airspeed exceeds the current maximum. The result of the corresponding assignment is to set the actual_mode to op_clb or op_des as appropriate. The remaining guarded commands simply apply the control laws of the current actual_mode. To save space, we omit all assignments to the thrust variable (these would be modeled as application of the thrust control laws of the autothrottle).

Next, we specify the behavior of the pilots. Essentially, this is a nondeterministic choice among extending or retracting the flaps (only when the mental_mode is descend or climb, respectively), dialing a (nondeterministic) value into fcu_alt, switching the mental mode to descend or climb, or doing nothing. When the mental_mode switches to descend or climb, the fcu_mode is set to vs_fpa and a nondeterministic negative or positive flight path angle, respectively, is dialed into fcu_fpa.

```
pilots: MODULE =
BEGIN
 OUTPUT
  flaps:    flap_config,   fcu_alt:     altvals,   fcu_fpa: pitchvals,
  fcu_mode: vertical_mode, mental_mode: mental_modes
 INPUT
  airspeed:    speedvals,    altitude:    altvals
 INITIALIZATION
  mental_mode = level;  fcu_mode = other;   fcu_alt = 0;  flaps = retracted;
 TRANSITION
[  extend_flaps: mental_mode = descend and flaps = retracted -->
    flaps' = extended
[] retract_flaps: mental_mode = climb and flaps = extended -->
    flaps' = retracted
[] dial_fcu_alt: fcu_mode = other --> fcu_alt' IN {x: altvals | TRUE}
[] dial_descend: mental_mode /= descend -->
    mental_mode' = descend;     fcu_mode' = vs_fpa;
    fcu_fpa' IN {x: pitchvals | x < 0};
[] dial_climb: mental_mode /= climb -->
    mental_mode' = climb;    fcu_mode' = vs_fpa;
    fcu_fpa' IN {x: pitchvals | x > 0};
[] pilots_idle: TRUE -->
] END;
```

Next, we present the model of the airplane. This has two modes, depending on whether

or not the flaps are extended, and simply applies the appropriate speed and altitude dynamics to yield (nondeterministic) new values for its output variables `airspeed` and `altitude`. We initialize these variables to values representative of the later stages of a descent.

```
airplane: MODULE =
BEGIN
 INPUT
  thrust: thrustvals,   pitch: pitchvals,   flaps: flap_config
 OUTPUT
  airspeed: speedvals,  altitude: altvals
 INITIALIZATION
  airspeed = 200;       altitude = 3000;
 TRANSITION
[  flying_clean: flaps = retracted -->
    airspeed' IN
      speed_dynamics_clean(airspeed, altitude, thrust, pitch);
    altitude' IN alt_dynamics_clean(...);
[] flying_flaps: flaps = extended -->
    airspeed' IN speed_dynamics_flaps(...);
    altitude' IN alt_dynamics_flaps(...);
] END;
```

We now need to specify a `constraints` module that enforces suitable relations on `altitude` and `airspeed` (thereby giving more interpretation to the `airplane` model) and on `thrust` and `pitch` (thereby giving more interpretation to the `automation`). This module will be a *synchronous observer* that takes many of the state variables of the system as inputs, and sets a variable `ok` to false whenever they violate the desired constraints. When we use model checking to examine properties of interest, we will restrict attention to those scenarios in which `ok` is true.

```
constraints: MODULE =
BEGIN
 OUTPUT
  ok: BOOLEAN
 INPUT
  airspeed:    speedvals,      altitude:    altvals,
  thrust:      thrustvals,     pitch:       pitchvals,
  actual_mode: vertical_mode,  flaps:       flap_config,
  fcu_alt:     altvals,        fcu_fpa:     pitchvals,
  fcu_mode:    vertical_mode,  mental_mode: mental_modes
 INITIALIZATION
   ok = TRUE;
 TRANSITION
[  actual_mode = op_des AND pitch > 0 --> ok' = FALSE;
[] actual_mode = op_clb AND pitch < 0 --> ok' = FALSE;
[] actual_mode = vs_fpa AND fcu_fpa <= 0 AND pitch > 0 --> ok' = FALSE;
[] actual_mode = vs_fpa AND fcu_fpa >= 0 AND pitch < 0 --> ok' = FALSE;
[] pitch > 0 AND altitude' < altitude --> ok' = FALSE;
[] pitch < 0 AND altitude' > altitude --> ok' = FALSE;
[] pitch=0 AND altitude' /= altitude --> ok' = FALSE;
[] ELSE -->
] END;
```

The first guarded command ensures that when the actual_mode is op_des, then the pitch angle is not positive (the guarded commands specify what is *not* allowed, so the constraints are the negation of these); it is a constraint on the otherwise uninterpreted op_des_pitch_law of the automation. The next three guarded commands similarly add constraints to the other control laws. The fourth guarded command ensures that the altitude increases when the pitch angle is positive and the remaining commands deal with the cases of negative or zero pitch angle; these three commands are constraints on the airplane model.

Finally, we can specify the property we wish to examine: that is, an automation surprise where the mental_mode of the pilots is descend but the airplane is (strongly) climbing. We specify this in another synchronous observer that raises (and latches) an alarm variable when it sees a violation of the desired property: altitude' - altitude > 0 indicates a climb and we (somewhat arbitrarily) substitute 90 for 0 as the indication of a strong climb.

```
observer: MODULE =
BEGIN
 OUTPUT
  alarm: BOOLEAN
 INPUT
  mental_mode: mental_modes,   altitude:    altvals
 INITIALIZATION
   alarm = FALSE
 TRANSITION
   alarm' = alarm OR (mental_mode = descend AND altitude' - altitude > 90)
END;
```

We specify the system as the synchronous composition of the five modules introduced above (in this kind of composition, a step of the system comprises a step by each of its components).

```
system: MODULE = airplane || automation || pilots || constraints || observer;
```

Then we specify (as the theorem surprise) the invariant that the alarm is never raised, provided the constraints are satisfied (i.e., ok is true); any counterexample to this invariant will be an anomalous scenario that manifests an automation surprise.

```
surprise: THEOREM system |- G(ok IMPLIES NOT alarm);
```

We use infinite BMC to examine this claim.

```
sal-inf-bmc a320sp.sal surprise -v 3 -it -d 20
```

This command names the claim surprise to be examined in the file a320sp.sal, sets the verbosity level to 3, and instructs the model checker to iteratively increase the number of steps in the examination from 1, 2, ... to a maximum depth of 20 (the default is 10).

The model checker discovers a violation of the property in five steps in a fraction of a second. The scenario is summarized below (we use simple Unix scripts to format these tables from the raw SAL output). The numbered rows give the values of the state variables (we abbreviate names and omit several variables to save space) and the intervening unnumbered rows name the commands of the airplane, automation, and pilots modules, respectively, that produced the transition from the state above to the one below.

(To conserve space, we omit the first step, in which `fcu_alt` is set by `dial_fcu_alt`.)

step	act_mde	airspd	alt	fcu_alt	fcu_fpa	fcu_md	flaps	mx_spd	mntl_md	pitch
1	other	200	3000	3001	-1	other	rtrctd	400	level	0
	Commands: flying_clean, track_fcu_md, dial_descend									
2	vs_fpa	401	3000	3001	-2	vs_fpa	rtrctd	400	descend	0
	Commands: flying_clean, mode_reversion, extend_flaps									
3	op_clb	180	3000	3001	-2	vs_fpa	extnd	180	descend	0
	Commands: flying_flaps, op_clb_mode, pilots_idle									
4	op_clb	0	3000	3001	-2	vs_fpa	extnd	180	descend	1
	Commands: flying_flaps, op_clb_mode, pilots_idle									
5	op_clb	0	3091	3001	-2	vs_fpa	extnd	180	descend	0

We see that a mode reversion has occurred, causing a climb while the `mental_mode` is `descend`, but it is caused by the `airspeed` abruptly increasing from 200 to 401 (thereby exceeding `VMAX`). We observe some other unfortunate attributes in this counterexample: for example, in steps 4 and 5 the airspeed decays to 0.

These abrupt changes in `airspeed` are contrary to our intuition, but consistent with our specification because we have no constraints on the uninterpreted functions `speed_dynamics_clean` and `speed_dynamics_flaps` that represent the dynamics of this variable. SAL and its backend SMT solver Yices [SAL] find "simple" satisfying instantiations for uninterpreted functions: there is nothing that requires these functions to be "continuous," so the solver just finds values for the points needed to construct the counterexample. We need to add further constraints to our model and to refine some of those already present so that they more accurately represent the dynamics.

```
[] airspeed' > airspeed+10 OR airspeed' < airspeed-10 --> ok' = FALSE;
[] pitch > 0 AND altitude' < altitude+10*pitch --> ok' = FALSE;
[] pitch < 0 AND altitude' > altitude+10*pitch --> ok' = FALSE;
[] pitch=0 AND
 (altitude' > altitude+10 OR altitude' < altitude-10) --> ok' = FALSE;
```

The first of these guarded commands is a new rule that requires `airspeed` to change by no more than 10 between steps. The next three commands are refinements of those already present: they couple `altitude` more tightly to `pitch`. With these adjustments to the `constraints` module, we invoke the model checker again, and receive the following counterexample.

step	act_mde	airspd	alt	fcu_alt	fcu_fpa	fcu_md	flaps	mx_spd	mntl_md	pitch
1	other	200	3000	3291	-1/50	other	rtrctd	400	level	-1/100
	Commands: flying_clean, track_fcu_md, dial_descend									
2	vs_fpa	201	2989	3291	-1/100	vs_fpa	rtrctd	400	descend	-1/100
	Commands: flying_clean, vs_fpa_mode, extend_flaps									
3	vs_fpa	200	2988	3291	-1/100	vs_fpa	extnd	180	descend	0
	Commands: flying_flaps, mode_reversion, pilots_idle									
4	op_clb	201	2989	3291	-1/100	vs_fpa	extnd	180	descend	0
	Commands: flying_flaps, op_clb_mode, pilots_idle									
5	op_clb	200	2990	3291	-1/100	vs_fpa	extnd	180	descend	1/50
	Commands: flying_flaps, op_clb_mode, pilots_idle									
6	op_clb	190	3291	3291	-1/100	vs_fpa	extnd	180	descend	3/100

This counterexample manifests the automation surprise from Section 2, and does so with a plausible scenario: the `fcu_alt` is set to 3291 while the aircraft is flying at 3000; the `pilots` decide to `descend` and enter a negative `fcu_fpa`; they then extend the `flaps`, which causes overspeed and a mode reversion to `op_clb` mode, which in turn causes a strong climb.

Some infelicities remain in the scenario: for example, the values for `pitch` and `fcu_fpa` are implausible. These can be adjusted by adding additional commands to the `constraints` module. Although formally equivalent, there is a conceptual distinction between constraints that truly refine the model and those that serve merely to nudge the counterexample in a preferred direction; if desired, the latter can be placed in a separate synchronous observer module. Although we have not developed this example beyond what is described here, we are confident (because of the strength of abstraction available) that the approach can be applied to more detailed and realistic specifications of aircraft automation and human mental models.

5 Conclusions

We have described and illustrated a method for modeling and analyzing interactive hybrid systems at a very abstract level. This enables us to model human-machine interactions in the presence of physical plant, such as airplanes, whose dynamics are described by differential equations. The method uses relational abstractions for the hybrid components: these are approximations that specify relations between the continuous state variables "now" and any future state, provided the discrete variables remain the same. We described how relations may be specified to a conventional model checker using synchronous observers. Automated analysis of our approximations builds on the ability SMT solvers to reason over the theories of uninterpreted functions and arithmetic, and the exploitation of this ability by infinite bounded model checkers.

Of course, any discrete approximation for the hybrid components enables model checking of interactive hybrid systems; the arguments in favor of our method are that we can be reasonably confident of soundness, it is easy to refine the models, and it is easy to analyze them. For soundness (with respect to safety properties) it is necessary that the behaviors of our relational approximations are a superset of those of a fully accurate model (i.e., with differential equations). It is possible to derive such sound approximations by analysis of the accurate model [ST11] but this is computationally challenging. What we do instead is assert simple relations in whose soundness we are confident (e.g., when pitch angle is positive, altitude increases).[4] If model checking reveals a potential automation surprise, then it is easy to refine the approximation by adding additional relational constraints to the synchronous observer so that a realistic scenario is developed, or the anomaly is found to be due to excessive approximation.

This approximation method is suitable for examining new air traffic procedures for human-computer interaction issues; when a possible problem is identified, additional constraints can be used to push the counterexample toward a plausible scenario that can be examined in a higher fidelity modeling environment, such as a simulator. The NextGenAA project plans to evaluate this method on new procedures such as Continuous Descent Approach (CDA) and Oceanic Airspace In-Trail Procedure (ATSA ITP) [BBF+11]. Verifying safety, or finding anomalies, in complex interactions such as these, involving humans, automation, and hybrid/dynamic systems, requires very strong, but appropriate use of abstraction in modeling. The method introduced in this paper

[4] This might not be true in certain conditions, such as downdrafts; that is outside our model, but could be added.

adds a new approach to the collection of highly abstract modeling and analysis methods available for this and similar domains.

Acknowledgements: Ashish Tiwari introduced us to relational abstractions, and to some of the ways to represent them in SAL.

Bibliography

[BBF+11] E. J. Bass, M. L. Bolton, K. M. Feigh, D. Griffith, E. Gunter, W. Mansky, J. Rushby. Toward a Multi-Method Approach to Formalizing Human-Automation Interaction and Human-Human Communications. In *IEEE International Conference on Systems, Man, and Cybernetics*. Anchorage, AK, Oct. 2011. To appear.

[BEA94] Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile. Report on the incident on 24 September 1994 during approach to Orly (94) to the Airbus A 310 registered YR-LCA operated by TAROM. 1994. Report YR-A940924A.

[CJR00] J. Crow, D. Javaux, J. Rushby. Models and Mechanized Methods that Integrate Human Factors into Automation Design. In Abbott et al. (eds.), *International Conference on Human-Computer Interaction in Aeronautics: HCI-Aero 2000*, pp. 163–168. Toulouse, France, Sept. 2000.

[Jav98] D. Javaux. Explaining Sarter and Woods' Classical Results. In Leveson and Johnson (eds.), *2nd W'kshop on Human Error, Safety, and S/W Design*. Seattle, WA, Apr. 1998.

[JL83] P. N. Johnson-Laird. *Mental Models*. Cognitive Science Series 6. Harvard University Press, Cambridge, MA, 1983.

[MRS02] L. de Moura, H. Rueß, M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In Voronkov (ed.), *18th International Conference on Automated Deduction*. LNCS 2392, pp. 438–455. Copenhagen, Denmark, July 2002.

[MRS03] L. de Moura, H. Rueß, M. Sorea. Bounded Model Checking and Induction: From Refutation to Verification. In Hunt, Jr. and Somenzi (eds.), *Computer-Aided Verification*. LNCS 2725, pp. 14–26. Boulder, CO, July 2003.

[Rus00] J. Rushby. Analyzing Cockpit Interfaces Using Formal Methods. In Bowman (ed.), *Proceedings of FM-Elsewhere*. Electronic Notes in Theoretical Computer Science 43. Elsevier, Pisa, Italy, Oct. 2000.

[Rus02] J. Rushby. Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises. *Reliability Eng. and System Safety* 75(2):167–177, Feb. 2002.

[Rus06] J. Rushby. Harnessing Disruptive Innovation in Formal Verification. In Hung and Pandya (eds.), *Fourth International Conference on Software Engineering and Formal Methods*, pp. 21–28. Pune, India, Sept. 2006.

[SAL] SAL and Yices home pages. http://sal.csl.sri.com/ and http://yices.csl.sri.com/.

[ST11] S. Sankaranarayanan, A. Tiwari. Relational Abstractions for Continuous and Hybrid Systems. In *Computer-Aided Verification*. LNCS. Snowbird, UT, 2011. To appear.

[SWB97] N. B. Sarter, D. D. Woods, C. E. Billings. Automation Surprises. In Salvendy (ed.), *Handbook of Human Factors and Ergonomics*. Wiley and Sons, 2nd edition, 1997.