Model Checking Interactor Specifications

José C. Campos[1,2,*] and Michael D. Harrison[1]
[1] *Human-Computer Interaction Group, The University of York, UK*
[2] *Departamento de Informática, Universidade do Minho, Portugal*

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, 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 provide an opportunity to analyse problems that arise in such systems. In this paper we consider the role that an interactor based specification has as a partial model of an interactive system, so that mode consequences can 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 in raising questions that have to be addressed in a broader context of analysis.

Keywords: software verification, interactive systems, automation surprise, interface mode confusion, model checking, interactor based specifications

1. Introduction

This paper is primarily about using automated reasoning techniques (more specifically model checking) during interactive systems design. Model checking is a verification technique which is being used with success in hardware and protocol verification. We believe it can also play an important role in the development of safety critical interactive systems, where the consequence of failure can become unacceptable.

More specifically, 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". An automation surprise happens when the system behaves differently from the expectations of the user. A particular class of such surprises, interface mode changes, 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 offers an opportunity

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

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 and 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 and Paternò, 1990; Duke and Harrison, 1993) has 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. The main advantage of the interactor concept is that it allows for the specification of both system state and behaviour, and user interface presentation and behaviour, in the same framework. 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 — a unit of human activity, carried out to achieve a specific goal (Newman and Lamming, 1995) — and user interface mode — which defines how the system responds to input, and how the state is represented in the output — involve a broad range of concerns, from hardware restrictions to more subjective human factors issues. In (Campos and 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.

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 im-

plications for the safety of the aircraft 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 note that the analysis is not primarily concerned with the behaviour of the system by itself, but with the interaction between the system and its users. This means that input from human-factors specialists can be incorporated in the verification process, further stressing that the analysis does not concern simply how the behaves, rather how system and user behave together.

Two basic types of automated verification techniques can be identified: model checking and theorem proving. Each technique has its own strong points. Model checking is usually best at verifying properties related to the temporal behaviour of the system, while theorem provers are best at verifying properties related to the system's state. As the title of this paper indicates, we will be using the former. This is not to say that we propose model checking as the technique of choice for the analysis of interactive systems. Our view is that both techniques can be useful. The choice of which to use will depend on the particular aspect of the system being analysed. In (Doherty et al., 2000) we show how theorem proving can be useful to reason about the relation between the system state and the proposed user interface. In (Campos and Harrison, 1999) we show how both verification techniques can be integrated in to a coherent verification process.

In Section 2 we discuss the role that verification should play during design, and introduce the interactor language used in the following sections. In section 3 we describe a tool that enables us to check interactors in SMV. In Section 4 we use the interactor language to build a model of the Mode Control Panel (MCP) of the aircraft. The MCP is one element of the interface between the pilot and the aircraft 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 5 we show how to go about model checking the resulting specification. In Section 6 we compare our work with other approaches to the verification of software requirements in general, and of interactive systems in particular. Finally in Section 7 we analyse the results of the case study, and draw some conclusions.

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 and verification technique that we shall be using.

2.1. The Role of Verification in Interactive Systems Design

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 used in practice. At this level, approaches tend to be based around the development of a specification of the entire system, which can then be analysed (cf. Paternò, 1995, Heitmeyer et al., 1998), or even on the reverse engineering of the actual system implementation (cf. Bumbulis et al., 1996). See (Campos and Harrison, 1997) for a review of current approaches to the automated verification of interactive systems. In general, this holistic style of approach means that design decisions may have to be remade with costly consequences if a problem is found. Additionally, for systems as complex as interactive systems are, it becomes difficult, if not impossible, to analyse specifications which represent whole systems (cf. Campos and Harrison, 1998).

At this stage, it should be mentioned that the aim of formal verification is not proving a system correct. Correctness assumes some absolute measure of quality against which the specification can be verified. Trying to define it, we are faced with the problem of its own correctness. Hence, as Henzinger (1996) states:

> The only sensible goal of formal methods is to detect the presence of errors and to do so early in the design process. Indeed, "falsification" would be a more appropriate name for the endeavor called "verification."

In (Campos and Harrison, 1998) we argue that in order to explore the full potential of formal verification, we must move the verification step into the development process. Instead of being used as a sanity check on the end result of design, verification should be used as a guide in the process of design decision making (cf. the *"verify-while-develop"* paradigm, de Roever, 1998). This can be achieved by using verification techniques on partial models that highlight the specific concerns of different development stages (cf. Fields et al., 1997). Once a specific artifact of the system is identified which needs to be analysed, a model can be built that highlights the aspects of the artifact which are of interest. The results of the analysis of such a model can them be fedback to the design process (see Figure 1). This process can be applied repeatedly.

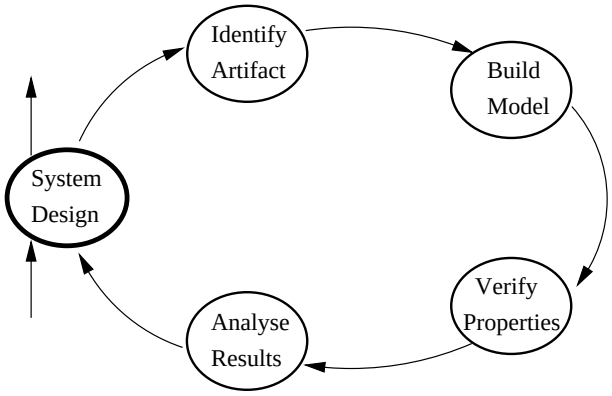

Figure 1. Verification process

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 require 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 to 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

Traditional specification languages do not usually cater for the specificities of interactive systems. Interactors (Faconti and Paternò, 1990; Duke and Harrison, 1993) have been proposed as a structuring concept for such a task.

The interactor model, as developed by the York group (Duke and Harrison, 1993) (see Figure 2), is that of an object which, besides interacting with the environment through events, is capable of rendering (part of) its state into some presentation medium (cf. rho in Figure 2). The model does not prescribe a specification notation for the description of interactor state and behaviour. Instead, it acts as a mechanism for structuring the use of standard specification techniques in the context of interactive systems specification.

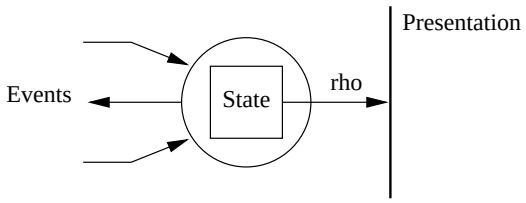

Figure 2. York Interactor

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

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(T)
attributes
needle: T

This declaration introduces an interactor named dial. This interactor has only one attribute (needle), and the type of the attribute is T (the range of values in the dial). Note that here the type is a parameter of the interactor. This means that it will be possible to have dials with different ranges.

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(T)

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).

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 $[\_]\_$: $[ac]expr$ is the value of $expr$ in the state resulting from the occurrence of action ac;
- the special reference event []: if $[]expr$ then $expr$ is true 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 (*incr*) which increments the value of attribute needle above. In Structured MAL we would write:

$$(needle = aux) \rightarrow [incr](needle = aux + 1)$$

where *aux* is an auxiliary variable introduced to *carry* the value of *needle* into the next state (after *incr*).

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

$$([incr]needle) = needle + 1$$

Which reads: the value of *needle* after *incr* equals the current value of *needle* plus one.

To further simplify more complex axioms where the modal operator is applied to more than one attribute, we can factor out the operator and use priming to indicate which attributes are affected by it. Hence, we write the axiom above as:

$$[incr](needle' = needle + 1)$$

We will omit the parentheses whenever the scope of the modal operator can be inferred.

The behaviour of set can be defined by the following axiom:

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

The axiom reads: the value of needle after action set(v) is 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 3.

The composition of interactors is done via inclusion as in (Ryan et al., 1991). To use a dial in some other interactor we would write:

```
interactor Panel
includes
  dial(Range) via speedDial
...
```

where speedDial becomes the name of a particular instance of dial in the context of interactor Panel. We shall assume that all actions and attributes of an interactor are always accessible to other interactors

interactor dial(T)
attributes
vis needle: T
action
vis set(T)
axioms
(1) [set(v)] needle'=v

Figure 3. Simple dial interactor

that include it. Hence, to initialise the needle of speedDial to zero, we could place the following axiom in interactor Panel:

[] speedDial.needle=0

In this case we are assuming the existence of type Range. Types can be defined by enumeration or as subranges of integer:

types
$T_1 = \{a, b, c\}$
$T_2 = 0..10$

In order to make the checking of specifications possible, we will represent types as enumerations of the "key values" of each type.

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:

- $\mathsf{per}(ac) \rightarrow guard$ — action ac is permitted only if $guard$ is true;
- $cond \rightarrow \mathsf{obl}(ac)$ — 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 components can be reused.

The next section gives introductions to SMV and CTL as well as a detailed description of how MAL descriptions can be translated into SMV. The translation is summarised in Table I. A reader familiar with SMV and CTL, prepared to believe that the translation works,

and more interested in strategies for proving properties of interactive behaviour may skip to Section 4, pausing briefly at Table I.

3. Model Checking Interactors

3.1. SMV

Model checking was originally proposed as an alternative to the use of theorem provers in concurrent program verification (Clarke et al., 1986). The basic premise was that a finite state machine specification of a system can be subject to exhaustive analysis of its entire state space to determine what properties hold of the system's behaviour. Typically the properties are expressed in some temporal logic that allows reasoning over the possible execution paths of the system (see Figure 4[1]). In this context, the possible execution paths are interpreted as alternative futures.

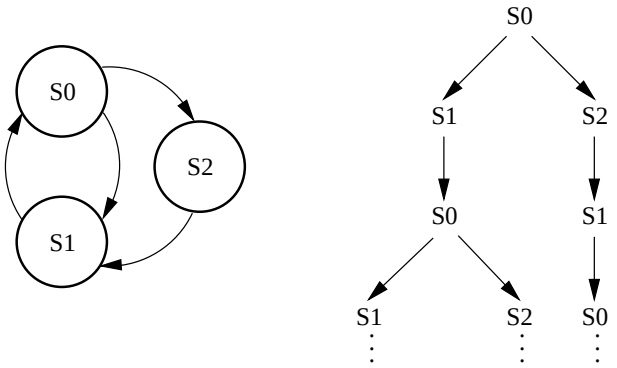

Figure 4. Execution paths of a finite states machine

By using an algorithm to perform the state space analysis, the two main drawbacks of theorem provers were avoided:

- the analysis is fully automated (as opposed to theorem provers' high reliance on the skills of its users);
- the validity of a property is always decidable (as opposed to theorem provers' undecidability problems).

A main drawback of Model Checking has to do with the size of the finite state machine needed to specify a given system: useful specifications may generate state spaces so large that it becomes impractical

[1] Figure adapted from (Abowd et al., 1995).

```
MODULE blink
VAR
  light: boolean;
INIT
  light=0
TRANS
  next(light) = !light
```

Figure 5. An example SMV module

to analyse the entire state space. Hence, theoretically decidable systems may become undecidable in practice. The use of Symbolic Model Checking somewhat diminishes this problem. Avoiding the explicit representation of states, state spaces as big as 10^{20} states may be analysed (Burch et al., 1990).

In this paper, the symbolic model checker SMV (McMillan, 1993) will be used.

3.1.1. *The SMV input language*

An SMV specification is a collection of modules. Each module 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 axioms). In order to express MAL axioms in SMV, it is only necessary to consider a subset of the SMV input language. The ASSIGN declaration and case expressions, in particular, will not be used.

Figure 5 shows an example SMV module. Attributes are declared in clause `VAR`. In this case, there is only one attribute (`light`) and its type is `boolean`. Clause `INIT` defines the initial state of the module. This is done with an axiom on the values of the attributes. In this case the axiom states that in the initial state attribute `light` is false (zero representing false). Clause `TRANS` defines how the state evolves. In this case the attribute will repeatedly toggle between true and false, every time a state change happens. Note the use of `next` to reference the next state. Axioms in `TRANS` clauses are written using a temporal logic where `next` is the only temporal operator. The usual propositional operators are also present: `!` stands for logical not, `&`/`|` stand for logical and/or respectively, `->` and `<->` stand for implication and equivalence. In this case the axiom reads: the value of attribute `light` in the next state will be the negation of the value of attribute `light` in the current state.

The complete list of declarations used is now presented:

- `VAR` — allows the declaration of the variables that define the module's state. The types associated with the attributes can be either boolean, an enumeration, an array, or another module. The use of arrays will not be addressed in this paper (see Campos, 1999).
- `INIT` — allows the definition of the initial state of the module. This is done using propositional formulae on the module's attributes (the use of `next` is not allowed).
- `TRANS` — allows the definition of the behaviour of the module. This is done using temporal formulae. The operator `next` is used to refer to the next state.
- `INVAR` — allows the specification of invariants over the state. These invariants must be evaluated *inside the state* so, as for `INIT`, they are written using propositional formulae.
- `FAIRNESS` — allows the specification of fairness constraints. The behaviour of the module will have to obey the fairness constraints infinitely often. Fairness constraints can be temporal formulae over paths (CTL formulae), or simply propositional (*inside the state*) formulae. In practice, only propositional formulae tend to be useful/meaningful. Hence, only those will be considered.
- `SPEC` — allows the definition of a CTL formula to be checked.

3.1.2. *CTL*

As just said, in the case of SMV, CTL (Computational Tree Logic — Clarke et al., 1999) is used to express properties about the behaviour of the system. For a complete description of the CTL syntax see (Clarke et al., 1999). Besides the usual propositional logic connectives, CTL allows for operators over paths:

- A – for all paths (universal quantifier over paths);
- E – for some path (existential quantifier over paths);

and over states in a path:

- G – for all states in the path (universal quantifier over states in a path);
- F – for some state in the path (existential quantifier over states in a path);
- X – for the next state in the path;

– U – some property will hold in the path until some other property holds.

These operators allow us to express concepts such us:

– universally: $AG(p)$ – p is universal (for all paths, in all states, p holds);

– inevitability: $AF(p)$ – p is inevitable (for all paths, for some state along the path, p holds);

– possibility: $EF(p)$ – p is possible (for some path, for some state along that path, p holds).

3.2. From Interactors to SMV

For model checking of interactor models in SMV to be possible, first the models must be expressed in the SMV specification language. Unlike Abowd et al.'s approach, where only a single PPS is used, this implies representing models composed of multiple interactors. For each interactor, its state and behaviour must be expressed in SMV.

In this section an algorithm is developed to carry out this translation. The section ends with a brief description of a tool that implements the translation.

3.2.1. *Expressing interactor state in SMV*

SMV has been previously used in the verification of interactive system specifications by Abowd et al. (1995). Their approach, however, relies on a simple propositional production system written in Action Simulator (Monk and Curry, 1994). With interactors we build specifications compositionally. An SMV module is similar to an interactor in that is also has a state (a collection of attributes), and axioms describing how the state evolves. These similarities raise the possibility of representing interactors as SMV modules.

State attributes in SMV can be declared as booleans or as having an enumerated type. This means that restrictions will have to be enforced in the types used in interactors. If, for example, a variable is defined as having type *nat*, its type will have to be restricted to an appropriate subrange of *nat* before translation to SMV is carried out. Hence, it is possible to define a translation rule:

Translation Rule 1. (Attributes)

attributes $\mathsf{a}_1 : \mathsf{T}$ *translates to:* `VAR a1: T;`

whenever `T` is a valid SMV type.

The **includes** clause is used to allow interactors to have other interactors as part of their state. This notion has a direct counterpart in S-MV: modules can have instances of other modules as part of their state. Hence, representing interactor inclusion is straightforward. Instances of included interactors become variables whose types are the appropriate SMV modules resulting from the translation of the interactors. The translation rule for interactor inclusion is:

Translation Rule 2. (Interactor inclusion)
includes $\mathsf{i}_{\mathsf{name}}$ **via** i_1 *translates to:* `VAR i1: iname;`

where `iname` is the SMV module resulting from the translation of interactor $\mathsf{i}_{\mathsf{name}}$.

The concept of importing does not exist in SMV. However, importing clauses can be eliminated from an interactor based model by substituting them by the definition of the imported interactors. Additionally, SMV modules cannot be parameterised by types (they can have attributes as parameters, although this feature will not be used since it is not present in the interactor language). Again, it is possible to eliminate type parameters from interactor based models. This is done by instantiating each parameterised interactor with the types actually used as parameters. This means that a parameterised interactor will generate many SMV modules (as many as the number of times it is instantiated with different parameters). Note that both these transformations can easily be done automatically.

So, with appropriate restrictions it is possible to represent the state of an interactor in the state of an SMV module. The major problem remains of expressing the behaviour of the interactors in SMV. The remainder of this section deals mainly with showing how a translation from MAL to SMV axioms can be deduced. This fulfils two objectives: it provides the translation algorithm needed for each type of axiom, and it guarantees the correctness of the translation process. The approach taken follows from (Fiadeiro and Maibaum, 1991).

3.2.2. *Expressing interactor behaviour in SMV*

This section discusses how interactor axioms can be expressed using the SMV input language. Five types of interactor axioms can be identified:

- invariants — these are formulae that do not involve any kind of event (i.e., simple propositional formulae). They must hold for all states of the interactor.
- initialisation axioms — these are formulae that involve the reference event ($[]$). They define the initial state of the interactor.

- modal axioms — these are formulae involving the modal operator. They define the effect of actions in the state of the interactor.
- permission axioms — these are deontic formulae involving the use of per. They take the shape $\mathsf{per}(ac) \rightarrow cond$: action ac is permitted only when the propositional formula $cond$ holds.
- obligation axioms — these are deontic formulae involving the use of obl. They take the shape $cond \rightarrow \mathsf{obl}(ac)$: action ac becomes obligatory when the propositional formula $cond$ holds.

In the discussion that follows each of these types of axioms will be addressed. To help in the presentation, the notation $p(expr_1, .., expr_n)$ will be used to denote a formula on expressions $expr_1$ to $expr_n$ using propositional operators only. Note, however, that expressions $expr_1$ to $expr_n$ need not necessarily be propositional.

3.2.2.1. *Invariants* These are axioms $p(a_1, .., a_n)$ such that a_1 to a_n are interactor attributes. Invariants must hold in all states of the model. This has a direct counterpart in SMV: the `INVAR` clause. The translation rule for invariants is then:

Translation Rule 3. (Invariants)
$\mathsf{prop}(\mathsf{a}_1, .., \mathsf{a}_n)$ *translates to:* `INVAR prop(a1,..,an)`

3.2.2.2. *Initialisation axioms* These are axioms that take the shape $[]p(a_1, .., a_n)$. They are used to define the initial state. Again this has a direct counterpart in SMV: the `INIT` clause. Hence, initialisation axioms are translated by removing the reference events and placing the resulting axioms in INIT clauses:

Translation Rule 4. (Initialisation axioms)
$[]\mathsf{prop}(\mathsf{a}_1, .., \mathsf{a}_n)$ *translates to:* `INIT prop(a1,..,an)`

3.2.2.3. *Modal axioms* These axioms are used to specify the effect of actions in the state. They take the shape $p([e]a_1, .., [e]a_g, a_h, .., a_n)$.

As stated above, there is no direct counterpart for this in SMV. However, Fiadeiro and Maibaum (1991) have shown how it is possible to reason about the temporal properties of the normative behaviours of deontic specifications. Since it is the normative behaviours of the interactor models that are of interest in what follows, we can assume that all permissions will be respected, and all obligations will be fulfilled. Therefore it is possible to make use of results from (Fiadeiro and Maibaum, 1991).

First the occurrence operator $>_{action}$ must be introduced. This corresponds to the $>_{\mathtt{T}}$ operator in (Fiadeiro and Maibaum, 1991). In a given state, this operator is used to signal that the state has been reached by the occurrence of some specific action. Hence, $>_{action}\ e$ holds in a state when e is the action that causes the transition to that state.

Note that, instead of taking Abowd et al.'s approach of encoding information about the next action, information relating to the previous action is encoded (the action corresponding to the last transition that occurred). This avoids the problem of duplicated initial states described in (Campos, 1999). In this case states are duplicated when they can be reached using different actions (see Figure 6).

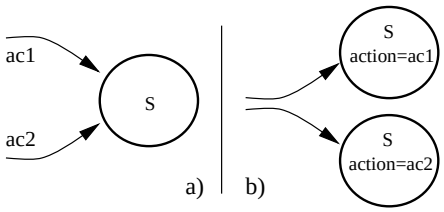

Figure 6. State duplication

Using the occurrence operator it becomes possible to reason about actions in SMV. From (Fiadeiro and Maibaum, 1991) it is known that[2]:

$$(\rightarrow [e]p) \Rightarrow ((>_{action}\ e) \rightarrow p) \tag{1}$$

The equation reads: if p holds after e, then p must hold in all states where $>_{action}\ e$ holds (all states that can be reached by performing e).

In Equation 1, the modal operator is being applied to whole propositions. However, in this paper the modal operator works at the level of the attributes. Hence, the following equation will be used instead:

$$(([e]a_1) = a_2) \Rightarrow ((>_{action}\ e) \rightarrow (a_1 = Y(a_2))) \tag{2}$$

That is, if the value of a_1 after e equals the current value of a_2 (the value before e), then in every state reached by performing e, the value of a_1 will equal the value of a_2 before e had happened (the previous value of a_2 — hence the use of Y, the *previous* operator).

Since in SMV the Y operator does not exist, we rewrite the equation using the `next` operator:

$$(([e]a_1) = a_2) \Rightarrow (\mathtt{next}(>_{action}\ e) \rightarrow (\mathtt{next}(a_1) = a_2)) \tag{3}$$

[2] In this context $a \Rightarrow b$ means that $a \rightarrow b$ for all states in the model.

To write this equation in SMV it is first necessary to model the occurrence operator. The operator will be modeled by a state attribute ($>_{\texttt{action}}$) indicating the event for which the operator holds true. Hence, $>_{action}\ e$ becomes $>_{\texttt{action}}=\texttt{e}$. The type of this attribute will be an enumeration of all the possible events.

It is now possible to write the translation rule for modal axioms:

Translation Rule 5. (Modal axioms)
$\mathsf{prop}([\mathsf{e}]\mathsf{a}_1, .., [\mathsf{e}]\mathsf{a}_g, \mathsf{a}_h, .., \mathsf{a}_n)$
translates to:

```
TRANS
next(>action)=e -> prop(next(a1),..,next(ag),ah,..,an)
```

i.e., modal axioms are translated into axioms that test whether the next action is the appropriate one, and assert the condition using `next` to reference the next state.

This translation rule guarantees that all behaviour specified by modal axioms will be present in the SMV model. The opposite is not true (note that Equation 3 is an implication, not an equivalence). SMV generates all state transitions that do not infringe the given axioms. Hence, it is possible to have some behaviours that are present, not because they were explicitly specified, but because no axioms were given which state how the system behaves in some particular circumstances. This is actually a useful feature, since it allows for nondeterminism and underspecified models.

3.2.2.4. *Permission axioms* These axioms are used to restrict action permission to some specific conditions. They take the shape $\mathsf{per}(e) \rightarrow p(a_1, .., a_n)$.

As for modal operators, in SMV there is no notion of permission. However, from (Fiadeiro and Maibaum, 1991), it can be deduced that formulae of the form

$$\mathsf{per}(e) \rightarrow cond$$

lead to

$$\texttt{next}(>_{action}\ e) \rightarrow cond.$$

This can be used to define the translation rule for permission axioms:

Translation Rule 6. (Permission axioms)
$\mathsf{per}(\mathsf{e}) \rightarrow \mathsf{prop}(\mathsf{a}_1, .., \mathsf{a}_n)$
translates to:

```
TRANS next(>action)=e -> prop(a1,..,an)
```

The behaviour of the SMV model is guaranteed not to infringe the SMV axioms. Hence, this translation rule guarantees that all (MAL

level) permission conditions, set for some action, will have to be met (in the corresponding SMV model) in order for the state transition associated with that action to take place.

3.2.2.5. *Obligation axioms* These axioms are used to assert the obligation of performing some action. They take the shape $p(a_1, .., a_n) \rightarrow obl(e)$.

Once more there is no direct way to express this in SMV, but from (Fiadeiro and Maibaum, 1991) it can be deduced that formulae of the form

$$cond \rightarrow \mathsf{obl}(e)$$

lead to

$$cond \rightarrow F(\mathtt{next}(>_{action}\ e))$$

with F the *sometime in the future* operator.

It is not possible to express this last equation directly as an SMV axiom. The SMV input language allows reference to the current and next state only, and the equation above makes reference to some arbitrary state in the future. The only way to influence future states of the system is by fairness conditions. Since a fairness condition needs to hold infinitely often, if the formula $\mathtt{next}(>_{\mathtt{action}}) = e$ is placed as a fairness condition, then eventually the action will happen. This strategy would require formulae to be added to, and removed from, the set of fairness conditions as obligations are successively raised and fulfilled during the execution of the state machine. Unfortunately, fairness is defined by a static set of formulae in the SMV text. One way to get around this is to use a boolean flag signaling when a specific obligation is raised. Hence, instead of adding the formula to the set of fairness conditions, the flag is set to true. By only unsetting the flag when the action happens, and asserting, as a fairness condition, that the flag must infinitely often be false, it is guaranteed that the action will eventually happen once an obligation for it is raised.

It is now possible to enumerate the rules for the translation of an interactor into an SMV module. This is done in Table I. On the left hand side of the table, the various interactor expressions are listed. The right hand side gives the corresponding SMV expressions The last rule gives the translation for obligation axioms, following the reasoning just described.

3.2.3. *Some final comments regarding the translation*
As stated initially, the discussion above has only considered actions with no parameters. As with parameterised interactors, it is possible to eliminate a parameterised action automatically by substituting it by

Table I. Translation from interactors to SMV

Interactor	**SMV Module**
interactor name	`MODULE name`
attributes $a : \{v_1, .., v_n\}$	`VAR a : {v1, .., vn}` `VAR` $<_{\texttt{action}}$`: {ac1, .., acn};`
interactor inclusion: **includes** i_{name} **via** i_1	`VAR i1: iname;`
invariants: $prop(a_1, .., a_n)$	`INVAR prop(a1, .., an)`
initialisation axioms: $prop([]a_1, .., []a_n)$	`INIT prop(a1, .., an)`
modal axioms: $prop([e]a_1, .., [e]a_g, a_h, .., a_n)$	`TRANS next(`$<_{\texttt{action}}$`) = e ->` `prop(next(a1), .., next(ag), ah, .., an)`
permission axioms: $per(e) \rightarrow prop(a_1, .., a_n)$	`TRANS next(`$<_{\texttt{action}}$`) = e ->` `prop(a1, .., an)`

obligation axioms:
$prop(a_1, .., a_n) \rightarrow obl(e)$
	`VAR oble : boolean;`
	`INIT !oble`
	`FAIRNESS !oble`
	`TRANS next(`$<_{\texttt{action}}$`)!= e ->`
	`next(oble) = (prop(`$\texttt{a}_1, .., \texttt{a}_\texttt{n}$`) | oble)`
	`TRANS next(`$<_{\texttt{action}}$`) = e->!next(oble)`

a set of actions, one for each possible combination of the parameters' values. Parameterised actions can appear in three types of axioms (note that only universally quantified variables are accepted as parameters):

- modal axioms — Axioms are repeated as many times as needed, replacing the parameterised actions by the appropriate values.
- permission axioms — As for modal axioms, permission axioms are repeated as many times as needed.
- obligation axioms — Any of the actions generated according to the reasoning above will discharge the obligation.

Another feature of the interactor language that has not been mentioned is type definition: the possibility of giving names to enumerated types. This is not possible in SMV but, once again, type names can be eliminated by substituting all the occurrences of a type name by its definition.

Two additional clauses were added to the language. Clause **fairness** allows the definition of fairness constraints, clause **test** allows the definition of CTL formulae to be checked. Clause **test** should only be used

in the master interactor of a model. Additionally, this interactor should be called main.

All of the above focus on translating each interactor into an SMV module. It is assumed that the semantics of combining interactors and of combining SMV modules are the same. This is true except for one problem. SMV modules work in lock-step. Whenever a module performs a transition all modules must perform a transition. Theoretically it is possible to overcome this using processes. However, when attributes of two processes are related, the progress of each of the processes becomes locked to one another. Since the desired semantics for interactors is that they can evolve independently, subject to explicit synchronisations, a way was needed to model this in SMV. This was done by allowing stuttering in the SMV modules. That is, modules can perform state transitions in which no actions actually happen. To this end, a special action `nil` is introduced along with axioms stating that this action does not change the state of the module. This allows a module to engage in an event, while another module does nothing (or, more precisely, it performs a stuttering step).

3.3. The tool

A tool to implement the translation just described has been implemented in Perl (see Wall et al., 1996). The Perl script (`i2smv`) works by reading a interactor model, and building an intermediate representation of that model. The intermediate representation is then manipulated by performing the following steps:

1. eliminate interactor importing;
2. eliminate type parameters from interactors;
3. eliminate parameters from actions;
4. eliminate type names;
5. create the stuttering action;
6. generate SMV code according to the translation in Table I.

The tool acts as a filter, receiving interactor code at the input and generating SMV code at the output. A file can also be provided for the input. Supposing an interactor model was written into file `model.i`, the command:

```
indy033:~>  i2smv model.i | smv
```

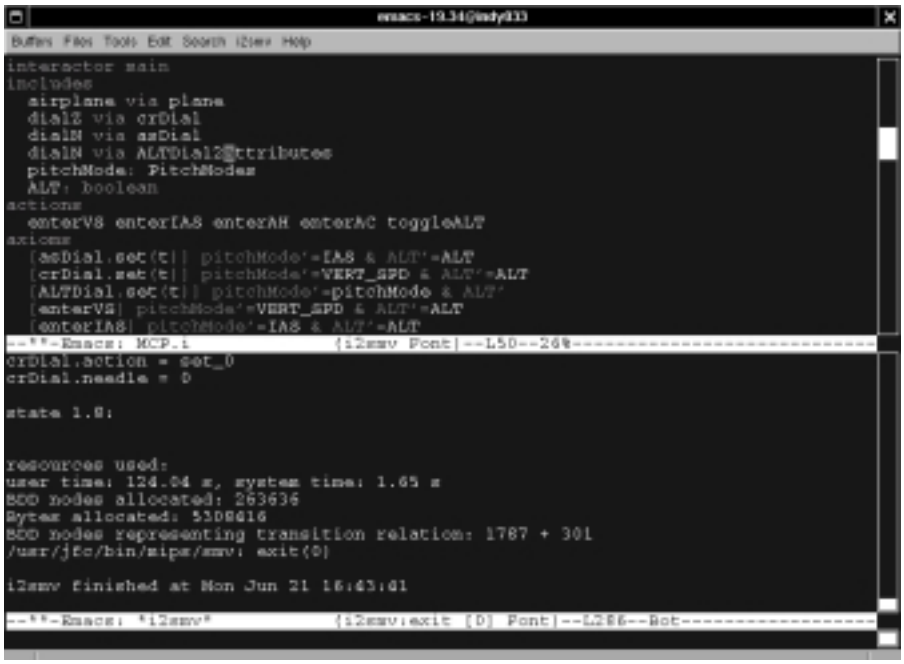

Figure 7. The Interactors to SMV compiler

will automatically generate and model check the SMV equivalent of the interactor model.

An Emacs[3] mode has been written to provide an integrated environment for the development, translation and verification of interactor specifications. Figure 7 shows the tool in use. The top pane holds an interactor model, the bottom pane shows the result of model checking the model with SMV. Once a model is written, the menu option i2smv on the menu bar, provides two alternatives:

- Compile & Verify — This results in what is shown in Figure 7. The model is compiled and SMV automatically used on the resulting code. A pane is created to show the result of the verification. This is the option used during normal operation. It allows `i2smv` and `smv` to be work together in a completely transparent manner: only the interactor model and the result of the verification need to be seen.

- Compile — This option will not usually be used during normal operation of the tool. It is provided to allow access to the generated SMV code. The model is compiled to SMV code and the generated

[3] `http://emacs.org` (last accessed on the 21st of July, 1999).

file is then opened in Emacs. SMV has its own mode, so Emacs will change from interactor mode to SMV mode.

4. Modelling 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 notes that similar problems are frequently reported to the Aviation Safety Reporting System[4].

4.1. 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 et al., 1994). It is important to note that this type of problem relates to how the system and user interact, and not to the behaviour of the system on its own. In fact, 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). Because of this need to consider the user, usability related issues are usually addressed by performing simulations of real-life interaction situations, using real users.

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

[4] `http://olias.arc.nasa.gov/ASRS/ASRS.html` (last accessed on the 2nd of November, 1999).

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 build a generic model of the artifact under consideration, and then analyse those aspects of the behaviour that are highlighted by the case study. Hence, if we are able to detect the problem 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 approach. We have read the Palmer paper and therefore have been tainted by it. This may influence the model we develop or the questions we might ask. We argue that the model and the questions are not affected by the details of the Palmer scenario.

Note that since we are interested in the interaction between the user and the artifact, our model will focus mainly on what is relevant in that dialogue. In particular we will not model in great detail the inner workings of the artifact, only the manifestations that are present at the interface. This process of abstraction is common in model checking. Of course a question might be raised as to whether the interface presentation accurately reflects the internal state of the system and whether the model has been so biased towards the question that other important characteristics of the system will go undetected. This appears not to be the case, the model focusses on the key actions and the parameters that are presented by the interface.

In summary then, what we have done is quite different from building a model around the Palmer scenario. Here we are using generic use case type questions as a starting point for the 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.

4.2. Selecting what to analyse

As seen in Figure 1 above, the first step is deciding exactly what features of the systems we wish to analyse. Identifying relevant requirements and properties to ensure correctness can be a nontrivial task. This is especially true of open systems, such as is the case with interactive systems, where the correctness of system behaviour can be

verified only in the context of assumptions made about the environment (cf. the assumption-commitment paradigm, de Roever, 1998).

Since these requirements and properties are related to the user, the process of obtaining them becomes the focus for interdisciplinary discussion. In practice designers can resort to verification whenever a decision has to be made, about some particular aspect of the interface design, which might have a critical impact on the system safety, or when the consequences of such decision are not fully clear.

In the present case we will be looking at how the automation and user interact during altitude acquisition. A reasonable expectation for the pilot to have of the system is that:

> Whenever the pilot sets the automation to climb up to a given altitude, the aircraft will climb until such altitude is acquired and then maintain it.

We will proceed as if such a request for analysis had been made by the design team, and follow the process outlined in Figure 1.

The property above relates to the vertical guidance subsystem of the aircraft mode logic. On the MD-88 the pilot interacts through a panel called the Mode Control Panel (MCP). The functionality of the MCP will be described in section 4.4, together with the model that was built. Information regarding the current flying modes is displayed on the Flight Mode Annunciator (FMA). We will include the relevant components of the FMA as attributes (pitchMode and ALT) of the MCP model (see Figure 10).

4.3. Modelling the context as a finite system

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. In this case we need to model the aircraft state in order to relate it to the automation state.

The aircraft is a continuous system, but our specifications are discrete. This means we will need to substitute state variables that range over continuous state spaces, by corresponding (abstracted) state variables that range over discrete domains (cf. Heitmeyer et al., 1998). In the present context we are specially interested in the altitude. Hence, state changes will correspond to changes in the altitude by some amount (say 1 in some unit of measure). In order to model the state transitions action fly is introduced. The model for the aircraft is shown in Figure 8. Besides asserting the change of altitude in each transition, the axiom for fly relates climb rate to the altitude change.

interactor aircraft
attributes
altitude: Altitude
airSpeed: Velocity
climbRate: ClimbRate
actions
fly
axioms
(1) [fly] (altitude′ >=altitude - 1 ∧ altitude′ <=altitude + 1) ∧
(altitude′ <altitude → climbRate′ <0) ∧
(altitude′=altitude → climbRate′=0) ∧
(altitude′ >altitude → climbRate′ >0)

Figure 8. The aircraft

We must be careful that the abstraction process above does not affect the behaviour of the system regarding the properties we will be checking. In this case we must ensure that the steps in altitude are small enough, when compared with the remaining behaviour of the system to provide a realistic basis for analysis in context. This will be further addressed in Section 5.1, when the domain of the types is discussed.

4.4. Modelling the MCP

When modelling the MCP (see Figure 9[5]), we can take into account the specific analysis we will be wanting to perform in order to build a simpler model. In this case we want to validate the pilot's assumption that setting both the altitude and an adequate pitch mode will cause the aircraft to climb to that altitude. This amounts to verifying the safety

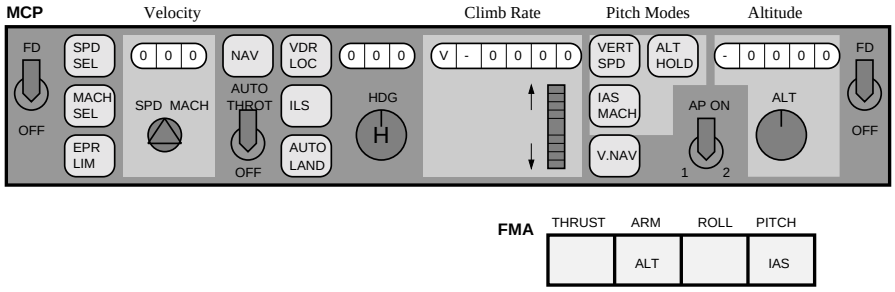

Figure 9. The MCP

[5] Figure adapted from (Honeywell Inc., 1988).

of operation of the pitch modes. The components that were deemed relevant are shown in Figure 9 in a lighter background. The model will include setting velocity, climb rate, and altitude, and selecting the appropriate pitch mode (see below). We are then making the assumption that the other components of the MCP (for example, lateral navigation, and thrust) will not affect the safety of operation of the pitch modes. This assumption could then be discharged by a separate proof process. In this way, we are able to not only assess specific design decisions regarding particular aspects of the design, but also verify the safety of the overall design in a compositional manner.

The model was built using interactors. From Figure 9 we see we must consider three main dials:

- airspeed (velocity);
- vertical speed (climb rate);
- the altitude window (i.e. altitude to which the aircraft 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 the parameterised interactor introduced in Section 2.2 (see Figure 3) to represent the different dials. At this level of abstraction, dials are represented by an attribute (needle) and an action (set). The action corresponds to setting a value (Axiom 1). The attribute represents the value that has been set.

How the MCP influences the automation will depend on its operating pitch mode. The pitch mode defines how the aircraft behaves during aircraft ascent/descent. There are four pitch modes:

- `VERT_SPD` (vertical speed pitch mode): instructs the aircraft to maintain the climb rate indicated in the MCP (the airspeed will be adjusted automatically);
- `IAS` (indicated airspeed pitch mode): instructs the aircraft to maintain the airspeed indicated in the MCP (the climb rate will be adjusted automatically);
- `ALT_HLD` (altitude hold pitch mode): instructs the aircraft to maintain the current altitude;
- `ALT_CAP` (altitude capture pitch mode): internal mode used by the aircraft to perform a smooth transition from `VERT_SPD` or `IAS` to `ALT_HLD` (see `ALT` below).

interactor MCP
includes
aircraft **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) (ALT ∧ |ALTDial.needle - plane.altitude|≤2) → obl(enterAC)
(11) (pitchMode=ALT_CAP ∧ plane.altitude=ALTDial.needle) → obl(enterAH)
Invariants
(12) pitchMode=VERT_SPD → plane.climbRate=crDial.needle
(13) pitchMode=IAS → plane.airSpeed=asDial.needle
(14) pitchMode=ALT_HLD → plane.climbRate=0
(15) pitchMode=ALT_CAP → plane.climbRate=1

Figure 10. The MCP model

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 aircraft to stop climbing when the altitude indicated in the MCP is reached.

The MCP operation is described by the interactor in Figure 10. 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). These axioms specify mode changes that are implicitly carried out by the automation as a consequence of user activity. 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. Note that action enterAC (setting the pitch mode to ALT_CAP) is an internal system event. Axioms 9 and 10 specify the mode logic that regulates when the event happens: 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. Similarly, Axiom 11 specifies that the system must set the pitch mode automatically to ALT_HLD once the desired altitude has been reached. And finally, Axioms 12 to 15 describe the effect of the pitch modes on the state of the aircraft.

Having built a model we now want to analyse it. Since the property under analysis relates to the temporal behaviour of the model, model checking is the natural choice of the technique to be used (Campos and Harrison, 1998). 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 how the properties we are interested in can be expressed in CTL. The next section deals with these issues.

5. Checking the Design

Having developed a model for the MCP we will now analyse it using SMV, with the help of the tool described in Section 3. This section deals with the steps required to perform the analysis.

5.1. Converting the Model

In order to check the specification in SMV 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 aircraft 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 5. 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).

This abstraction process is similar to the Application State abstraction in (Dwyer et al., 1997). Hence, because the properties we will check are all universally quantified, we will not have the problem of false positives.

We add the following types to the specification:

Altitude = {0, 1, 2, 3, 4, 5}
Velocity = {0, 1, 2, 3, 4, 5}
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 and minimum altitudes (see Appendix A).

Since the use case we are considering deals with altitude acquisition we have chosen not to include negative (below the sea-level) altitudes in the model: the minimum value for altitude is zero. Considering negative altitudes would be trivial: only the definition of Altitude, and the minimum altitude in interactor plane would need to be adapted to the new minimum value. If this was done, the kill the capture problem would still be detected.

Besides the changes above, the name of interactor MCP must be changed to main. Additionally we add a fairness condition: in this case that the system should not be continuously idle. The checkable version of the specification is presented in Appendix A. This version is automatically convertible to SMV using the compiler.

We are now able to translate our model to SMV. We now have to express, as CTL formulae, the properties which we want to analyse using the model checker. These formulae can then be included in the model using test clauses.

5.2. Formulating and checking properties

CTL formulae can be automatically checked by SMV. We need, then, to express relevant user properties as CTL formulae.

As seen in Section 4.2, the design of the interface has been based on the plausible assumption that pilots expect that, if the altitude capture (`ALT`) is armed, the aircraft will stop at the desired altitude (selected in `ALTDial`). We can express this as the CTL formula:

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

which reads: it always (AG) happens that, if the plane is below the altitude set on the MCP and the altitude capture is on, then it is inevitable (AF) that the altitude be reached and the pitch mode be changed to altitude hold.

Note that no knowledge of the SMV input language is needed to write the property, only knowledge of CTL. The properties are written at the interactor model level, and the translation and verification steps can be seen as a single "black-box" step. In order to refer to actions in CTL the occurrence operator can be used. Hence, it is possible to check whether an action will be possible or not, although CTL has no direct notion of action.

When we model check a specification, the checker answers whether or not the test succeeds. If the answer is false, and a counter example can be found, it gives the first counter example it finds. When we check the model against the formula above, we get the following result[6]:

```
-- specification AG (plane.altitude < ALTDial... is false
-- as demonstrated by the following execution sequence
state 1.1:
...

state 1.2:
...

state 1.3:
...

-- loop starts here --
state 1.4:
plane.climbRate = 1
plane.altitude = 1
ALTDial.action = set_4
crDial.action = set_1
crDial.needle = 1

state 1.5:
plane.climbRate = -1
plane.altitude = 0
crDial.action = set_-1
crDial.needle = -1
```

[6] 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.

```
state 1.6:
plane.climbRate = 1
plane.altitude = 1
crDial.action = set_1
crDial.needle = 1

resources used:
user time: 198.7 s, system time: 2.91 s
BDD nodes allocated: 625225
Bytes allocated: 11075584
BDD nodes representing transition relation: 1787 + 301
```

What the model checker points out is that the pilot might continuously change the climb rate so as to keep the aircraft flying below the altitude set on the MCP (look at `crDial.action`). Although this might seem an obvious (if artificial) situation, it does raise the issue of how the automation reacts to changes in the climb rate when an altitude capture is armed, in particular changes that cause the aircraft to deviate from the target altitude.

Since the model does not describe that aspect in detail, we would have to refer back to the designers in order to raise the point. If needs be, the model could then be refined to include this particular aspect of the automation behaviour in greater detail. 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.

With this information, we can continue to explore the model. As a result of the previous scenario, our expectations of pilot's beliefs must be refined to include the fact that changing the climb rate can prevent the aircraft from reaching the desired altitude. The test formula now becomes:

```
AG((plane.altitude < ALTDial.needle & ALT) ->
    AF((pitchMode=ALT_HLD & plane.altitude=ALTDial.needle)
       | plane.climbRate = -1))
```

It reads: in the conditions stated, the plane will stop at the desired altitude, unless action is taken to start descending. Again, this is a reasonable expectation to have. Note how the tool has prompted us to include the circumstance of the plane starting to descend.

Despite being a reasonable expectation, 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 VERT_SPD (for instance by setting the corresponding dial) when in ALT_CAP, effectively

kills the altitude capture (i.e. the request to stop climbing at the target altitude). In effect, when the pitch mode changes to ALT_CAP, the altitude capture is automatically switched off. However, the aircraft is still climbing. This means that any subsequent action from the pilot that causes the pitch mode to change, will cause the aircraft to keep climbing past the target altitude.

If we refer back to (Palmer, 1995) we see that this is a similar problem to that detected during simulation. Basically, once the aircraft changes into ALT_CAP mode, there are user actions that might lead to a "kill the capture" mode error and a consequent altitude bust. We claim that we could have achieved this result without knowledge of the simulation results. In particular, we gave SMV no specific chain of events to analyse, rather the analysis revolved around a simple generic use case concerned with altitude capture. It was the tool that pointed out to us a particular sequence of events that could lead to a hazardous situation. We could have applied this automated verification 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.

As stated previously, finding a problem is just a trigger for further analysis and discussion. Dialogue must be undertaken with the designers and human-factors experts in order to clarify the full consequences of the problem, and how it can be solved. How aware will the pilot be of the mode change to ALT_CAP performed by the automation? Is this issue adequately covered in the manuals, and during training? Should the system be redesigned and how? What engineering constraints come into play regarding the design? Being able to raise these issues against a formal proof background in early design stages, and not only when the design reaches the level of prototyping and user testing, will undoubtedly allow for a better/safer design from the start.

6. Related Work

In recent years, the use of automated reasoning tools for software verification has attracted considerable interest. In this paper we have considered model checking interactive systems' specifications for the verification of interactive systems designs. Our work can be compared to current literature in a number of ways.

6.1. The Case study

The specific case study that we have used is also analysed in (Leveson and Palmer, 1997) and (Rushby, 1999). Leveson and 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 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 and will potentially remove some elements of analyser bias. 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).

Unlike us, however, Rushby (1999) builds his specification around the specific sequence of events that is identified in (Palmer, 1995) as the cause for the altitude bust. We believe 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 and Harrison, 1999) we give an example involving the analysis of awareness in a computer mediated communications system.

6.2. SMV and requirements verification

Although the use of model checking as a verification tool has met with more acceptance in the areas of hardware and communication protocols design, its use in more general settings (as is the case in this paper) is also being addressed. In (Atlee and Gannon, 1993) the use of the MCB model checker for the verification of safety properties of software requirements is reported. More recently (see Sreemani and Atlee, 1996) the use of SMV has also been addressed. In both cases the model

checker is used to analyse properties of model transition tables of SCR (Software Cost Reduction) specifications.

The work above relates to properties of single mode transition tables with boolean variables only. This has been expanded upon by Heitmeyer's group to consider properties of complete SCR specifications (Heitmeyer et al., 1998; Bharadwaj and Heitmeyer, 1999). In order to reduce the complexity of the state machines being analysed, (Bharadwaj and Heitmeyer, 1999) proposes two abstraction methods that allow the elimination of unnecessary variables. We note that these abstractions are applied to the whole specification. This differs from our approach where abstractions and information about the properties to be checked are used to build a partial models of the system. We believe our approach to be more appropriate for early stages of design, since it does not impose the need for a full model of the system. In any case, abstractions such as those proposed can be used in the context of our approach.

Also in the context of the verification of requirements specifications, SMV is also used by (Chan et al., 1998) to analyse RSML (Requirements State Machine Language) specifications. In this case, however, the translation to SMV is not necessarily semantics-preserving, which might lead to SMV models whose semantics differs from the original RSML specifications.

All of the work above concentrates on verification of the requirements specification. This differs from our work in that we are mainly interested in verifying the interaction between the user and the system, not the system by itself. While it is obvious that the system must behave correctly, this is clearly not enough. It is also necessary that system and user interact correctly. Hence, our work can be seen as complementary to the work in requirements verification.

While SCR and RSML have been used with success for the specification of safety critical systems, we think that the use of MAL as a specification language provides greater expressive power. The deontic operators for permission and obligation allow the specification of more complex behaviour patterns, yet maintain a good degree of readability and ease of use. In fact, experience has shown that behaviour of MAL based interactor models will be mostly based on the notion of event, with permission axioms stating the conditions for the events to be valid, and modal axioms stating the effect of the event on the state. This is basically the traditional style of specifying a system using pre- and post-conditions for the possible events.

Additionally, we are able to translate the MAL based models into SMV in a fully automated manner. As far as we can tell, in all the approaches above some degree of manual intervention continues to be

needed. The possibility of automated translation is crucial, since human intervention has the potential to introduce translation errors, which can be hard to detect.

Chan et al. (1998) discus the need for an iterative approach to development, where model checking is used as a design tool. This is similar to our view of the role of verification in interactive systems design. In (Campos and Harrison, 1998) we have argued for the use of both model checking and theorem proving during design to help guide the design process. Additionally, the need for ways to identify meaningful properties to check is also mentioned in (Chan et al., 1998). We believe that considering the user during verification is one such approach to generating properties..

6.3. Interactive Systems verification

In recent years a number of authors have started studying the application of automated reasoning tools to the development of interactive systems. Paternò has proposed the use of the Lite tool set (Mañas et al., 1992) in the analysis of interactive systems specifications (see, for example, Paternò, 1995, Paternò and Mezzanotte, 1995). He uses a flavour of Interactors written in LOTOS (Bolognesi and Brinksma, 1987) to make a hierarchical specification of the user interface, based on the task analysis output. The translation process from a LOTOS specification to a finite state machine implies that information will be lost. The loss of conditional guards, in particular, will cause the checkable version of the specification to admit more traces of behaviour than the original LOTOS version. Approaches have been proposed to solve this problem, but they imply either restrictions on how systems can be modelled (Paternò, 1995) or a manual translation of the specification (Palanque et al., 1996).

The properties to verify are written in ACTL (Nicola et al., 1993), this means that the approach is heavily based on the notion of event. We believe that in this case the state based approach of CTL is a better choice. While it is simple to encode actions as state attributes, encoding state information as actions is a complicated task (and one that is not easily amenable to automation). Additionally, since the specifications are basically architectural descriptions of the user interface, as resulting from the task analysis, it is not possible to reason about the relation between interface and system behaviour. Our iterative view of the verification process allows simpler models and properties which are more directly connected to user considerations, encoding not only information about the system, but also about the user. In particular, a task based approach could be used to generate properties for

verification but there is danger in such an approach that we can be over prescriptive about what the operator does. Humans do not follow procedures instruction by instruction in general.

Bumbulis et al. (1996) reports on the use of HOL (a theorem prover) for interactive systems verification. The approach deals with properties of the interface at the device level. This type of approach is rather restrictive in the properties that can be verified. In our proposal, the analysis starts much sooner and can be performed as the development progresses. In (Doherty et al., 2000) we show how theorem proving can be used to perform a more powerful analysis of the actual interface being built. This is accomplished by analysing the relationship between user interface devices, underlying system state, and the perception the users might have of the system.

7. Discussion and Conclusions

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 described a tool to automate this translation. We have also shown how we can use such interactor specifications in conjunction with the tool to model and analyse a realistic case of mode confusion. Having decided to analyse the MCP panel, we built a model of the artifact. We then used CTL formulae to express user expectations about the operation of the artifact. During verification of such formulae, issues were raised about the behaviour of the system, and scenarios were found where the system did not behave as expected. The analysis of these results acted as a focus for further interdisciplinary discussion regarding the meaning of the results and how they should influence the design. A revised version of the proposed verification process taking into account these discussions is presented in Figure 11.

One problem in relation to model checking is to find a model that is both sufficiently expressive and consisting of a small enough number of states. One aim of the paper has been to show how reasoning about interesting features of a complex system can be done without resorting to a complete specification of the system.

The use of interactors and SMV gives us a degree of freedom and expressive power, but that does not come without a cost. In particular,

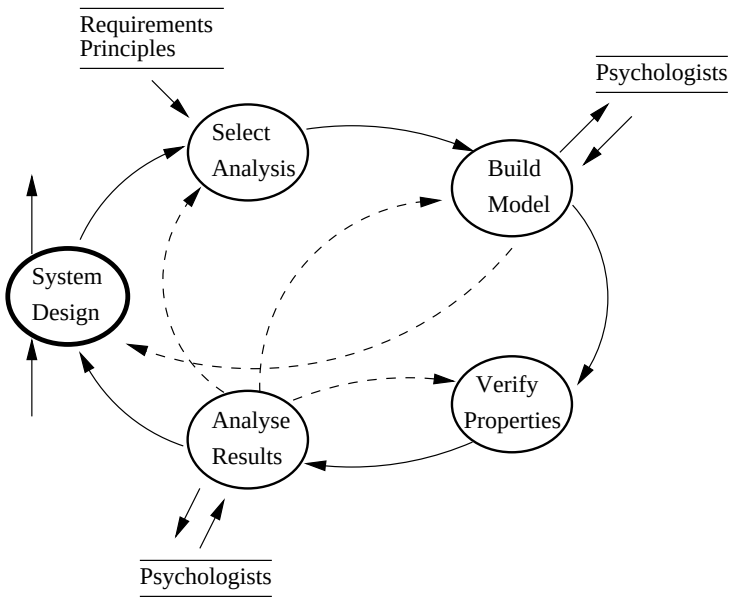

Figure 11. The verification process revisited

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 sets the climb rate of the aircraft to zero, 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, or by reworking the specification with the particular context of analysis in mind.

Another potential problem with model checking is the size of the counter examples generated by the tool. So far experience has shown the counter examples to be small (tens of states). We believe this is due to the use of partial models of the systems.

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. Rather, it is a characteristic of verification in general. 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 wish, 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. Formal verification

does not give us an absolute guarantee of correctness (Clarke and Wing, 1996; Henzinger, 1996), it is up to designers and human-factors experts to identify what are the critical issues in the design of an interactive system. What formal verification techniques offer is a way to rigorously reason about such issues, and to prove formally whether the criteria are met or not early in the design cycle.

Another question that could be raised is how to guarantee that different models of the same system are consistent between each other. In (Campos and Harrison, 1999) we show how this can be achieved by consistent overlapping of the different models, and even how discrepancies between different models can be used to detect problems in the design.

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., H.-M. Wang, and A. F. Monk: 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.

Atlee, J. M. and J. Gannon: 1993, 'State-Based Model Checking of Event-Driven Systems Requirements'. *IEEE Transactions on Software Engineering* **19**(1).

Bharadwaj, R. and C. L. Heitmeyer: 1999, 'Model Checking Complete Requirements Specifications Using Abstractions'. *Automated Software Engineering* **6**(1), 37–68.

Bodart, F. and J. Vanderdonckt (eds.): 1996, 'Design, Specification and Verification of Interactive Systems '96', Springer Computer Science. Springer-Verlag/Wien.

Bolognesi, T. and E. Brinksma: 1987, 'Introduction to the ISO Specification Language LOTOS'. *Computer Networks and ISDN Systems* **14**(1), 25–59.

Bumbulis, P., P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena: 1996, 'Validating Properties of Component-based Graphical User Interfaces'. in (Bodart and Vanderdonckt, 1996), pp. 347–365.

Burch, J. R., E. M. Clarke, and K. L. McMillan: 1990, 'Symbolic model checking: 10^{20} States and Beyond'. In: *Proceedings of the Fifth Annual IEEE Symposium on Logic In Computer Science*. IEEE Computer Society Press, pp. 428–439.

Campos, J. C.: 1999, 'Automated Deduction and Usability Reasoning'. DPhil thesis, Department of Computer Science, University of York.

Campos, J. C. and M. D. Harrison: 1997, 'Formally Verifying Interactive Systems: A Review'. in (Harrison and Torres, 1997), pp. 109–124.

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

Campos, J. C. and M. D. Harrison: 1999, 'Using automated reasoning in the design of an audio-visual communication system'. In: D. J. Duke and A. Puerta (eds.): *Design, Specification and Verification of Interactive Systems '99*, Springer Computer Science. Springer-Verlag/Wien, pp. 167–188.

Chan, W., R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese: 1998, 'Model Checking Large Software Specifications'. *IEEE Transactions on Software Engineering* **24**(7), 498–520.

Clarke, E. and J. M. Wing: 1996, 'Tools and partial analysis'. *ACM Computing Surveys* **28**(4es), 116–es.

Clarke, E. M., E. A. Emerson, and A. P. Sistla: 1986, 'Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications'. *ACM Transactions on Programming Languages and Systems* **8**(2), 244–263.

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

de Roever, W.-P.: 1998, 'The Need for Compositional Proof Systems: A Survey'. in (de Roever et al., 1998), pp. 1–22.

de Roever, W.-P., H. Langmaack, and A. Pnueli (eds.): 1998, 'Compositionality: The Significant Difference', Vol. 1536 of *Lecture Notes in Computer Science*. Springer.

Doherty, G., J. C. Campos, and M. D. Harrison: 2000, 'Representational Reasoning and Verification'. *Formal Aspects of Computing* (in press).

Duke, D., P. Barnard, J. May, and D. Duce: 1995, 'Systematic Development of the Human Interface'. In: *Asia Pacific Software Engineering Conference*. IEEE Computer Society Press, pp. 313–321.

Duke, D. J. and M. D. Harrison: 1993, 'Abstract Interaction Objects'. *Computer Graphics Forum* **12**(3), 25–36.

Dwyer, M. B., V. Carr, and L. Hines: 1997, 'Model Checking Graphical User Interfaces Using Abstractions'. In: M. Jazayeri and H. Schauer (eds.): *Software Engineering — ESEC/FSE '97*, No. 1301 in Lecture Notes in Computer Science. Springer, pp. 244–261.

Faconti, G. and F. Paternò: 1990, 'An Approach to the Formal Specification of the Components of an Interaction'. In: C. Vandoni and D. Duce (eds.): *Eurographics '90*. North-Holland, pp. 481–494.

Fiadeiro, J. and T. Maibaum: 1991, 'Temporal Reasoning over Deontic Specifications'. *Journal of Logic and Computation* **1**(3), 357–395.

Fields, B., N. Merriam, and A. Dearden: 1997, 'DMVIS: Design, Modelling and Validation of Interactive Systems'. in (Harrison and Torres, 1997), pp. 29–44.

Harrison, M., R. Fields, and P. C. Wright: 1996, 'The User Context and Formal Specification in Interactive System Design (invited paper)'. In: C. R. Roast and J. I. Siddiqi (eds.): *Formal Aspects of the Human Computer Interface*, *electronic* Workshops in Computing. Springer-Verlag London.

Harrison, M. D. and J. C. Torres (eds.): 1997, 'Design, Specification and Verification of Interactive Systems '97', Springer Computer Science. Eurographics, Springer-Verlag/Wien.

Heitmeyer, C., J. Kirby, and B. Labaw: 1998, 'Applying the SRC Requirements Method to a Weapons Control Panel: An Experience Report'. In: *Proceedings of*

the Second Workshop on Formal Methods in Software Practice (FMSP '98). pp. 92–102.

Henzinger, T. A.: 1996, 'Some myths about formal verification'. *ACM Computing Surveys* **28**(4es), 119–es.

Honeywell Inc.: 1988, 'SAS MD-80: Flight Management System Guide'. Honeywell Inc., Sperry Commercial Flight Systems Group, Air Transport Systems Division, P.O. Box 21111, Phoenix, Arizona 85036, USA. Pub. No. C28-3642-22-01.

Leveson, N. G. and E. Palmer: 1997, 'Designing Automation to Reduce Operator Errors'. In: *Proceedings of the IEEE Systems, Man, and Cybernetics Conference*.

Mañas, J. A. et al.: 1992, 'Lite User Manual'. LOTOSPHERE consortium. Ref. Lo/WP2/N0034/V08.

McMillan, K. L.: 1993, *Symbolic Model Checking*. Kluwer Academic Publishers.

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

Newman, W. M. and M. G. Lamming: 1995, *Interactive System Design*. Addison-Wesley.

Nicola, R. D., A. Fantechi, S. Gnesi, and G. Ristori: 1993, 'An action-based framework for verifying logical and behavioural properties of concurrent systems'. *Computer Networks and ISDN Systems* **25**(7), 761–778.

Palanque, P., F. Paternò, R. Bastide, and M. Mezzanote: 1996, 'Towards an integrated proposal for Interactive Systems design based on TLIM and ICO'. in (Bodart and Vanderdonckt, 1996), pp. 162–187.

Palmer, E.: 1995, '"Oops, it didn't arm." - A Case Study of Two Automation Surprises'. In: R. S. Jensen and L. A. Rakovan (eds.): *Proceedings of the Eighth International Symposium on Aviation Psychology*. Columbus, Ohio, pp. 227–232.

Paternò, F. and M. Mezzanotte: 1995, 'Formal Analysis of User and System Interactions in the CERD Case Study'. Technical Report SM/WP48, Amodeus Project.

Paternò, F. D.: 1995, 'A Method for Formal Specification and Verification of Interactive Systems'. Ph.D. thesis, Department of Computer Science, University of York.

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., J. Fiadeiro, and T. Maibaum: 1991, 'Sharing Actions and Attributes in Modal Action Logic'. In: T. Ito and A. R. Meyer (eds.): *Theoretical Aspects of Computer Software*, Vol. 526 of *Lecture Notes in Computer Science*. Springer-Verlag, pp. 569–593.

Sreemani, T. and J. M. Atlee: 1996, 'Feasibility of Model Checking Software Requirements: A Case Study'. In: *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS '96)*. pp. 77–88.

Wall, L., T. Christiansen, and R. L. Schwartz: 1996, *Programming Perl*. O'Reilly & Associates, Inc., 2nd edition.

Woods, D. D., L. J. Johannesen, R. I. Cook, and N. B. Sarter: 1994, 'Behind Human Error: Cognitive Systems, Computers, and Hindsight'. State-of-the-Art Report SOAR 94-01, CSERIAC.

Appendix

A. Checkable Specification

This is the translatable version of the interactor specification for the MCP. Besides the usual substitution of `>=` for $\geq$, etc., `action` is used instead of $>_{\texttt{action}}$. The compiler is line oriented, hence, each expression must be fully contained in a single line. However, line breaks can be escaped with the backslash character, thus allowing, for example, multi-line axioms.

```
# MCP example
types
  PitchModes = {VERT_SPD, IAS, ALT_HLD, ALT_CAP}
  Altitude   = {0, 1, 2, 3, 4, 5}
  Velocity   = {0, 1, 2, 3, 4, 5}
  ClimbRate  = {-1, 0, 1}

interactor aircraft
attributes
  altitude: Altitude
  airSpeed: Velocity
  climbRate: ClimbRate
actions
  fly
axioms
  (altitude>0 & altitude<5) -> [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=5 -> [fly] \
     ((altitude'>=altitude - 1 & altitude'<=altitude) & \
      (altitude'<altitude -> climbRate'<0) & \
      (altitude'=altitude -> climbRate'>=0))
fairness
  !action=nil
```

```
interactor dial(T)
attributes
  needle: T
actions
  set(T)
axioms
  [set(v)] needle'=v

interactor main
includes
  aircraft via plane
  dial(ClimbRate) via crDial
  dial(Velocity) via asDial
  dial(Altitude) via ALTDial
  pitchMode: PitchModes
  ALT: boolean
actions
  enterVS enterIAS enterAH enterAC toggleALT
axioms
  [asDial.set(t)] pitchMode'=IAS & ALT'=ALT
  [crDial.set(t)] pitchMode'=VERT_SPD & 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
  per(enterAC) -> (ALT & \
                   (ALTDial.needle - plane.altitude)<=2 &\
                   (ALTDial.needle - plane.altitude)>=-2)
  [enterAC] pitchMode'=ALT_CAP & !ALT'
  (ALT & pitchMode!=ALT_CAP & \
   (ALTDial.needle - plane.altitude)<=2 & \
   (ALTDial.needle - plane.altitude)>=-2) -> obl(enterAC)
  pitchMode=VERT_SPD -> plane.climbRate=crDial.needle
  pitchMode=IAS -> plane.airSpeed=asDial.needle
  pitchMode=ALT_HLD -> plane.climbRate=0
  pitchMode=ALT_CAP -> plane.climbRate=1
  ALTDial.needle < 5
  (pitchMode=ALT_CAP & plane.altitude=ALTDial.needle) -> \
                                              obl(enterAH)
  [] plane.altitude = 0
fairness
  !action=nil
```

```
test
  AG((plane.altitude < ALTDial.needle & ALT) ->
    AF((pitchMode=ALT_HLD & plane.altitude=ALTDial.needle)
       | plane.climbRate = -1))
```

Address for Offprints:
José C. Campos
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga, Portugal
e-mail: jfc@di.uminho.pt
fax: +351 253 60 4471