From Interactors to SMV: A Case Study in the Automated Analysis of Interactive Systems

José C. Campos* and Michael D. Harrison
The University of York

July 6, 1999

Abstract

Recent accounts of accidents have drawn attention to problems that arise in safety critical systems through "automation surprises". A particular class of such surprises, interface mode changes, may have significant impact on the safety of a dynamic interactive system and may take place *implicitly* as a result of other system action. Formal specifications of interactive systems may provide an opportunity to analyse problems that arise in such systems. In this paper we consider the role that an interactor based specification may have as a partial model of an interactive system, so that mode consequences might be checked early in the design process. We show how interactor specifications can be translated into the SMV model checker input language, and how we can use such specifications in conjunction with the model checker to analyse potential for mode confusion in a realistic case. Our final aim is to develop a general purpose methodology for the automated analysis of interactive systems, and, in this context, we show how the verification process can be useful by raising questions that have to be addressed in a broader context of analysis.

1 Introduction

This paper responds to two issues. The first is that recent accounts of accidents, incidents and simulations (Palmer 1995) have drawn attention to problems that arise in safety critical systems through "automation surprises". A particular class of such surprises, interface mode changes, may have significant impact on the safety of a dynamic interactive system and may take place *implicitly* as a result of other system action. The second is that the formal specification of an interactive system may provide an opportunity to analyse the consequences of such a design and thereby reduce the risk of this type of interface problem.

We note recent relevant analyses (Leveson & Palmer 1997, Rushby 1999) of mode complexity in aviation based systems. These analyses have been conducted retrospectively using experience based on flight simulations. Scenarios have provided the foundation for the analysis. In this paper we consider the role that an interactor based specification (Faconti & Paternò 1990, Duke &

*José C. Campos is supported by Fundação para a Ciência e a Tecnologia (FCT, Portugal) under grant PRAXIS XXI/BD/9562/96.

Harrison 1993) may have as a partial model of an interactive system so that mode consequences might be checked early in the design process. An *Interactor* is an object (consisting of state and operations) with the additional property that state that is perceivable to the user, and actions that are accessible to the user, are identified explicitly. This notion will be developed further in Section 2.2.

What makes interactive systems interesting (and hard) from the point of view of verification, is the multiplicity of areas and concerns that come into play during the design of such systems. To the traditional concerns of software engineering, interactive systems design adds a requirement to accommodate a consideration of the context in which the system is used. This means that aspects of psychology, sociology, and ergonomics, may all have a bearing on design, and may need to be taken into account during verification.

General concepts of usability derived from psychological or sociological understandings are difficult if not impossible to express in a form that can be used as part of a verification process. Even more concrete concepts such as task and mode involve a broad range of concerns, from hardware restrictions to more subjective human factors issues. In (Campos & Harrison 1998) we argue that to address these questions effectively a tighter integration between design and verification is required, and that this integration can be achieved by developing, and verifying, a range of partial models of the system under development, each model focusing on a specific set of features of the system.

In (Doherty, Campos & Harrison 1998) it was shown how theorem proving can be used to analyse representational issues of interfaces. Palmer (1995) reports on problems found during a set of simulations of realistic flight missions. One of these was related to the task of climbing and maintaining altitude in response to Air Traffic Control instructions. A change in the flying mode, performed by the autopilot without intervention of the pilot, caused pilot action to cancel the capture on the desired altitude inadvertently. This situation has implications for the safety of the airplane as it can result in an "altitude bust" and consequent air traffic problems of loss of separation with other aircraft. We will show how checking specifications using a model incorporating the interface between the pilot and the automation, may detect problems such as these in early stages of design.

We will be using interactors to write our models, and the SMV model checker (McMillan 1992) to perform their verification. We start by writing an interactor based specification of the system under analysis. We then determine what context of operation and what properties we are interested in, and write those properties as CTL (Computational Tree Logic) (Clarke, Emerson & Sistla 1986) formulae. Finally we translate the specification into SMV and model check it. The last two steps are automated. If the property is false we will (hopefully) get a counter example and we can re-evaluate the specification and repeat the process.

In Section 2 we discuss the role verification should play during design, and introduce the interactor language used in the following sections. In Section 3 we use the interactor language to build a model of the Mode Control Panel (MCP) of the airplane. The MCP is one element of the interface between the pilot and the aircraft's autopilot. This model is derived from the description of the case study in (Palmer 1995). In building it we will see how we can use abstraction to keep the model simple, yet meaningful, even in the presence of continuous

and non-continuous subsystems that have to be modeled together. In Section 4 we show how to go from interactor specification into SMV code which can be model checked, and in Section 5 we show how to go about model checking the resulting specification. Finally in Section 6 we analyse the results of the case study, and compare it with other approaches to the detection of mode related problems (Leveson & Palmer 1997, Rushby 1999).

2 Interactors and Partial Models

In this section we discuss the role of verification in interactive systems development, and how interactors can be used in the verification process. We also introduce the specific flavor of interactor we shall be using.

2.1 Role of Verification

Work on formal verification of interactive systems tends to fall into one of two categories:

- analysis of known problems of existing systems: mainly trying to explain why the problems arise;
- analysis of properties of specifications: mainly trying to establish if a given system specification exhibits some desired properties.

In the first case, hindsight is often used to drive the development of a model that will be analysed in order to elicit the particular problem under analysis. While this type of analysis can be useful in explaining what went wrong, it works *a posteriori* so it cannot predict design problems, only explain them. More than explaining problems, we would like to be able to predict them in order to adjust the design accordingly.

In the second case, the aim is that errors be detected and prevented before the system is actually built. It is still the case however that a complete specification has to be built. In systems as complex as interactive systems, this means a lot of design decisions and work might have to be redone if a problem is found.

In (Campos & Harrison 1998) we argue that in order to explore the benefits of formal verification fully, we must move the verification step into the development process. Instead of being used as a sanity check as the end result of design, verification should be used as a guide in the process of design decision making. This can be achieved by using verification techniques on partial models that highlight the specific concerns of different development stages (cf. Fields, Merriam & Dearden 1997).

Besides allowing for a more informed process of design decision making, the move towards a tighter integration between verification and design has a number of additional advantages:

- complexity control: interactive systems tend to be complex systems where a number of different areas come in to play; by building partial models focused on the specific aspects we want to analyse, we can better control the complexity of the models.
- reuse: we can think of reusing the verification process when similar aspects of two different systems are being analysed.

- technique fit: different styles of properties ask for different styles of verification; by using a number of models instead of a single one, we avoid being tied down to a particular verification technique.
- property relevance: when verifying a complex specification, we run the risk of some properties being more relevant of the specification itself than of the system being specified; by focusing our models on the specific aspects we want to analyse we are able to better ensure that the properties we formulate are relevant of the system and not only of how the system is specified.

2.2 The Interactor Language

Interactors, as developed by the York group (Duke & Harrison 1993) (see Figure 1), do not prescribe a specific specification notation for the description of interactor state and behaviour. Instead, they act as a mechanism for structuring the use of standard specification techniques in the context of interactive systems specification.

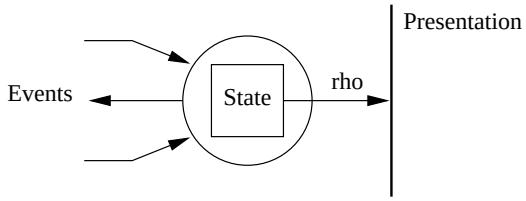

Figure 1: York Interactor

Several different formalisms have been used to specify interactors. These include Z (Duke & Harrison 1993), modal action logic (MAL) (Duke, Barnard, May & Duce 1995), and VDM (Harrison, Fields & Wright 1996). We will be using a (deontic) modal logic based on work done at Imperial College, London (Ryan, Fiadeiro & Maibaum 1991, Fiadeiro & Maibaum 1991), following its adaptation to interactor specification by Duke (see, for instance, Duke et al. 1995).

A first version of the interactor language used here was initially introduced in (Campos & Harrison 1998). Since then the language has evolved to include deontic operators (permission and obligation).

The definition of an interactor has three main components:

- state;
- behaviour;
- rendering.

The state of an interactor is modeled by a collection of typed attributes. Consider for example a dial which indicates a value, we would write:

interactor Dial
attributes
needle: Range

This declaration introduces an interactor named Dial. In this case the interactor has only one attribute (needle), and the type of the attribute is Range (the range of values in the dial).

In order to manipulate the state, actions have to be introduced. In this case we will only have an action to set the value in the dial:

action

set(Range)

The type, between parentheses after the action name, indicates that the action will have a parameter of that type (so set represents, in fact, a family of actions).

In this case we are assuming the existence of type Range. We can also define enumerated types:

types

T = {a, b, c}

This will be specially useful when translating specifications into SMV.

Until now nothing has been said about how the interactor behaves. In order to describe interactor behaviour we will use a logic based on Structured MAL (Ryan et al. 1991). Here propositional logic is augmented with the notion of action and:

- the deontic operator obl: if obl(ac) then action ac is obliged to happen some time in the future;
- the deontic operator per: if per(ac) then action ac is permitted to happen next;
- the modal operator $[\_]\_$: $[ac]a$ is the value of attribute a *after* action ac takes place;
- the special reference event []: $[]a$ is the value of attribute a in the initial state.

A major difference between our logic and Structured MAL is the treatment of the modal operator. There the modal operator is applied to whole propositions. This means that there is no way to relate *old* and *new* values of attributes directly. In Structured MAL this is usually done by the introduction of auxiliary variables. Suppose for example that we want to define an action ($++cpy$) which increments the value of an attribute (x) and stores its old value in another attribute (y). In MAL we would write:

$$(oldx = x) \rightarrow [++cpy](x = oldx + 1 \wedge y = oldx)$$

where $oldx$ is an auxiliary variable introduced to *carry* the value of x into the next state (after $++cpy$).

To avoid these auxiliary variables we follow the definition of modal operator from (Fiadeiro & Maibaum 1991), and we apply the operator to attributes only. This allows us to write:

$$[++cpy]x = x + 1 \wedge [++cpy]y = x$$

Since both modal operators in the axiom above concern the same action, we can factor them out and write:

$$[++cpy](x' = x + 1 \wedge y' = x)$$

interactor Dial
attributes
vis needle: Range
action
vis set(Range)
axioms
(1) [set(v)] needle'=v

Figure 2: Simple dial interactor

where priming is used to indicate which attributes are affected by the modal operator. We will omit the parentheses whenever the scope of the modal operator can be inferred.

The behaviour of set can be defined thus:

axioms
(1) [set(v)] needle'=v

Finally we have to define the rendering relation for the interactor presentation. This is done by annotating actions and attributes to show that they are perceivable. In addition, the annotation also indicates the modality of the perceivable attribute/action.

These annotations can be seen as defining additional operators. In this case operator vis asserts the fact that the parameter/action is visible.

Taking all these together we get the Dial in Figure 2.

The composition of interactors is done via inclusion as in (Ryan et al. 1991). We shall assume that all actions and attributes of an interactor are always accessible to other interactors that include it. It could be argued that it only makes sense to access attributes with an appropriate modality. For instance, if an attribute is not perceivable at all, then it might not make sense to access its value. (In this sense the notion of modality is a generalisation of the concept of sharable in (Ryan et al. 1991)). At this stage the specifier is free to access all attributes and actions whatever their modality.

To use a Dial in some other interactor we would write:

interactor Panel
includes
Dial via speedDial
...
axioms
(1) [] speedDial.needle=0
...

where speedDial becomes the name of a particular instance of Dial in the context of interactor Panel.

The modal operator allows us to talk about the effect of actions in the state. It says nothing, however, about when actions are permitted or required to happen. For this we must use the permission and obligation operators. As in (Ryan et al. 1991), we only consider the assertion of permissions and the denial of obligations. That is, axioms of the form:

- per(ac) $\rightarrow$ *guard* — action ac is permitted only if *guard* is true;
- *cond* $\rightarrow$ obl(ac) — if *cond* is true then action ac becomes obligatory.

This amounts to saying that permissions are asserted by default, and that obligations are off by default.

The rationale behind this decision is that it makes it easier to add permissions and obligations incrementally when writing specifications. For instance, permission axioms $\mathsf{per}(ac) \rightarrow guard_1$ and $\mathsf{per}(ac) \rightarrow guard_2$ add up to yield $\mathsf{per}(ac) \rightarrow (guard_1 \wedge guard_2)$. This logic is particularly appropriate to describing a system in which some components can be reused.

3 Modeling the MCP with interactors

As we have said already, the Palmer (1995) case study deals with a problem relating to altitude acquisition in a real aircraft (MD-88). This particular problem was identified during simulation of realistic flight missions, although Palmer (1995) notes that similar problems are frequently reported to the Aviation Safety Reporting System (ASRS n.d.).

3.1 Basic principles

When using automated systems, operators build mental models of the system which lead to expectations about system behaviour. When the system behaves differently from the expectations of the operator we have what is called an *automation surprise* (Woods, Johannesen, Cook & Sarter 1994). The interesting point about the present example is that the system behaved as designed (i.e. it did not malfunction) but nevertheless an automation surprise happened. This suggests that the system is misleading operators into forming false beliefs about its behaviour (i.e. wrong mental models).

While the use of simulation allows for the detection of shortcomings in design, it has some intrinsic problems. In order to perform a simulation an actual system or prototype has to be built, this means that simulations are costly and can only be done late in the design/development life cycle, when design decisions have already been committed to, and change is difficult.

The ability to analyse and predict potential problems from the initial stages of design would reduce the number of problems found later in the simulation stage. One of the implications of doing this early analysis is that it has to be done without the benefit of hindsight (apart from what has been learnt from previous analysis and systems). We are not trying to explain why something went wrong, instead we are using the analysis, during design, to identify potential sources of problems. In this context, hindsight is not available. So, when developing a methodology for the integration of verification into design, we must be careful to avoid relying on it.

In keeping with the above principle, we will not model the system around the scenario presented by Palmer (1995). Instead we will model the system from a more general point of view, and then analyse those aspects of the behaviour that are highlighted by the case study. Hence, if we are able to detect the problem, then we will have shown that it would be possible to have prevented that same problem from creeping unnoticed into the design of the aircraft.

There are of course problems with this particular case study. We had already read the Palmer (1995) paper and therefore were tainted by it. Nevertheless, it is quite different to build the model around the scenario, or to build a generic

interactor dial(T)
attributes
vis needle: T
actions
vis set(T)
axioms
(1) [set(v)] needle′ = v

Figure 3: Parametrised dial interactor

model and use the scenario as a starting point for analysis. In the first case the results of the scenario directly influence the model so that the analysis is biased by hindsight. In the second case we use the scenario only to set up a context for verification, the verification process itself is independent from the results described in the scenario. In fact, in the case of a system still under development, the scenario might very well be only an idealised description of how the system should function in a particular situation.

A specification of the system was built using interactors that describe how the MCP influences the airplane flight path regarding its altitude. The MCP has three main dials, indicating:

- vertical speed;
- airspeed;
- the altitude window (i.e. altitude to which the airplane should climb)

While airspeed, and altitude can only be positive values, the vertical speed can either be positive (going up) or negative (going down). Hence, we use a parametrised interactor to represent dials (see Figure 3).

At this level of abstraction, dials are represented by an attribute (*needle*) and an action (*set*). The attribute represents the value indicated by the dial's needle. The action sets the value of the attribute (Axiom 1).

In order to analyse a system we need to place it in its context of operation. The MCP will not be unsafe in itself, it only makes sense to talk of shortcomings in its design in relation to the actual system that the MCP is influencing. This is further illustrated by the fact that pilots usually become aware of unexpected or erroneous aircraft behaviour, not by looking at the automation state, but by looking at the actual system state (Palmer 1995).

3.2 Modelling the context as a finite system

In this case we need to model the aircraft state in order to relate it to the automation state. Regarding the scenario that will be under consideration (altitude acquisition) the interesting information is its vertical speed, airspeed, and altitude.

Modelling this information poses an interesting problem. The aircraft is a continuous system, but our specifications are discrete. This means we must find a way to discretise the behaviour of the airplane in order to be able to express it in our specification language. We can do this using abstractions. In this case

interactor airplane
attributes
altitude: Altitude
airSpeed: Velocity
climbRate: ClimbRate
actions
fly
axioms
(1) (altitude$>0 \rightarrow$ [fly]
(altitude$' >=$altitude - 1 $\wedge$ altitude$' <=$altitude + 1) $\wedge$
(altitude$' <$altitude $\rightarrow$ climbRate$' <0$) $\wedge$
(altitude$'=$altitude $\rightarrow$ climbRate$'=0$) $\wedge$
(altitude$' >$altitude $\rightarrow$ climbRate$' >0$))
(2) altitude$=0 \rightarrow$ [fly] ((altitude$' >=$altitude $\wedge$ altitude$' <=$altitude + 1) $\wedge$
(altitude$'=$altitude $\rightarrow$ climbRate$'=0$) $\wedge$
(altitude$' >$altitude $\rightarrow$ climbRate$' >0$))

Figure 4: The airplane

we need to substitute a state variable that ranges over a (potentially) non-finite state space, by a corresponding (abstracted) state variable that ranges over a (small) finite domain (cf. Dwyer, Carr & Hines 1997). By doing this we restrict the possible behaviours of the original system to a subset that we can model. The spirit of this process is similar to that applied by Hayes (1990) in order to specify a continuous (ideal) process in a system with physical limitations. However, he does it by adding detail to the specification, while we abstract away details of the ideal system.

We must be careful that our abstraction process captures the interesting features of the system. If we imagine the state of the aircraft as a (continuous) flow over time, and place by its side a sequence of states representing interactor behaviour, we can then match the interactor states to interesting moments in the behaviour of the aircraft.

In the present context we are specially interested in the altitude, so states will correspond to changes in the altitude by some amount (say 1 in some unit of measure). In order to make the state transitions possible the action *fly* is introduced. Besides asserting the change of altitude in each transition, the axioms relate climb rate to the altitude change.

The model for the airplane is shown in Figure 4

3.3 Modelling the MCP

How the MCP influences the automation will depend on its operating Pitch Mode. There are four pitch modes:

- `VERT_SPD` (vertical speed pitch mode): instructs the airplane to maintain the climb rate indicated in the MCP (the airspeed will be automatically adjusted);
- `IAS` (indicated airspeed pitch mode): instructs the airplane to maintain the airspeed indicated in the MCP (the climb rate will be automatically

interactor MCP
includes
airplane via plane
dial(ClimbRate) via crDial
dial(Velocity) via asDial
dial(Altitude) via ALTDial
attributes
vis pitchMode: PitchModes
vis ALT: boolean
actions
vis enterVS, enterIAS, enterAH, toggleALT
enterAC
axioms
Action effects
(1) [crDial.set(t)] pitchMode′=VERT_SPD ∧ ALT′=ALT
(2) [asDial.set(t)] pitchMode′=IAS ∧ ALT′=ALT
(3) [ALTDial.set(t)] pitchMode′=pitchMode ∧ ALT′
(4) [enterVS] pitchMode′=VERT_SPD ∧ ALT′=ALT
(5) [enterIAS] pitchMode′=IAS ∧ ALT′=ALT
(6) [enterAH] pitchMode′=ALT_HLD ∧ ALT′=ALT
(7) [toggleALT] pitchMode′=pitchMode ∧ ALT′ ≠ALT
(8) [enterAC] pitchMode′=ALT_CAP ∧ ¬ALT′
Permissions
(9) per(enterAC) → (ALT ∧ (ALTDial.needle - plane.altitude)≤2)
Obligations
(10) (pitchMode=ALT_CAP ∧ plane.altitude=ALTDial.needle) → obl(enterAH)
Invariants
(11) pitchMode=VERT_SPD → plane.climbRate=crDial.needle
(12) pitchMode=IAS → plane.airSpeed=asDial.needle
(13) pitchMode=ALT_HLD → plane.climbRate=0

Figure 5: The Mode Control Panel

adjusted);

- `ALT_HLD` (altitude hold pitch mode): instructs the airplane to maintain the current altitude;

- `ALT_CAP` (altitude capture pitch mode): internal mode used by the airplane to perform a smooth transition from `VERT_SPD` or `IAS` to `ALT_HLD` (see `ALT` below).

We therefore define the type:
PitchModes = {VERT_SPD, IAS, ALT_HLD, ALT_CAP}
Additionally, there is a capture switch (`ALT`) which, when armed, causes the airplane to stop climbing when the altitude indicated in the MCP is reached.

The MCP operation is described by the interactor in Figure 5. Note that setting the climb rate or airspeed causes the pitch mode to change accordingly (Axioms 1 and 2), and that setting the altitude dial arms the altitude capture

(Axioms 3). Axioms 4 to 8 are introduced to define the effect of the interactor's own actions, basically changing between different pitch modes and toggling the altitude capture. Axiom 9 states under which conditions the pitch mode can be set to ALT_CAP: the altitude capture must be armed, and the plane must be inside some neighbourhood of the target altitude. Regarding our abstract specification, the restriction on the value of this distance is that it should not be too small, in order to allow for the system to evolve while inside the neighbourhood of the target altitude. We have chosen the value 2. Axiom 10 states the obligation to enter ALT_HLD once the desired altitude has been reached. And finally, Axioms 11 to 13 describe the effect of the pitch modes on the state of the airplane.

At this stage we have left the types unspecified.

Having written the model we now want to analyse it. Before we can do it two further steps are necessary. First, we need to obtain a checkable version of the model. Second, we must define what properties we are interested in.

Since we are interested in automated verification, and we have no tools to analyse MAL specifications, we will be translating our models into SMV. We have developed a model to automate this translation and Section 4 explains how the translation is done. Identifying and checking interesting properties is dealt with in Section 5.

4 From Interactors to SMV

In order to allow for the automated verification of interactor models, a compiler has been developed to perform the translation from our interactor language to SMV (a model checking tool). Here we will briefly describe the rationale behind it. A more formal justification for the translation process is given in Appendix A.

4.1 Interactors as SMV modules

SMV (McMillan 1992) has been previously used in the verification of interactive system specifications by Abowd, Wang & Monk (1995). Their approach, however, relies on a simple propositional production system written in Action Simulator (Monk & Curry 1994). In fact, Action Simulator is proposed as a discount tool to be used in very early stages of design.

With interactors we build specifications compositionally. Fortunately SMV has a notion of module, which corresponds roughly to the notion of interactors as agents. So, we can translate each interactor into a SMV module.

Each module can be seen as defining a finite state machine. In its simplest form, a module consists of a number of state variables (cf. interactor attributes), and a set of rules specifying how the module can progress from one state to the next (cf. interactor modal axioms). Each module might also have an initial condition defining its initial state (cf. interactor reference event axioms), a set of invariant formulas that must hold in all states of the module (cf. interactor invariant axioms), and a fairness constraint stating a property that must hold infinitely often during the execution of the module. Additionally, a module might include other modules as state attributes (cf. interactor inclusion).

We can already see a number of similarities that can be used in the translation process. In order to better illustrate the process we will translate parts of the MCP interactor presented in Figure 3 above. Using the similarities noted above we can start to write the SMV module that corresponds to the MCP interactor (question marks are used where the translation is not yet possible):

```
MODULE MCP
VAR
  plane: airplane;
  crDial: dialClimbRate;
  asDial: dialVelocity;
  ALTDial: dialAltitude;
  pitchMode: {VERT_SPD, IAS, ALT_HLD, ALT_CAP};
  ALT: boolean;
-- Axioms 1..8 - modal axioms
TRANS ??
-- Axiom 9 - permission axiom
TRANS ??
-- Axiom 10 - obligation axiom
TRANS ??
-- Axioms 11..13 - state invariants
INVAR pitchMode=VERT_SPD -> plane.climbRate=crDial.value
INVAR pitchMode=IAS -> plane.airSpeed=asDial.value
INVAR pitchMode=ALT_HLD -> plane.climbRate=0
```

We can see (clause `VAR`) how both included interactors and attributes become state variables of the SMV module. Note however that the Dial interactor has been split up into three different modules: `dialClimbRate`, `dialVelocity`, and `dialAltitude`. Since SMV modules cannot be parameterised with types, we have to create one module for every different instantiation of the original interactor. Note also that we have replaced type names by their definitions. `boolean` is a given type in SMV, so we do not have to replace it. Additionally, we can see how invariants are translated directly into `INVAR` clauses (clauses defining conditions that must be true in all states of the module).

At this stage we have not included information about the visibility of actions or attributes. This information could be encoded in SMV using additional boolean state variables defining whether each given action/attribute is visible or not.

Axioms 1 to 10 have also not been translated. We will now see how to translate each particular type of those axioms.

4.2 Modal Axioms

The modal axioms (Axioms 1 to 8) state the effect of the various actions on the state of the interactor. However, we do not have such a notion of action in SMV. To solve this we use a strategy similar to that used by (Abowd et al. 1995). We augment the state information of each module with a special variable named `action`. The role of this variable is to indicate which action was responsible for the transition into the current state.

This is different from the Abowd et al. (1995) approach, where each state holds information about which action will happen next. We choose to include information about the last action instead of the next one, since Abowd et al.'s

(1995) approach can cause problems when analysing some specific types of properties (see below).

Figure 6 exemplifies how this variable works. We can imagine interactors as

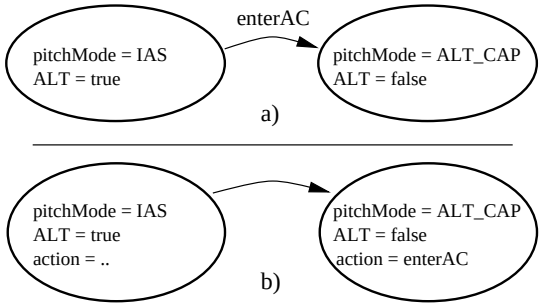

Figure 6: From labelled to unlabelled state machines

defining labelled state machines where the states hold the values of the attributes and transitions are labelled by the actions that cause the state changes (Figure 6-a). Since SMV finite state machines are unlabelled, what we do is to move the information about the label of each transition into the arrival state (Figure 6-b).

The pitfall with this approach is that when we have two different actions leading into the same state, we have to duplicate that state (see Figure 7). If

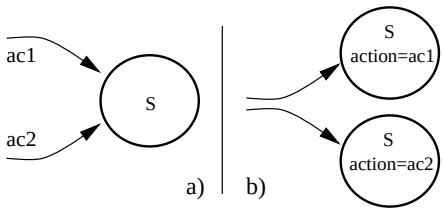

Figure 7: State duplication

we had encoded in each state which action will happen next, then we would have the symmetrical problem of having to duplicate states which could be left in more than one way.

With our approach, when we identify a set of states by some condition (say all states where the altitude toggle is off), we guarantee that they all share a set of common actions: those actions which the condition enables (in this case all but enterAC). With the Abowd et al. (1995) approach, however, this is not true. In this specific case we would have each state where ALT is off, *exploded* into as many states as the actions that are possible from it. This can complicate the analysis when we want to look at the behaviour of the system from all states that obey some given property.

Axioms 1 to 3 pose an additional problem since the actions involved have parameters. As for interactors, we split each action into as many actions as the possible values the parameter might take. For instance, action crDial.set(t)

(the set action of the crDial interactor) generates SMV actions `crDial.set_-1`, `crDial.set_0`, and `crDial.set_1`.

Using the strategy above, Axiom 1 generates the following clauses:

```
TRANS
  next(crDial.action)=set_-1 ->(next(pitchMode)=VERT_SPD & next(ALT)=ALT)
TRANS
  next(crDial.action)=set_0 -> (next(pitchMode)=VERT_SPD & next(ALT)=ALT)
TRANS
  next(crDial.action)=set_1 -> (next(pitchMode)=VERT_SPD & next(ALT)=ALT)
```

Axioms 2 and 3 follow in the same way; and Axiom 4 generates the clause:

```
TRANS next(action)=enterVS -> (next(pitchMode)=VERT_SPD & next(ALT)=ALT)
```

with Axioms 5 to 8 following in the same way. Note the use of `next` (identify the next state of the module) instead of priming.

Each of these `TRANS` clauses states the relation between the current and the next state if the respective action is the next action to happen. In the clauses resulting from Axiom 1-3 we are referring to the effect of an action of a different module (interactor) and what effect it has on the state of this module.

Initially the value of `action` will be `nil` (no action has happened), and the value of `oblenterAH` will be false. This is stated in the `INIT` clause, which defines the initial state of the module. This clause is also used when translating axioms involving the reference event [] (i.e. axioms defining the initial state of the interactor). See Section 5.1 for an example of one such axiom.

The `INIT` clause for this module is:

```
INIT action=nil & !oblenterAH
```

4.3 Deontic Axioms

We are left with permission and obligation axioms. Permission axioms are simple to translate. By default actions are always permitted, permission axioms are used to restrict their permission to specific conditions. Hence we can create for each such axiom a `TRANS` clause stating which condition must hold in order for the action to be the next action. These clauses will look like the ones for modal axioms except that the right side of the implication refers only to the current state. Axiom 9 generates the SMV clause:

```
TRANS next(action)=enterAC -> (ALT & (ALTDial.value - plane.altitude)<=2)
```

The translation of obligations is more problematic. Two interpretations of obligation are possible:

1. the obligation to perform an action means that the action must happen immediately;

2. the obligation to perform an action means that the action must happen some time in the future.

Both alternatives have their own merits, but the second seems more natural and is usually adopted (cf. Ryan et al. 1991, Fiadeiro & Maibaum 1991).

The choice of this interpretation means that we have no direct way of expressing obligations. Remember that `TRANS` clauses refer only to the immediate

next state. To express this we introduce one auxiliary state variable for each action that has an associated obligation axiom. This will be a boolean variable which will be true if an obligation to perform the action is in effect, and false otherwise. By additionally setting up a fairness constraint requiring that the variable be infinitely often false, we guarantee that every time the variable becomes true (an obligation is asserted), it will eventually become false again (the obligation is discharged).

Hence, Axiom 11 generates the following SMV clauses:

```
-- Fulfilling an obligation
TRANS next(action)=enterAH -> !next(oblenterAH)
-- Keeping existing obligations
TRANS !next(action)=enterAH ->
     next(oblenterAH)=((pitchMode=ALT_CAP & plane.altitude=ALTDial.value)
                       | oblenterAH)
FAIRNESS
  !oblenterAH
```

Note how the obligation axiom generates the `oblenterAH` variable (here we have omitted its declaration) and two `TRANS` clauses. One for the fulfilment of the obligation (when the next action is `enterAH`). Another for setting or keeping the value of the variable, depending on the obligation condition (when the next action is not`enterAH`). Finally the fairness condition is also generated.

We have shown how the obl operator can be translated into SMV. We have chosen to give it a weak interpretation of obligation (some time in the future). In Section 5.3 we will show how it is also possible to express the stronger version of the concept.

The translation algorithm has been implemented in Perl (see Wall, Christiansen & Schwartz 1996, for an introduction to Perl).

5 Checking the Design

Having defined the translation from interactors to SMV, we will now analyse the specification developed in Section 3.

5.1 Situating the analysis

In the same way that we needed to place our model in its context of operation (cf. Section 3), we also need to place it in specific contexts of usage in order to be able to reason about usability related properties.

We must make it clear that this notion of context of usage is different from the traditional notion of scenario. We do not use it as a *post hoc* device depicting a specific interaction story. Instead, we use it as a tool to illustrate possible uses that are envisaged for the system. In this context, usage sets the scene for a plausible interaction story, enabling us to situate the analysis.

Usually these contexts might be obtained from the requirements definition stage (cf. Fields, Harrison & Wright 1997). In the present case we will use one based on one of the scenarios put forward in (Palmer 1995) (altitude acquisition). Note, however, that we do not use the incident description, only the context in which the interaction takes place.

We consider a situation where the airplane is climbing in autopilot towards the altitude indicated in the MCP. In order to express this we add the following initialisation axiom to interactor MCP:

$$[] \quad \mathsf{crDial.needle} = 1 \wedge \mathsf{ALTDial.needle} = 3 \wedge \mathsf{plane.altitude} = 0 \wedge \mathsf{pitchMode} = \mathsf{VERT\_SPD} \wedge \mathsf{ALT}$$

So, initially the airplane is at altitude 0, climbing in vertical speed pitch mode, and with the altitude capture on. The target altitude is 3.

5.2 Converting the Model

In order to check the specification some adjustments have to be made. The most relevant is the need to only have enumerated types in the specification. Regarding altitude and velocity this does not represent a problem. In fact, the airplane will have its own physical limitations on maximum speed and altitude. We must only make sure that the selected maximum value (hence, the maximum altitude) is higher than the tolerance distance in Axiom 9 of interactor MCP. We choose to represent both as the range 0 to 4.

Regarding climb rate, we have to distinguish between three situations: climbing, holding altitude, or descending. Hence, we will consider three values: -1 (to represent all negative climb rates), 0, and 1 (to represent all positive climb rates).

We add the following types to the specification:

Velocity = {0, 1, 2, 3, 4}
Altitude = {0, 1, 2, 3, 4}
ClimbRate = {-1, 0, 1}

As a consequence of this, we have to change the behaviour of the interactor plane to take into account the maximum altitude. We also change the name of interactor MCP to main, and add two extra clauses to it: test, and fairness. test is used to specify the CTL formula that should be checked by SMV. In fairness we state that the system should not be continuously idle.

The checkable version of the specification is presented in Appendix B. This version is automatically convertible to SMV using the compiler.

We have set up the context for our analysis, and translated our specification to SMV. We now have to identify relevant properties which can be analysed using the model checker.

5.3 Formulating and checking properties

CTL formulae can be automatically checked by SMV. We need, then, to express relevant user properties as CTL formulae. Since they are related to the user, the process of obtaining these formulae becomes, as we will see, the focus for interdisciplinary discussion. We could, for instance, ask a human factors expert to produce a set of meaningful expectations that the user might have about the system.

One plausible expectation pilots might have is that, if the altitude capture (`ALT`) is armed, the airplane will stop at the desired altitude (selected in `ALTDial`). We can express this as the CTL formula:

```
AG(ALT -> AF(pitchMode=ALT_HLD & plane.altitude=ALTDial.needle))
```

which we include as the clause test in interactor main. The formula reads: it always (AG) happens that, if the altitude capture is on, then it is inevitable (AF) that the altitude set on the altitude dial will be reached, and the pitch mode changed to altitude hold.

When we model check a specification in SMV, the checker answers whether or not the test succeeds, and if the answer is false it tries to give a counter example. If we try to check the specification against the formula above, we get the following result[1]:

```
-- specification AG (ALT -> AF (pitchMode = ALT_HLD & ... is false
-- as demonstrated by the following execution sequence
state 1.1:
oblenterAH = 0
ALT = 1
pitchMode = VERT_SPD
ALTDial.needle = 3
ALTDial.action = nil
asDial.needle = 4
asDial.action = nil
crDial.needle = 1
crDial.action = nil
plane.climbRate = 1
plane.airSpeed = 4
plane.altitude = 0
plane.action = nil
action = nil

state 1.2:
ALT = 0
plane.altitude = 1
plane.action = fly
action = toggleALT

...

resources used:
user time: 37.54 s, system time: 0.38 s
BDD nodes allocated: 126733
Bytes allocated: 3145728
BDD nodes representing transition relation: 6408 + 174
```

What the model checker points out is that the user might toggle the altitude capture off. This means that the formula is not true.

This situation prompts us to analyse whether this particular situation is (or should be) covered in the operation manuals, or about how easy it is to switch the capture on or off and whether the mechanism needs reviewing.

These are valuable outcomes of the verification process and show that the process is not self contained, but prompts questions that have to be dealt with at other stages of design.

Getting back to the case study, we consider that the point is a valid one, and so we refine our abstraction of the pilot's belief. The test formula now becomes:

[1]Note that from state to state only those values that have changed are shown. Also, for brevity we only show enough of the counter example to make the point about ALT.

```
AG(ALT -> AF((pitchMode=ALT_HLD & plane.altitude=ALTDial.needle)
             |
             action=toggleALT))
```

It reads: in the conditions stated, either the plane stops at the desired altitude, or else the pilot has switched the capture off.

When we try to check the specification with this formula we still get a negative result. The obligation to perform enterAH is too weak. Axiom 10 states that the airplane is obliged to enter altitude hold when both the target altitude is reached and the pitch mode is ALT_CAP. Since the notion of obligation is *some time in the future*, and not *immediately* the airplane can go above the target altitude before it engages ALT_HLD.

Depending on whether this models the actual behaviour of the airplane, this can be seen as a problem with the system or a problem with our specification. In any case it is a reminder that caution must be taken regarding how ALT_CAP is implemented (it must monitor the current altitude sufficiently frequently).

Assuming that this situation arises due to our notion of obligation, we strengthen Axiom 10 to:

$$\mathsf{pitchMode} = \mathsf{ALT\_CAP} \wedge \mathsf{plane.altitude} = \mathsf{ALTDial.needle} \rightarrow \mathsf{action}' = \mathsf{enterAH}$$

What the axiom now asserts is the stronger version of obligation mentioned previously: if the automation is in altitude capture pitch mode, and the airplane reaches the target altitude, then the next immediate action must be enterAH.

Since the airplane is obligated to immediately switch to altitude hold when it reaches the desired altitude, we expect this to eventually happen. When we try the previous property in the new system, the answer is still no! This time, the model checker points out that changing the pitch mode to IAS (for instance by setting the air-speed dial) when in ALT_CAP, effectively kills the altitude capture (i.e. the request to stop at the target altitude). In effect, when the pitch mode changes to ALT_CAP, the altitude capture is automatically switched off, however the airplane is still climbing. This means any subsequent action from the pilot that causes a change of the pitch mode, will cause the aircraft to keep climbing past the target altitude.

If we refer back to (Palmer 1995) we see that this is exactly the problem that was detected during simulation. Note that we achieved this result without using knowledge from the simulation results. We could have applied this process based only on a pen and paper scenario of an aircraft that was yet in its early design stages (in fact, that is the aim of the process) and effectively detected the problem.

Having decided to analyse the MCP panel, we built a generic model of it. In order to be able to analyse such a model we had to include relevant abstractions of the airplane. We then used CTL formulae to express user expectations about the operation of the system. The formulation and interpretation of these formulae acted as a focus for interdisciplinary discussion. More important than verifying that a property is true, the process of trial and error raises issues about the specification. Issues that might not be solvable at the level of the verification alone, but trigger discussion and intervention from other areas of expertise.

6 Conclusion

In this paper we have looked at the automated verification of early specifications of interactive systems.

Interactive systems are complex systems which pose difficult challenges to verification. By bringing the verification process closer to the design process we aim at better capturing the multiple concerns that come into play in the design of interactive systems, and at making better use of the available techniques.

We have shown how interactor specifications can be translated into SMV and how we can use such specifications in conjunction with the model checker to model and analyse a realistic case of mode confusion. It is worth noticing how focusing our specification in the specific context of altitude acquisition, allowed us to produce a small yet meaningful specification.

The specific case study that we have used is also analysed in (Leveson & Palmer 1997) and (Rushby 1999). Leveson & Palmer (1997) write a formal specification, based on a control loop model of process-control systems, using AND/OR tables. This specification is then analysed, manually, in order to look for potential errors caused by indirect mode changes (i.e. changes that occur without direct user intervention).

An advantage of using a manual analysis process is greater freedom in the specification language, which can potentially lead to more readable specifications. However, we feel that the possibility of performing the analysis in an automated manner will be an advantage when analysing complex systems. We address the issue of readability by using a high level specification language (interactors) which is then translated into SMV.

Rushby (1999) reports on the use of Murϕ to automate the detection of potential automation surprises, using (Palmer 1995) as an example. He builds a finite state machine specification that describes both the behaviour of the automation, and of a proposed mental model of its operator. He then expresses the relation between the two as an invariant on the states of the specification. Murϕ is used to explore the state space of the specification and look for states that fail to comply with the invariant (i.e. mismatches between both behaviours).

Looking at his specification, we see that it is much more focused on the specific mode problem that will be under consideration than ours. This seems to confirm our belief that our approach is more flexible. In fact, our aim is to develop a general purpose methodology for the automated analysis of interactive systems. While we used the mode problem as a case study, the methodology can also be applied to the analysis of other issues. For example, task related properties, lock-in and interlock issues, or awareness (in Campos & Harrison 1998, we give a simple example involving the analysis of awareness).

The use of interactors and SMV gives us a good degree of freedom and expressive power, but that does not come without a cost. In particular, the use of CTL, while allowing for the expression of possibility, means that fairness concerns become an issue. As an example, in the case study above we can have a situation where the pilot repeatedly changes the climb rate of the airplane between positive and negative, effectively preventing it from reaching the altitude set in the capture. Situations of this kind can be solved either by imposing stronger fairness constraints on the system, by altering the property being checked (as was done for the ALT toggle), or by reworking the specification with the particular context of analysis in mind.

An issue that is raised by the use of partial models is whether we are using the appropriate scenarios and abstractions. However, this is not an exclusive problem of this approach. Even if we could build a complete specification encompassing all relevant aspects of a system, and we had a powerful enough tool to analyse every aspect of it that we might which. It would still be the case that it would be up to us to decide what questions to ask of that specification. And we would still have the problem of determining if we have asked all the right questions. In the end it is always a matter of expertise (cf. Fields, Merriam & Dearden 1997).

Finally we have hinted at how the verification process can be useful by raising questions that have to be addressed in a broader context than the verification itself. This is in line with our aim of developing a comprehensive methodology for the development of interactive systems.

Acknowledgements

The authors thank Bob Fields and Karsten Loer for their useful comments on earlier versions of this paper.

References

Abowd, G. D., Wang, H.-M. & Monk, A. F. (1995), A formal technique for automated dialogue development, *in* 'Proceedings of the First Symposium of Designing Interactive Systems - DIS'95', ACM Press, pp. 219–226.

ASRS (n.d.), 'Aviation safety reporting system'.
*http://olias.arc.nasa.gov/ASRS/ASRS.html

Campos, J. C. & Harrison, M. D. (1998), The role of verification in interactive systems design, *in* P. Markopoulos & P. Johnson, eds, 'Design, Specification and Verification of Interactive Systems '98', Springer Computer Science, Eurographics, Springer-Verlag/Wien, pp. 155–170.

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

Doherty, G., Campos, J. C. & Harrison, M. D. (1998), Representational reasoning and verification, *in* J. I. Siddiqi, ed., 'Proceedings of the BCS-FACS Workshop: Formal Aspects of the Human Computer Interaction', SHU Press, pp. 193–212. ISBN 0 86339 7948.

Duke, D., Barnard, P., May, J. & Duce, D. (1995), Systematic development of the human interface, *in* 'Asia Pacific Software Engineering Conference', IEEE Computer Society Press, pp. 313–321.

Duke, D. J. & Harrison, M. D. (1993), 'Abstract interaction objects', *Computer Graphics Forum* **12**(3), 25–36.

Dwyer, M. B., Carr, V. & Hines, L. (1997), Model checking graphical user interfaces using abstractions, *in* M. Jazayeri & H. Schauer, eds, 'Software

Engineering — ESEC/FSE '97', number 1301 *in* 'Lecture Notes in Computer Science', Springer, pp. 244–261.

Faconti, G. & Paternò, F. (1990), An approach to the formal specification of the components of an interaction, *in* C. Vandoni & D. Duce, eds, 'Eurographics '90', North-Holland, pp. 481–494.

Fiadeiro, J. & Maibaum, T. (1991), 'Temporal reasoning over deontic specifications', *Journal of Logic and Computation* **1**(3), 357–395.

Fields, B., Harrison, M. D. & Wright, P. (1997), THEA: Human error analysis for requirements definition, Technical Report YCS 249, Department of Computer Science, University of York, Heslington, York, YO10 5DD, England.

Fields, B., Merriam, N. & Dearden, A. (1997), DMVIS: Design, modelling and validation of interactive systems, *in* M. D. Harrison & J. C. Torres, eds, 'Design, Specification and Verification of Interactive Systems '97', Springer Computer Science, Eurographics, Springer-Verlag/Vien, pp. 29–44.

Harrison, M., Fields, R. & Wright, P. C. (1996), The user context and formal specification in interactive system design (invited paper), *in* C. R. Roast & J. I. Siddiqi, eds, 'BCS-FACS Workshop on Formal Aspects of the Human Computer Interface', electronic Workshops in Computing, Springer. http://www.springer.co.uk/ewic/workshops/FAHCI/.

Hayes, I. (1990), Specifying physical limitations: A case study of an oscilloscope, Technical Report 167, Key Centre for Software Technology, Department of Computer Science, University of Queensland. revised 1993.

Leveson, N. G. & Palmer, E. (1997), Designing automation to reduce operator errors, *in* 'Proceedings of the IEEE Systems, Man, and Cybernetics Conference'.

McMillan, K. L. (1992), *The SMV system*, draft edn, Carnegie-Mellon University.

Monk, A. F. & Curry, M. B. (1994), Discount dialogue modelling with Action Simulator, *in* G. Cockton, S. W. Draper & G. R. S. Weir, eds, 'People and Computer IX - Proceedings of HCI'94', Cambridge University Press, pp. 327–338.

Palmer, E. (1995), "Oops, it didn't arm." - a case study of two automation surprises, *in* R. S. Jensen & L. A. Rakovan, eds, 'Proceedings of the Eighth International Symposium on Aviation Psychology', Ohio State University, Columbus, Ohio, pp. 227–232.
*http://olias.arc.nasa.gov/~ev/OSU95_Oops/PalmerOops.html

Rushby, J. (1999), Using model checking to help discover mode confusions and other automation surprises, *in* '(Pre-) Proceedings of the Workshop on Human Error, Safety, and System Development (HESSD) 1999', Liège, Belgium.

Ryan, M., Fiadeiro, J. & Maibaum, T. (1991), Sharing actions and attributes in modal action logic, *in* T. Ito & A. R. Meyer, eds, 'Theoretical Aspects of Computer Software', Vol. 526 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 569–593.

Wall, L., Christiansen, T. & Schwartz, R. L. (1996), *Programming Perl*, 2nd edn, O'Reilly & Associates, Inc.

Woods, D. D., Johannesen, L. J., Cook, R. I. & Sarter, N. B. (1994), Behind human error: Cognitive systems, computers, and hindsight, State-of-the-Art Report SOAR 94-01, CSERIAC.

A Expressing Interactors in SMV

A SMV specification is a collection of modules, much in the same way as our specifications are collections of interactors. Since SMV modules provide the same facilities for structuring as our interactors, including importing modules into other modules, we will be representing each interactor as a SMV module.

For an introduction to the SMV input language see (McMillan 1992). In the particular style we will be using, each module declares a collection of attributes, a condition defining the initial state, and a collection of axioms specifying the valid transitions between states. For instance, the axiom:

```
next(x):= x+1 & next(y):=x
```

defines a transition where in the next state the value of `x` has been incremented by one, and the value of `y` becomes the previous value of `x`.

Since at the agent level the languages are similar, we will focus our description of the translation process at the behaviour level (how modal and deontic axioms are translated). In particular, we will not address the syntax of both languages.

We start by defining interpretation structures for both languages and then show how we can interpret the interactor axioms in the SMV interpretation structure.

A.1 Interactors

Given a set of types T, an interactor defines, as we have seen, a family of attributes A, a mapping t: $\mathsf{A} \to \mathsf{T}$ giving the type of each attribute, a collection of actions E, and a collection of propositional, deontic and modal axioms describing the effect of the actions.

An interpretation structure for such an interactor is defined as (cf. Ryan et al. 1991, Fiadeiro & Maibaum 1991):

Definition 1 *Let* **I** *be an interactor with attributes* A*, a mapping* t: $\mathsf{A} \to \mathsf{T}$*, and actions* E*. An* Interpretation Structure *for* **I** *is a 5-tuple* $(\mathcal{W}, \mathcal{M}, \mathcal{I}, \mathcal{P}, \mathcal{O})$ *such that:*

- $\mathcal{W}$ *is a non-empty set of states;*
- $\mathcal{M}$ *maps each action to the transitions it may trigger (in the case of the special reference action [] it maps the action to a state — the initial state):*

- $\mathcal{M}([]) \in \mathcal{W}$
- $\mathcal{M}(e) : \mathcal{W} \to \mathcal{W}$ *for all* $e \in \mathsf{E}$

- $\mathcal{I}$ *maps each attribute to a state dependent value:*
  - $\mathcal{I}(a) : \mathcal{W} \to \mathsf{t}(a)$ *for all* $a \in \mathsf{A}$

- $\mathcal{P}$ *and* $\mathcal{O}$ *are relations in* $\mathsf{E} \times \mathcal{W}$ *stating which events are permitted/obligatory in each state.*

In the remainder of this section we will use the notation $prop(a_1, .., a_n)$ to denote a propositional axiom on state attributes $a_1, .., a_n$ — i.e. an axiom that relates attributes *in a state*, making no use of modal or deontic operators.

Given an interpretation structure $\mathbf{P}$ for an interactor, we define satisfaction of an axiom by $\mathbf{P}$ in a state $w \in \mathcal{W}$ (denoted by $P, w \models axiom$) as follows:

- $P, w \models prop(a_1, .., a_n)$ iff $prop(\mathcal{I}(a_1)(w), .., \mathcal{I}(a_n)(w))$ — i.e. the validity of a propositional axiom depends only on the current state;
- $P, w \models []prop(a_1, .., a_n)$ iff $w = \mathcal{M}([])$ and $prop(\mathcal{I}(a_1)(w), .., \mathcal{I}(a_n)(w))$ — i.e. the axiom is evaluated in the initial state;
- $P, w \models prop([e]a_1, .., [e]a_n, a_k, .., a_l)$ iff for all $w' \in \mathcal{W}$ if $w' = \mathcal{M}(e)(w)$ then $prop(\mathcal{I}(a_1)(w'), .., \mathcal{I}(a_n)(w'), \mathcal{I}(a_k)(w), .., \mathcal{I}(a_l)(w))$ — i.e. in modal axioms attributes in the scope of a modal operator are evaluated in the state after the action happens;
- $P, w \models \mathsf{per}(e)$ iff $(e, w) \in \mathcal{P}$;
- $P, w \models \mathsf{obl}(e)$ iff $(e, w) \in \mathcal{O}$.

A.2 SMV modules

Similarly to interactors, given a set $\mathtt{T}$ of (enumerated) types, a SMV module defines a family of attributes $\mathtt{A}$, a mapping $\mathtt{t} : \mathtt{A} \to \mathtt{T}$, and collection of axioms defining the valid transitions. In this case, however, no actions are defined. Additionally, a initial state condition, and a fairness condition are defined.

Definition 2 *An* Interpretation Structure *for a given SMV module is a 5-tuple* $(\mathtt{S}, \mathtt{R}, \mathtt{I}, \mathtt{F}, \mathtt{S}_0)$ *where (cf. Clarke et al. 1986):*

- $\mathtt{S}$ *is a finite set of states (cf.* $\mathcal{W}$*);*
- $\mathtt{R} \subseteq \mathtt{S} \times \mathtt{S}$ *is a binary relation which gives the possible transitions between states;*
- $\mathtt{I}$ *maps each attribute in* $\mathtt{A}$ *to a state dependent value:*
  - $\mathtt{I}(a) : \mathtt{S} \to \mathtt{t}(a)$ *(cf.* $\mathcal{I}$*)*
- $\mathtt{F} \subseteq 2^{\mathtt{S}}$ *is a collection of predicates on* $\mathtt{S}$ *identifying fair paths;*
- $\mathtt{S}_0 \subseteq 2^{\mathtt{S}}$ *is a collection of predicates on* $\mathtt{S}$ *identifying the initial state.*

A *Path* is a infinite sequence of states such that $\forall_i \cdot (s_i, s_{i+1}) \in \mathtt{R}$. A path p is $\mathtt{F}$*-fair* iff for each $f \in \mathtt{F}$ there are infinitely many states on p which satisfy f.

A.3 Satisfaction of interactor axioms in SMV

In order to be able to express our interactor specifications in SMV we must be able to define satisfaction of the interactor axioms in terms of the SMV interpretation structure.

The immediate obvious problem is the lack of the notion of action in SMV. To solve this we introduce the occurrence operator ($>_{\mathsf{ac}}$) (cf. Fiadeiro & Maibaum 1991).

For an interactor interpretation structure $P = (\mathcal{W}, \mathcal{M}, \mathcal{I}, \mathcal{P}, \mathcal{O})$ and a state $w \in \mathcal{W}$ the satisfaction of the occurrence operator is defined as follows:

- $P, w \models (>_{\mathsf{ac}} e)$ iff exists a state $k \in \mathcal{W}$ such that $\mathcal{M}(e)(k) = w$
- $P, w \models (>_{\mathsf{ac}} [])$ iff $w = \mathcal{M}([])$

i.e. $(>_{\mathsf{ac}} e)$ is the strongest post-condition for action e.

What the $(>_{\mathsf{ac}} e)$ operator does is to allow us to reference the occurrence of an action from within a state. This will enable us to eliminate the need for the explicit labeling of transitions, thus allowing us to express the interactor axioms in SMV.

We now show how this is done, starting with the definition of modal axioms satisfaction:

$$P, w \models prop([e]a_1, .., [e]a_n, a_k, .., a_l))$$

Using the definition of satisfaction we can rewrite this to

$$\forall_{w' \in \mathcal{W}} \cdot w' = \mathcal{M}(e)(w) \rightarrow prop(\mathcal{I}(a_1)(w'), .., \mathcal{I}(a_n)(w'), \mathcal{I}(a_k)(w), .., \mathcal{I}(a_l)(w))$$

we know that if $w' = \mathcal{M}(e)(w)$ then $P, w' \models (>_{\mathsf{ac}} e)$, so we can rewrite to

$$\forall_{w' \in \mathcal{W}} \cdot P, w' \models (>_{\mathsf{ac}} e) \rightarrow prop(\mathcal{I}(a_1)(w'), .., \mathcal{I}(a_n)(w'), \mathcal{I}(a_k)(w), .., \mathcal{I}(a_l)(w))$$

We have eliminated the use of $\mathcal{M}$, so we have eliminated the need for transition labels in the finite state machine in order to express this axiom. This means we can now express the axiom in the SMV interpretation structure.

In order to do this we start by defining, for an interactor with attributes A, mapping $\mathsf{t} : \mathsf{A} \rightarrow \mathsf{T}$, actions E, and an interpretation structure $(\mathcal{W}, \mathcal{M}, \mathcal{I}, \mathcal{P}, \mathcal{O})$, a SMV module with attributes $\mathsf{A} \cup \{>_{action}\}$, and mapping $\mathtt{u} : \mathsf{A} \cup \{>_{action}\} \rightarrow \mathsf{T} \cup \{\mathtt{T}\}$ (where $\mathtt{T}$ is the type for actions) such that $\mathtt{u}(a) = \mathsf{t}(a)$ for all $a \in \mathsf{A}$ and $\mathtt{u}(>_{action}) \in \mathtt{E}$.

The attribute $>_{action}$ is a special attribute which identifies the action that makes $>_{\mathsf{ac}}$ true in each state. Note that we are enforcing that each state can only be reached by one action. This might make it necessary to have bigger state machines, but allows us to keep track of which action causes which transition.

Let $M = (\mathtt{S}, \mathtt{R}, \mathtt{I}, \mathtt{F}, \mathtt{S}_0)$ be the interpretation structure for the SMV module above. We can now finally relate the satisfaction of the interactor axioms to the SMV modules interpretation structures. For every axiom

$$[e]prop(a'_1, .., a'_n, a_k, .., a_l)$$

and for every state $s \in \mathtt{S}$ the axiom is satisfied by M in s if for all pairs $(s, s') \in \mathtt{R}$:

$$\mathtt{I}(>_{action})(s') = e \rightarrow prop(\mathtt{I}(a_1)(s'), .., \mathtt{I}(a_n)(s'), \mathtt{I}(a_k)(s), .., \mathtt{I}(a_l)(s))$$

This can be written in SMV as the axiom (remember that `next` indicates the next state in the transition):

```
next(action)=e -> prop(next(a_1),..,next(a_n),a_k,..,a_l)
```

Let us recall Axiom 8 from interactor MCP in Section 3:

$$[\mathsf{enterAC}]\mathsf{pitchMode}' = \mathsf{ALT\_CAP} \wedge \neg \mathsf{ALT}'$$

According to the rule above, this modal axiom is translated to the SMV axiom:

```
next(action)=enterAC -> next(pitchMode)=ALT_CAP & !next(ALT)
```

The initialisation axioms

$$[]prop(a_1, .., a_n)$$

are satisfied in all $s \in S_0$ such that:

$$prop(\mathtt{I}(a_1)(s), .., \mathtt{I}(a_n)(s)) \in \mathsf{S}_0$$

This is written in SMV thus:

```
INIT prop(a_1,..,a_n)
```

Permission axioms undergo a similar process as for modal axioms and we obtain that for every axiom

$$\mathsf{per}(e) \rightarrow prop(a_1, .., a_n)$$

in our interactors specification, and for every pair $(s, s') \in \mathtt{R}$

$$\mathtt{I}(>_{\mathtt{action}})(s') = e \rightarrow prop(\mathtt{I}(a_1)(s), .., \mathtt{I}(a_n)(s))$$

The translation into SMV code follows as for the modal case above.

Finally we have the obligation axioms. The axiom

$$prop(a_1, .., a_n) \rightarrow \mathsf{obl}(e)$$

asserts that every time proposition $prop(a_1, .., a_n)$ is true then at some time in the future action e must happen. This cannot be expressed in R as there we only deal with pairs of states.

In order to express obligation we have to use fairness constraints. For each action $e \in \mathtt{R}$ we define a new boolean attribute $\mathsf{o}bl_e$ in the module and we include the predicate $\neg \mathsf{o}bl_e$ in F. Thus stating that $\mathsf{o}bl_e$ cannot be true indefinitely. Then, for each obligation axiom on an action e and for every pair $(s, s') \in \mathtt{R}$ we require that

$$\begin{array}{c} \mathtt{I}(>_{\mathtt{action}})(s') \neq e \rightarrow (\mathtt{I}(\mathsf{o}bl_e)(s') = (prop(\mathtt{I}(a_1)(s), .., \mathtt{I}(a_n)(s)) \vee \mathtt{I}(\mathsf{o}bl_e)(s)) \\ \wedge \\ \mathtt{I}(>_{\mathtt{action}})(s') = e \rightarrow \neg \mathtt{I}(\mathsf{o}bl_e)(s') \end{array}$$

That is, if the next action is not e, then there will be an obligation to perform e if either the condition $prop(\mathtt{I}(a_1)(s), .., \mathtt{I}(a_n)(s))$ is true or there is already an obligation to perform the action. If the action is e, then any obligation regarding e is dismissed.

Since $\mathsf{o}bl_e$ (the obligation to perform e) cannot be true indefinitely, we guarantee that, once an obligation arises, it will eventually be fulfilled.

The last point to address has to do with the execution models of both languages. While we understand interactors as progressing independently from each other (modulo explicitly specified synchronisations), SMV modules all execute in simultaneous steps. To solve this mismatch, we introduced the possibility of stuttering by adding a special action `nil` to each module, and requiring that for every two states s and s':

$$(s, s') \in \mathtt{R}\ if\ (\mathtt{I}(>_{action})(s') = \mathtt{n}il \wedge \forall_{a \in \mathtt{A}} \cdot \mathtt{I}(a)(s) = \mathtt{I}(a)(s')$$

B Checkable specification

This is the translatable version of the interactor specification for the MCP. The step of creating the non parametrised versions of the Dial interactor has been done manually. This is a simple syntactic transformation that will be implemented in the next version of the compiler.

```
# MCP example
types
  PitchModes = {VERT_SPD, IAS, ALT_HLD, ALT_CAP}
  Altitude   = {0, 1, 2, 3, 4}
  Velocity   = {0, 1, 2, 3, 4}
  ClimbRate  = {-1, 0, 1}

interactor airplane
attributes
  altitude: Altitude
  airSpeed: Velocity
  climbRate: ClimbRate
actions
  fly
axioms
  (altitude>0 & altitude<4) -> [fly] \
          ((altitude'>=altitude - 1 & altitude'<=altitude + 1) & \
           (altitude'<altitude -> climbRate'<0) & \
           (altitude'=altitude -> climbRate'=0) & \
           (altitude'>altitude -> climbRate'>0))
  altitude=0 -> [fly]
          ((altitude'>=altitude & altitude'<=altitude + 1) & \
           (altitude'=altitude -> climbRate'=0) & \
           (altitude'>altitude -> climbRate'>0))
  altitude=4 -> [fly]
          ((altitude'>=altitude - 1 & altitude'<=altitude) & \
           (altitude'<altitude -> climbRate'<0) & \
           (altitude'=altitude -> climbRate'>=0))
fairness
  !action=nil

interactor dialAltitude
attributes
  needle: Altitude
actions
  set(Altitude)
axioms
```

```
  [set(v)] needle'=v

interactor dialVelocity
attributes
  needle: Velocity
actions
  set(Velocity)
axioms
  [set(v)] needle'=v

interactor dialClimbRate
attributes
  needle: ClimbRate
actions
  set(ClimbRate)
axioms
  [set(v)] needle'=v

interactor main
includes
  airplane via plane
  dialClimbRate via crDial
  dialVelocity via asDial
  dialAltitude via ALTDial
attributes
  pitchMode: PitchModes
  ALT: boolean
actions
  enterVS, enterIAS, enterAH, enterAC, toggleALT
axioms
  [crDial.set(t)] pitchMode'=VERT_SPD & ALT'=ALT
  [asDial.set(t)] pitchMode'=IAS & ALT'=ALT
  [ALTDial.set(t)] pitchMode'=pitchMode & ALT'
  [enterVS] pitchMode'=VERT_SPD & ALT'=ALT
  [enterIAS] pitchMode'=IAS & ALT'=ALT
  [enterAH] pitchMode'=ALT_HLD & ALT'=ALT
  [toggleALT] pitchMode'=pitchMode & ALT'=!ALT
  [enterAC] pitchMode'=ALT_CAP & !ALT'
  per(enterAC) -> (ALT & (ALTDial.needle - plane.altitude)<=2)
  (pitchMode=ALT_CAP & plane.altitude=ALTDial.needle) -> obl(enterAH)
  pitchMode=VERT_SPD -> plane.climbRate=crDial.needle
  pitchMode=IAS -> plane.airSpeed=asDial.needle
  pitchMode=ALT_HLD -> plane.climbRate=0
  [] crDial.needle=1 & ALTDial.needle=3 & plane.altitude=0 & \
     pitchMode=VERT_SPD & ALT
fairness
  !action=nil
test
  AG(ALT -> AF((pitchMode=ALT_HLD & plane.altitude=ALTDial.needle))
```