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 draw attention to "automation surprises" that arise in safety critical systems. An automation surprise can occur when a system behaves differently from the expectations of the operator. Interface mode changes are one class of such surprises that have significant impact on the safety of a dynamic interactive system. They 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. This 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 concerned with the use of automated reasoning techniques (more specifically model checking) during interactive systems design. Model checking is a verification technique that 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.

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 operator. A particular class of such surprises, interface mode changes, has significant impact on the safety of a dynamic interactive system and may take place *implicitly* as a result of other system action. The formal specification of an interactive system offers an opportunity to

[*] Published in *Automated Software Engineering*, 8(3/4):275-310, August 2001.

analyse the consequences of its design and thereby reduce the risk of this type of interface problem.

Relevant analyses (Leveson and Palmer, 1997; Rushby, 1999) of mode complexity in aviation based systems have been conducted retrospectively using experience based on flight simulations. Scenarios of use based on past experience provide a foundation for analysis in these papers. In contrast we consider the role that interactor based models (Faconti and Paternò, 1990; Duke and Harrison, 1993) have in analysing mode consequences early in the design process. An *interactor* is an object (consisting of state and operations) with the additional property that state perceivable to the user, and actions that are accessible to the user, are identified explicitly. The main advantage of interactors is that they allow the specification of both system state and behaviour and user interface presentation and behaviour in the same framework. This 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. A property of concern may not represent a failure for the interactive system rather it may highlight scenarios where special care should be taken to understand how the user will interact with the system at this point.

Concepts of usability derived from psychological or sociological understandings are difficult if not impossible to rationalise in a form that can be used as part of a verification process. Concepts such as *task* (a unit of human activity carried out to achieve a specific goal) and user interface mode (how the system responds to input and how the state is represented and perceived as 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. The aim is that each model should focus on specific 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. An automatic change in the flying mode led to pilot action of cancelling the scheduled automatic function of climbing to a specific altitude. Clearly the fact that this situation may arise has significant impact on the safety of the aircraft. For example, air traffic problems may arise from the loss of separation with other aircraft. We will show how checking models that describe the interface between the pilot and the automation may help early detection of problems such as these. The kind of analysis we are dealing with is not primarily concerned with the behaviour of the system by itself, but with the interaction between the system and its users. Our methodology requires input from human-factors specialists to be incorporated in the verification process so that we can explore how system and user behave together.

Two basic types of automated verification technique can be identified: model checking and theorem proving. Each technique has its own strong points. Model checking is usually best at verifying reachability properties of systems, while theorem provers are best at verifying properties related to the system's state. The focus of this paper is on the former. 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 into a coherent verification process.

In Section 2 we expand on the role that verification can play during interactive system design and introduce the interactor notation. In Section 3 we describe a tool that enables us to check interactors in SMV. In Section 4 we use the interactor notation 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). We will show how abstractions can be used to keep the model clear for analysis by both systems and human factors specialists even in the presence of continuous and non-continuous subsystems that have to be modelled 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

We argue that there are nuances in the verification of interactive systems that differentiate the process from the more general problem of software verification. *Interactors* help to give proper emphasis to human interface components of the system and can be used in the verification process. We introduce the particular interactor and verification technique that we shall be using.

2.1. The Role of Verification in Interactive Systems Design

Formal verification of interactive systems can be seen to fall into two categories:

- known problems of existing systems: explaining why problems arise;
- discovery of consequences of a particular interactive system specification: establishing whether a proposed system exhibits desired properties.

In the case of known problems, hindsight drives the development of a model in order to analyse the particular problem. This type of analysis can be useful in explaining what went wrong but of course it cannot predict design problems.

If we can discover the consequences of a particular design proposal then errors can be detected and prevented before the system is used in practice. Approaches related to this issue tend to be based around the development of a specification of the entire system (cf. Paternò, 1995, Heitmeyer et al., 1998). This specification may be reverse engineered from the actual system implementation (cf. Bumbulis et al., 1996). (Campos and Harrison, 1997) provides a review of current approaches to the automated verification of interactive systems. Entire specifications can be hard and costly to change if problems are found and design decisions remade. Additionally, it is difficult to see how specifications which represent whole systems can be analysed effectively (cf. Campos and Harrison, 1998) for systems as complex as interactive systems.

As was mentioned earlier 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. In 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, "falsifi-

cation" would be a more appropriate name for the endeavor called "verification".

In (Campos and Harrison, 1998) it is argued that to explore the full potential of formal verification the verification step must be moved into the development process. Verification should be used as a guide in the process of design decision making (cf. the *"verify-while-develop"* paradigm, de Roever, 1998) rather than as a sanity check at the end of the process. Models can be built that highlight specific aspects of the artefact. Verification of these partial models can be used to highlight the specific concerns of different development stages (cf. Fields et al., 1997). The results of the analysis of such a model can then be fed-back to the design process (see Figure 1). This process can be applied repeatedly.

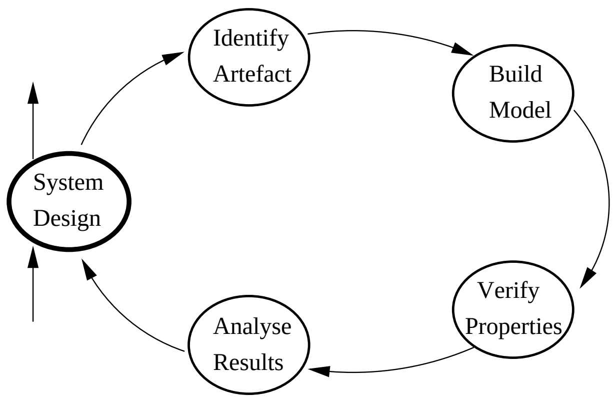


Figure 1. Verification process

The move towards a tighter integration between verification and design has a number of advantages:

- design decision making: allows for a more informed process.
- complexity control: building partial models focussed on the specific provides better control of the complexity of the models.
- reuse: it becomes possible to reuse models and/or proofs see (Campos, 1999, Chapter 5) for an example of reuse of a proof.
- technique fit: different properties require different styles of verification; by using a number of models we avoid being tied down to a particular verification technique (see Campos, 1999, Campos and Harrison, 1999, Doherty et al., 2000).

– property relevance: properties required to check the soundness of a complex specification may draw the focus away from system properties such as "freedom from unexpected mode changes"; by focusing our models on the specific aspects we want to analyse we are able to formulate properties that are relevant to that system rather than the specification of the system.

2.2. The Interactor Language

It has been argued elsewhere that traditional specification languages do not help designers focus on key issues in interactive systems. Interactors (Faconti and Paternò, 1990; Duke and Harrison, 1993) have been proposed as a structuring concept for such a task.

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

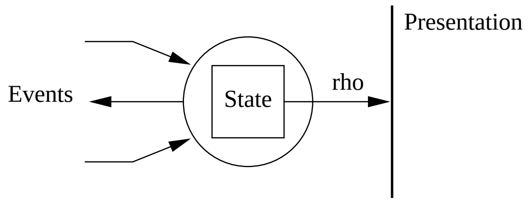


Figure 2. York Interactor

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 (Ryan et al., 1991; Fiadeiro and Maibaum, 1991) that has been adapted to interactor specification by Duke (Duke et al., 1995).

The definition of an interactor has three main components:

– state;

– behaviour;

– rendering.

The state of an interactor is modelled by a collection of typed attributes. We would specify a dial which indicates a value as follows:

interactor dial(T)
attributes
  needle: T

This interactor has only one attribute (needle) and the type of the attribute is T (the range of values in the dial). Type T is a parameter of the interactor which means that it is possible to have dials with different ranges.

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

action
  set(T)

The type T indicates that the action will have a parameter of that type. set is therefore a family of actions.

Interactor behaviour is described using a logic based on Structured MAL (Ryan et al., 1991). MAL (Modal Action Logic) is a (deontic) modal logic that incorporates a notion of *action*. Structured MAL adds mechanisms for structuring the specification to the basic MAL notation. A Structured MAL agent defines a labelled transition system where actions are used to label the transitions between states. MAL axioms will be used to define the behaviour of interactors. In addition to the usual propositional operators and actions the logic provides:

- a modal operator $[\_]\_$: if $[ac]expr$ then $expr$ is true in all states resulting from the occurrence of action ac — the modal operator is used to define the transition relation between states;
- a special reference event $[]$: if $[]expr$ then $expr$ is true in the initial state(s) — the reference event is used to define the initial state(s);
- a deontic operator per: if $\mathsf{per}(ac)$ then action ac is permitted to happen next;
- a deontic operator obl: if $\mathsf{obl}(ac)$ then action ac is obliged to happen some time in the future.

Deontic operators per and obl are a form of quantification over the actions in a given state:

- $\mathsf{per}(ac) \equiv \exists_{ac_1} \bullet ac = ac_1 \wedge [ac_1]true$
- $\mathsf{obl}(ac) \equiv \forall_{ac_1} \bullet [ac_1]true \rightarrow (ac = ac_1 \vee [ac_1]\mathsf{obl}(ac))$

where $[ac_1]true$ means that action ac_1 is possible in the current state. An obligation persists until the action occurs (cf. Fiadeiro and Maibaum, 1991).

A major difference between our logic and Structured MAL is in the treatment of the modal operator. In Structured MAL the modal operator is applied to whole propositions. There is no way to relate *old* and *new* values of attributes directly. Old and new values are often related in practice by the introduction of auxiliary variables. For example an action (*incr*) which increments the value of attribute needle above would be defined in Structured MAL as:

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

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 extend the definition of the modal operator of (Fiadeiro and Maibaum, 1991) by:

– applying the operator to attributes only;

– using priming to indicate which attributes are affected by it.

Hence the axiom above can be written as:

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

Parentheses will be omitted 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 rendering relation for the interactor presentation is defined by annotating actions and attributes to show that they are perceivable. The modality of the perceivable attribute/action is given using further attributes. For example $\boxed{\text{vis}}$ asserts that the parameter/action is visibly perceivable. In addition if attached to an action it can be invoked by the user. Additional annotations are introduced for further modalities. Taking all the above we get the dial in Figure 3.

Interactors are composed using inclusion (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

interactor dial(T)
attributes
  $\boxed{\text{vis}}$ needle: T
action
  $\boxed{\text{vis}}$ set(T)
axioms
  (1) [set(v)] needle'=v

Figure 3. Simple dial interactor

attributes of an interactor are always accessible to other interactors that include it. To initialise the needle of speedDial the following axiom can be added to interactor Panel:

  [] speedDial.needle=0

We assume the existence of type Range. Types will be represented as enumerations of the "key values" or as subranges of integer:

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

The modal operator allows us to prescribe the effect of actions in the state but says nothing 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:

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

Permissions are asserted therefore by default and obligations are off by default.

This makes it easier to add permissions and obligations incrementally when writing specifications. Permission axioms $\mathsf{per}(ac) \rightarrow guard_1$ and $\mathsf{per}(ac) \rightarrow guard_2$ together yield $\mathsf{per}(ac) \rightarrow (guard_1 \wedge guard_2)$ for example. This logic is particularly appropriate for 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 and more interested in strategies for proving properties of interactive behaviour than seeing a justification for a translation 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 theorem proving 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). In this context the possible execution paths are interpreted as alternative futures.

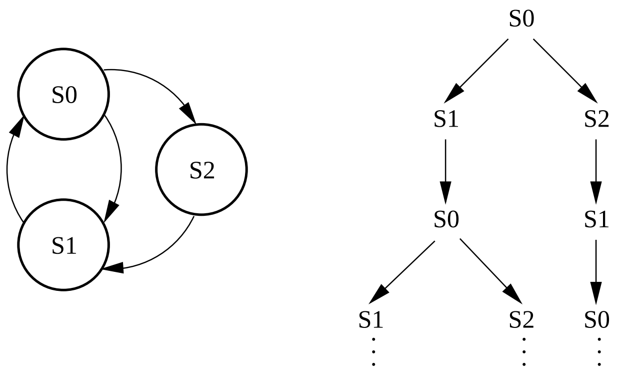


Figure 4. Execution paths (adapted from Abowd et al., 1995)

By using an algorithm to perform the state space analysis over a finite state system two major drawbacks of theorem provers can be avoided:

- the analysis is fully automated;
- the validity of a property is always decidable.

A main drawback of Model Checking is concerned 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 to analyse the entire state space. Decidable systems may become effectively undecidable in practice. The development of Symbolic Model Checking somewhat diminished this problem. By this means state spaces as large as 10^{20} states may be analysed (Burch et al., 1990).

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

Figure 5. An example SMV module

3.1.1. *The SMV input language*

An SMV specification is a collection of modules. Each module defines a finite state machine. A module consists of a number of state variables that are comparable with interactor attributes and a set of rules that specify how the module can progress from one state to the next (that are comparable with interactor axioms).

Figure 5 shows an example SMV module. Attributes are declared in clause `VAR`. Here there is only one attribute (`light`) and its type is `boolean`. Clause `INIT` defines the initial state of the module. Here the initial state attribute `light` is false (zero representing false). Clause `TRANS` defines how the state evolves. Axioms in `TRANS` clauses are written using a temporal logic in which `next` is the only temporal operator. `next` is used to reference the next state. The usual propositional operators are also present: `!` stands for logical not, `&`/`|` stand for logical and/or respectively, `->` and `<->` stand for implication and equivalence. Hence in the example the attribute will repeatedly toggle between true and false every time a state change happens.

The complete list of declarations used is as follows:

- `VAR` — allows the declaration of the variables that define the module's state. The types associated with the variables 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.

- `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. They are written using propositional formulae only.

- `FAIRNESS` — allows the specification of fairness constraints. The behaviour of the module (i.e., the states in its execution paths) will have to obey the fairness constraints infinitely often. Fairness constraints can be temporal formulae (CTL formulae) or simply propositional formulae.

- `SPEC` — allows the definition of a CTL formula to be checked.

3.1.2. *CTL*

CTL (Computational Tree Logic — Clarke et al., 1999) is used to express properties of the behaviour of the system specified in SMV. A formal description of CTL is given by (Clarke et al., 1999). An informal account of the operators is given here. Besides the usual propositional logic connectives CTL allows for operators over the computation paths that emanate from a state:

- A – for all paths (universal quantifier over paths);

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

and over states in a computation path:

- G – used to specify that a property holds at all states in the path (universal quantifier over states in a path);

- F – used to specify that a property holds at some state in the path (existential quantifier over states in a path);

- X – used to specify that a property holds at the next state in the path;

- U – used to specify that a property holds at all states in the path prior to a state where a second 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

Model checking of interactor models can be achieved by translating these models into SMV. To accomplish this each interactor's state and behaviour must be expressed in SMV.

3.2.1. *Expressing interactor state in SMV*

SMV has been used in the verification of interactive system specifications by Abowd et al. (1995). Their approach uses a 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 it also has a state (a collection of attributes) and axioms describing how the state evolves. These similarities make it possible to represent 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 on the types used in interactors. A variable defined as having type *nat* will have to be restricted to an appropriate subrange of *nat* before translation to SMV is carried out. With this in mind a translation rule can be defined:

Translation Rule 1. (Attributes)
attributes a_1: 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 SMV where modules can have instances of other modules as part of their state. Instances of included interactors are represented as variables. Their types are the SMV modules that result from the interactor translation. The translation rule for interactor inclusion is:

Translation Rule 2. (Interactor inclusion)
includes i_{name} **via** i_1 *translates to:* `VAR i1: iname;`

where `iname` is the SMV module that results from the translation of interactor i_{name}.

The concept of importing does not exist in SMV. Importing clauses can be eliminated from an interactor description by substituting the imported interactors syntactically. SMV modules cannot be parameterised by types. It is possible to eliminate type parameters from interactor based models by instantiating each parameterised interactor with the types actually used as parameters. Therefore a parameterised interactor will generate an SMV module for each instantiation of parameters. Each of these transformations can be done automatically.

With appropriate restrictions therefore it is possible to represent the state of an interactor in the state of an SMV module. The remaining problem is to express behaviour of interactors in SMV. The remainder of this section deals mainly with showing how a translation from MAL to SMV axioms can be effected. It does this by providing the translation algorithm needed for each type of axiom. The approach taken follows (Fiadeiro and Maibaum, 1991).

3.2.2. *Expressing interactor behaviour in SMV*

Five types of axioms are identified:

- invariants — these are formulae that do not involve any kind of action or (reference) 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 define specific conditions for actions to be permitted to happen.
- obligation axioms — these are deontic formulae involving the use of obl. They define the conditions under which actions become obligatory.

The notion of action in MAL means that the axioms are interpreted over labelled transition systems. Actions are associated with state transitions (Fiadeiro and Maibaum, 1991). This is unlike SMV axioms which are interpreted over (unlabelled) finite state machines (see Section 3.1). We shall use the notation $\mathsf{prop}(expr_1, .., expr_n)$ to denote a formula on expressions $expr_1$ to $expr_n$ using propositional operators only. The expressions $expr_1$ to $expr_n$ themselves need not necessarily be propositional. We will also use names a_1 to a_n to denote interactor attributes.

Invariants These are axioms $\mathsf{prop}(a_1, .., a_n)$. Invariants must hold in all states of the model. SMV has a direct counterpart in 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)`

Initialisation axioms These are axioms $[]\mathsf{prop}(a_1,..,a_n)$ that are used to define the initial state. This also has a direct counterpart in SMV namely the `INIT` clause. Initialisation axioms are translated by removing reference events and placing the resulting axioms in INIT clauses:

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

Modal axioms These axioms are used to specify the effect of actions in the state and are of the form $\mathsf{prop}([ac]a_1,..,[ac]a_g,a_h,..,a_n)$.

Fiadeiro and Maibaum (1991) show how it is possible to reason about the temporal properties of the normative behaviours of deontic specifications[1]. Normative behaviours of interactor models are the ones that are of interest in what follows. Therefore it will be possible to make use of these results to translate modal axioms to SMV.

The *occurrence* operator $>_{action}$ is based on the $>_{\mathtt{T}}$ operator in (Fiadeiro and Maibaum, 1991). This operator is used to signal that a given state has been reached through the occurrence of some specific action. $>_{action}\ ac$ holds in a state when ac is the action that causes the transition to that state.

The operator will be modelled by a state attribute (`action`) indicating the action for which the operator holds true. Hence $>_{action}\ ac$ becomes `action` $=$ `ac` in SMV. The type of this attribute will be an enumeration of all the possible actions. This approach avoids the problem of duplicated initial states described in (Campos, 1999). States are duplicated when they can be reached using different actions (see Figure 6). This is different from the approaches in (Atlee and Gannon, 1993)

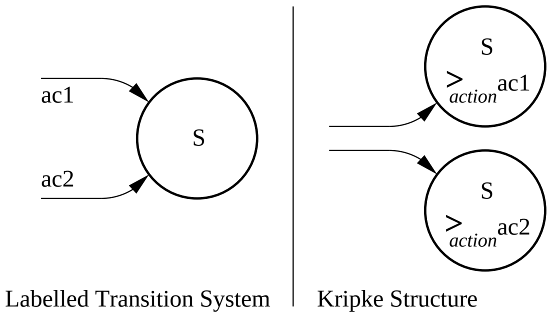


Figure 6. State duplication

and (Abowd et al., 1995) where information is encoded in the state

[1] A behaviour is said to be normative if all permissions are respected and all obligations are fulfilled.

about the next action that will happen. With this type of approach, models that originally had one single initial state with several actions leading from it will be transformed into models with several initial states. This might lead to situations where formulae involving the use of existencial quantification over paths fail on the transformed model even if they are true of the original model. This happens when despite the formula being true of the original single initial state it fails to be true of some of the generated initial states. Remember that to succeed the property must be true of all initial states, and that each of the inital states at the SMV level will capture a subset of the behaviour that is possible from the original single initial state in the model.

Using the occurrence operator the modal operator may be eliminated from modal axioms. Fiadeiro and Maibaum (1991) show the analogue of[2]:

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

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

The modal axiom $[ac]p$ can be written in SMV as `action` = `ac` → `p`. This translation works if all attributes in p are bound to the modal operator (i.e., all attributes are calculated in a single state resulting from the execution of ac). This amounts to a situation where the effect of actions is independent of the states where they occur. A typical modal axiom will use attribute values from both the state prior and after the occurrence of the action in order to express the state transformations generated by the action. Hence the translation must be adapted so that reference to both states is possible. This can be done using the `next` operator. The translation rule for modal axioms becomes:

Translation Rule 5. (Modal axioms)
$\mathsf{prop}([\mathsf{ac}]\mathsf{a}_1, .., [\mathsf{ac}]\mathsf{a}_\mathsf{g}, \mathsf{a}_\mathsf{h}, .., \mathsf{a}_\mathsf{n})$
translates to:
```
TRANS
next(action)=ac -> prop(next(a1),..,next(ag),ah,..,an)
```

Modal axioms are thereby translated into axioms that test whether the next action is the appropriate one and assert the desired property using `next` to reference the state after the action happens.

Permission axioms These axioms are used to restrict action permission to some specific conditions. Permission axioms are of the form

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

$\mathsf{per}(ac) \rightarrow \mathsf{prop}(a_1, .., a_n)$ and mean that action ac is permitted only when the propositional formula $\mathsf{prop}(a_1, .., a_n)$ holds.

Fiadeiro and Maibaum (1991) show that formulae of the form

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

lead to

$$X(>_{action}\ ac) \rightarrow cond.$$

where X is the *next state* temporal operator.

Hence the translation rule for permission axioms is:

Translation Rule 6. (Permission axioms)
$\mathsf{per(ac)} \rightarrow \mathsf{prop(a_1, .., a_n)}$
translates to:
```
TRANS next(action)=ac -> prop(a1,..,an)
```

An SMV model must satisfy its axioms. This translation rule therefore guarantees that all permission conditions will have to be met in the corresponding SMV model in order for the state transition associated with that action to take place.

Obligation axioms These axioms are used to assert the obligation of performing some action. They take the form $\mathsf{prop}(a_1, .., a_n) \rightarrow obl(ac)$ meaning action ac becomes obligatory when the propositional formula $\mathsf{prop}(a_1, .., a_n)$ holds.

Fiadeiro and Maibaum (1991) show that a formula of the form

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

leads to

$$cond \rightarrow F(X(>_{action}\ ac))$$

with F the *sometime in the future* operator.

It is not possible to express this last equation directly in terms of SMV. The SMV input language allows reference to the current and next state only, whereas the equation above makes reference to some arbitrary state in the future. The only way to influence future states of the system is through fairness conditions (see Section 3.1.1). A fairness condition must hold infinitely often. If the formula `next(action)` = `ac` is placed as a fairness condition then eventually the action will happen. This does not impose any limit on how long it will be necessary to wait for the action.

This strategy requires formulae to be added and removed from the set of fairness conditions as obligations are successively raised and fulfilled during the execution of the state machine. Fairness is defined by a static set of formulae in the SMV text. In order to overcome this it is necessary to use a boolean flag signaling when a specific obligation is raised/fulfilled.

For each axiom $prop(a_1, .., a_n) \rightarrow obl(ac)$ we create a boolean variable `oblac` which will represent the obligation. Following from the definition of `obl` in Section 2.2 this variable will be set to true whenever the obligation is raised (i.e., $prop(a_1, .., a_n)$ holds) and not immediately fulfilled. In addition `oblac` must be kept true while the action does not occur. This can be summarised in the following SMV axiom:

```
TRANS
next(action)!=ac -> next(oblac)=(prop(a1,..,an) | oblac)
```

If the next action is `ac` then `oblac` must be set to false:

```
TRANS next(action)=ac -> !next(oblac)
```

Initially the variable is set to false. In fact the variable is only set to true when the obligation is raised and not immediately fulfilled.

```
INIT !oblac
```

Finally, a fairness clause is added stating that `oblac` must be false infinitely often:

```
FAIRNESS !oblac
```

This guarantees that whenever an obligation is signalled it is eventually fulfilled.

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.

3.2.3. *Some final comments regarding the translation*

The discussion above has only considered actions with no parameters. It is possible to eliminate a parameterised action automatically by substituting it by a set of actions, one for each possible combination of the parameters' values. Parameterised actions can appear in three types of axioms. Only universally quantified variables are accepted as parameters.

– *modal axioms.* Axioms are repeated as many times as needed by instantiating the parameterised actions with the appropriate values.

– *permission axioms.* Axioms are repeated as many times as needed.

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 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:
$prop([ac]a_1, .., [e]a_g, a_h, .., a_n)$
	
	`TRANS next(action) = ac ->`
	`prop(next(a1), .., next(ag), ah, .., an)`

permission axioms:
$per(ac) \rightarrow prop(a_1, .., a_n)$
	
	`TRANS next(action) = ac ->`
	`prop(a1, .., an)`

obligation axioms:
$prop(a_1, .., a_n) \rightarrow obl(ac)$
	
	`VAR oblac : boolean;`
	`INIT !oblac`
	`TRANS next(action)!= ac ->`
	`next(oblac) = (prop(a1, .., an) | oblac)`
	`TRANS next(action) = ac->!next(oblac)`
	`FAIRNESS !oblac`

– *obligation axioms.* Any of the actions generated by the rules above will discharge the obligation.

Another feature of the interactor language is the ability to give names to enumerated types. This is not possible in SMV but type names can be eliminated by substituting all occurrences of a type name by its definition.

Two additional clauses are added to the language. Clause **fairness** allows the definition of fairness constraints and 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.

The focus has been individual interactors. It is assumed that the semantics of combining interactors and of combining SMV modules are the same. This is true except for one problem. By default SMV modules work in lock-step. Whenever a module performs a transition all its children (that is all module instances declared as variables of its state) must perform a transition. It is necessary to model the fact that interactors can evolve independently subject to explicit synchronisations in

SMV. SMV provides the *process* as a mechanism for interleaved execution (McMillan, 1993). Whenever an SMV module contains instances of other modules as its children the keyword `process` can be used to make that child run independently of the parent's behaviour. The SMV semantics of processes is too restrictive. No two processes with the same parent can be running simultaneously. This interpretation prevents modules (and the interactors that they represent) from synchronising on some action. To overcome this it is necessary to introduce stuttering in the SMV modules explicitly. A *stutter* means that modules can perform state transitions in which no actions actually happen. A special action `nil` is introduced along with axioms stating that this action does not change the state of the module. A module can thereby perform an action while another module does nothing. By this means it is possible to simulate the behaviour of both modules performing actions simultaneously.

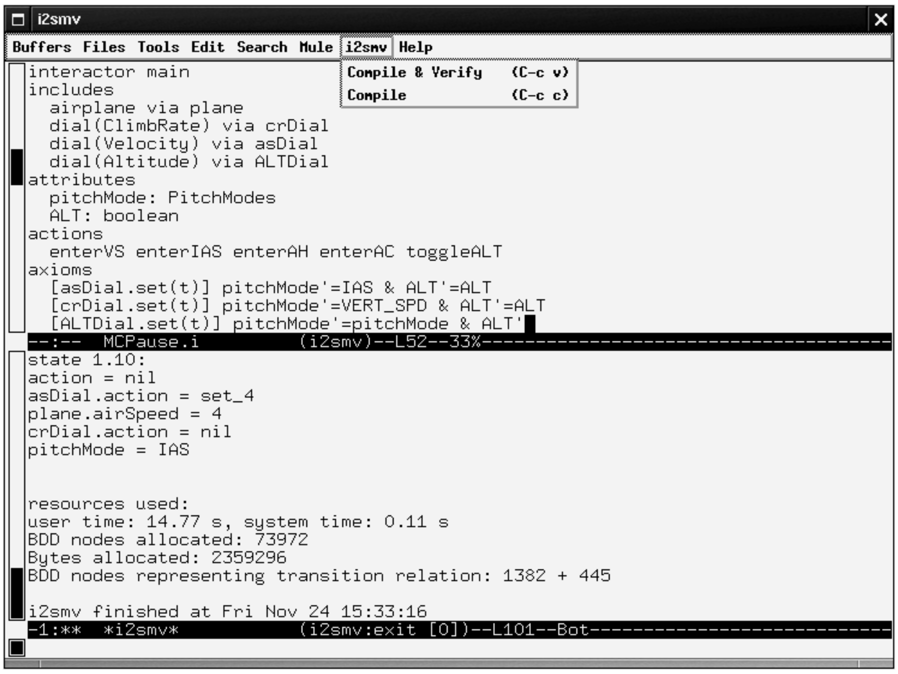
In order to refer to the enabledness of an action in the test clause operator enbl is introduced. The expression enbl(ac) is translated into `E[action=nil U action=ac]` (an action is enabled if there is a computation path where no action happens until action ac happens).

3.3. The tool

A tool to implement the translation just described has been implemented in Perl (see Wall et al., 1996 for a description of the language). The Perl script (`i2smv`) works by reading an 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 by receiving interactor code as input and generating SMV code as output. A file can also be provided for the input. Supposing an interactor model is contained in file `model.i` the command:



```
interactor main
includes
  airplane via plane
  dial(ClimbRate) via crDial
  dial(Velocity) via asDial
  dial(Altitude) via ALTDial
attributes
  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'
--:-- MCPause.i          (i2smv)--L52--33%-----------------------------------
state 1.10:
action = nil
asDial.action = set_4
plane.airSpeed = 4
crDial.action = nil
pitchMode = IAS


resources used:
user time: 14.77 s, system time: 0.11 s
BDD nodes allocated: 73972
Bytes allocated: 2359296
BDD nodes representing transition relation: 1382 + 445

i2smv finished at Fri Nov 24 15:33:16
-1:** *i2smv*            (i2smv:exit [0])--L101--Bot--------------------------
```

Figure 7. The Interactors to SMV compiler

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

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

An Emacs (Stallman, 1998) 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 while the bottom pane shows the result of SMV model checking. The option i2smv on the menu bar provides two alternatives for processing model descriptions.

- 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 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 file is then opened in Emacs using the SMV mode.

4. Modelling the MCP with Interactors

We have already noted that the Palmer (1995) case study deals with a problem relating to altitude acquisition in a real aircraft (MD-88). Although this particular problem was identified during simulation of realistic flight missions, Palmer notes that similar problems are frequently reported to the Aviation Safety Reporting System (Cheaney, 1991).

4.1. Basic principles

Operators of automated systems build mental models that lead to expectations about system behaviour. When the system behaves differently from the expectations of the operator, *automation surprises* (Woods et al., 1994) occur. This type of problem is concerned with how the system and user interact rather than with the behaviour of the system alone. In the present example the system behaved as designed (i.e., it did not malfunction) but nevertheless an automation surprise happened. The system is misleading operators into forming false beliefs about its behaviour. Because it is difficult to prejudge the behaviour of the system in the context of use, simulations of real-life interactions are required with real users.

Although the use of simulation allows for the detection of some shortcomings in design it also has some intrinsic problems. A full system or prototype has to be built which is costly late in the design/development life cycle when design decisions have already been made.

An ability to analyse and predict potential problems during the initial stages of design would reduce the number of problems found later in the simulation stage. This early analysis must be done without undue bias from hindsight. We are not trying to explain why something went wrong. Rather we want to exhaust the set of potential sources of problems.

The problem with the example of Palmer (1995) is that it is based precisely on hindsight. Even though it is difficult to be untainted by this previous experience we will attempt to build a generic model of the artefact under consideration. We will then analyse those aspects of the behaviour that are highlighted by the case study and hence attempt to demonstrate that it is possible to detect the problem and to prevent it from creeping into the design. We shall argue that the model and the questions we ask could have been generated without the benefit of experience of the Palmer scenario.

The model will focus mainly on what is relevant in that dialogue between the user and the artefact and will not dwell on the details of either the artefact or the user. 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. The model focuses on the key actions and the parameters that are presented by the interface in a way that is consistent with the more detailed description of the artefact.

In summary, 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 of the results described in the scenario. The scenario could be considered to be an idealised description of how the system should function in a particular situation in order to guide the system development.

4.2. Selecting what to analyse

The first step in the process, see Figure 1, 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 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).

These requirements and properties are related to the user and therefore 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. Such a discussion might have a critical impact on the system safety or may have consequences that are unclear to the designer.

In the present case the issue is how 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 as will 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

To analyse a system we need to place it in its context of operation. The MCP is not intrinsically unsafe. 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. We will abstract from this and use 1 as a 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.

We must be careful that the abstraction process above does not affect the behaviour of the system as it relates to properties we will be checking. Altitude steps must be 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

As we have argued, modelling the MCP (see Figure 9) involves taking account of the specific analysis we want to perform. In this case we want

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

Figure 8. The aircraft

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 of operation of the pitch modes.

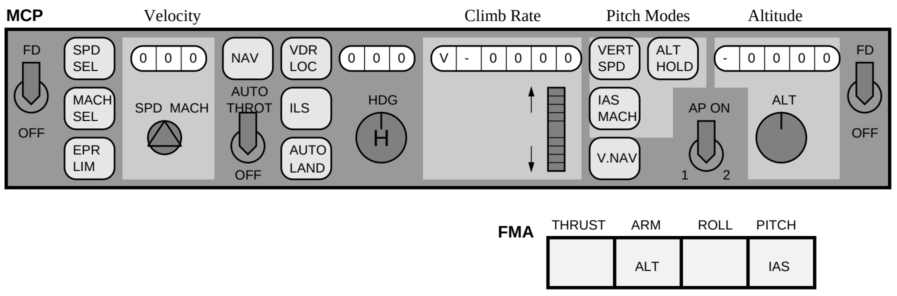


Figure 9. The MCP (adapted from Honeywell Inc., 1988)

The components that were deemed relevant are shown in Figure 9 in a lighter background. The choice of encoding will typically have been carried out through discussion with human factors and domain experts. 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 can be discharged by a separate proof process. Hence we are able to assess decisions relating to particular design aspects in a compositional manner.

Three main dials are involved (see Figure 9):

– airspeed (velocity);

– vertical speed (climb rate);

– altitude window (i.e., altitude to which the aircraft should climb).

Airspeed and altitude can only be positive values. The vertical speed can either be positive (going up) or negative (going down). The parameterised interactor introduced in Section 2.2 (see Figure 3) allows us to represent the different dials economically. Dials are represented abstractly by an attribute (needle) and an action (set). The action corresponds to setting a value (see 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).

We therefore define the type:

$\mathsf{PitchModes} = \{\mathsf{VERT\_SPD}, \mathsf{IAS}, \mathsf{ALT\_HLD}, \mathsf{ALT\_CAP}\}.$

Additionally there is a capture switch (ALT) which can be armed to cause 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. Setting the climb rate or airspeed causes the pitch mode to change accordingly (Axioms 1 and 2). 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. These allow changing between different pitch modes and toggling the altitude capture. Action enterAC (setting the pitch mode to ALT_CAP) is an internal system event (it is not annotated with $\boxed{\text{vis}}$) which therefore cannot be invoked directly by the user. Axioms 9 and 10

interactor MCP
includes
  aircraft **via** plane
  dial(ClimbRate) **via** crDial
  dial(Velocity) **via** asDial
  dial(Altitude) **via** ALTDial
attributes
  $\boxed{\text{vis}}$ pitchMode: PitchModes
  $\boxed{\text{vis}}$ ALT: boolean
actions
  $\boxed{\text{vis}}$ enterVS, enterIAS, enterAH, toggleALT
  enterAC
axioms
  # Action effects
  (1) [crDial.set(t)] $\text{pitchMode}'=\text{VERT\_SPD} \wedge \text{ALT}'=\text{ALT}$
  (2) [asDial.set(t)] $\text{pitchMode}'=\text{IAS} \wedge \text{ALT}'=\text{ALT}$
  (3) [ALTDial.set(t)] $\text{pitchMode}'=\text{pitchMode} \wedge \text{ALT}'$
  (4) [enterVS] $\text{pitchMode}'=\text{VERT\_SPD} \wedge \text{ALT}'=\text{ALT}$
  (5) [enterIAS] $\text{pitchMode}'=\text{IAS} \wedge \text{ALT}'=\text{ALT}$
  (6) [enterAH] $\text{pitchMode}'=\text{ALT\_HLD} \wedge \text{ALT}'=\text{ALT}$
  (7) [toggleALT] $\text{pitchMode}'=\text{pitchMode} \wedge \text{ALT}' \neq \text{ALT}$
  (8) [enterAC] $\text{pitchMode}'=\text{ALT\_CAP} \wedge \neg\text{ALT}'$
  # Permissions
  (9) $\text{per(enterAC)} \rightarrow (\text{ALT} \wedge |\text{ALTDial.needle - plane.altitude}| \leq 2)$
  # Obligations
  (10) $(\text{ALT} \wedge |\text{ALTDial.needle - plane.altitude}| \leq 2) \rightarrow \text{obl(enterAC)}$
  (11) $(\text{pitchMode}=\text{ALT\_CAP} \wedge \text{plane.altitude}=\text{ALTDial.needle}) \rightarrow$
      obl(enterAH)
  # Invariants
  (12) $\text{pitchMode}=\text{VERT\_SPD} \rightarrow \text{plane.climbRate}=\text{crDial.needle}$
  (13) $\text{pitchMode}=\text{IAS} \rightarrow \text{plane.airSpeed}=\text{asDial.needle}$
  (14) $\text{pitchMode}=\text{ALT\_HLD} \rightarrow \text{plane.climbRate}=0$
  (15) $(\text{pitchMode}=\text{ALT\_CAP} \wedge \text{plane.altitude}<\text{ALTDial.needle}) \rightarrow$
      $\text{plane.climbRate}=1$
  (16) $(\text{pitchMode}=\text{ALT\_CAP} \wedge \text{plane.altitude}>\text{ALTDial.needle}) \rightarrow$
      $\text{plane.climbRate}=-1$

Figure 10. The MCP model

specify the mode logic that regulates that this event happens when the altitude capture is armed and the plane is inside some neighbourhood of the target altitude. The restriction on the size of the neighbourhood is that it should not be too small to allow the system to evolve (have behaviour) while inside the neighbourhood of the target altitude and therefore it has been specified as the value 2. Axiom 11 specifies that the system must set the pitch mode automatically to ALT_HLD once the desired altitude has been reached. Finally, Axioms 12 to 16 describe the effect of the pitch modes on the state of the aircraft.

The property under analysis relates to the temporal behaviour of the model. Model checking is the therefore the natural choice of technique to be used (Campos and Harrison, 1998). Before we can apply this approach two further steps are necessary. We need to obtain a checkable version of the model and we must define how the properties can be expressed in CTL.

5. Checking the Design

Having developed a model for the MCP we will now analyse it using SMV and the tool described in Section 3.

5.1. Converting the Model

Every verification technique or tool forces restrictions on what can be analysed and how. SMV requires some adjustment to be made to the model we have developed to cut down the number of states. The most relevant is the need to have enumerated types in the specification only. Altitude and velocity provide no problem. The aircraft will have its own physical limitations on maximum speed and altitude and we need to ensure 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. Three situations characterise climb rate in this situation: climbing, holding altitude or descending. These will be represented as three values: -1 (to represent all negative climb rates), 0, and 1 (to represent all positive climb rates).

Abstraction implies removing information from the model which can lead to situations were properties can be proved of the abstracted model which are not true of the original model (false positives). The abstraction process above is similar to the Application State abstraction in (Dwyer et al., 1997). (Dwyer et al., 1997) shows that if the properties to be checked are universally quantified then false positives are not introduced by the abstraction process.

In summary we use the following types:

Altitude = {0, 1, 2, 3, 4, 5}
Velocity = {0, 1, 2, 3, 4, 5}
ClimbRate = {-1, 0, 1}

The behaviour of the interactor plane is modified to take into account the maximum and minimum altitudes (see Appendix A).

The use case we are considering deals with altitude acquisition and therefore it is not necessary to include negative (below sea level) altitudes in the model. The minimum value for altitude is zero. The specification could be extended trivially to include negative altitudes. Only the definition of Altitude and the minimum altitude in the plane interactor would need to be adapted to the new minimum value.

To make it compatible with the SMV checker the name of interactor MCP must be changed to main. The checkable version of the specification is presented in Appendix A which is automatically convertible to SMV using the compiler.

Having translated our model to SMV we now have to express the properties as CTL formulae that 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

The design of the interface (as seen in Section 4.2) has been based on the plausible assumption that if the altitude capture (`ALT`) is armed the aircraft will stop at the desired altitude (selected in `ALTDial`). This can be expressed 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 (AF) the altitude will always be reached and the pitch mode be changed to altitude hold.

The CTL formula above is somewhat weaker than the pilot's expectation introduced in Section 4.2 but it subsumes interesting properties of the interaction between a user and the MCP. Section 7 further discusses this issue.

Only two pieces of information are needed about how the translation from interactors to SMV works in order to express CTL formulae and to interpret the traces that provide counterexamples. Knowledge is needed of the existence of the occurrence operator (attribute `action` at the SMV level) and of the convention that the expression `ac_v` at the SMV level represents expression ac(v) at the interactor level.

When we model check a specification the checker answers whether or not the test succeeds. When we check the model against the formula above we get the following trace as counterexample[3]:

```
-- 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

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

resources used:
user time: 167.1 s, system time: 0.49 s
BDD nodes allocated: 936879
Bytes allocated: 16121856
BDD nodes representing transition relation: 1952 + 915
```

[3] Note that from state to state only those values that have changed are shown. For brevity we only show enough of the counter example to make the point.

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. It suggests changes that cause the aircraft to deviate from the target altitude.

There is not enough detail in the model to make this point clearly so we need to refer back to the designers in order to raise the point. The model could then be refined if necessary 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. Rather it prompts questions that have to be dealt with at other stages of design.

If the model is appropriate in this respect it will lead to a refinement of the assumptions about the user. A revised property must reflect the fact that changing the climb rate can prevent the aircraft from reaching the desired altitude. This process of refining the formulae is an important component of the verification process and it is one that should involve the insight and analysis of human factors and domain experts. This refinement process has the effect of incorporating knowledge about the user into the proof.

In the light of the counterexample produced by the check of the first formula 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.

When we try this property the answer is still no. The model checker points out that changing the pitch mode to VERT_SPD (for instance by setting the corresponding dial) when in ALT_CAP terminates the request to stop climbing at the target altitude. When the pitch mode changes to ALT_CAP the altitude capture is automatically switched off (see Axiom 8) even though the aircraft is still climbing. Any subsequent action from the pilot that causes the pitch mode to change will cause the aircraft to keep climbing past the target altitude. This is an alert to the designer. Is it a desirable state of affairs? Is enough information provided by the MCP to alert the pilot? Here discussions with human factors experts may provide help about whether this situation is likely to be problematic. Hence the property in itself does not encode user expectations. Rather failure to satisfy the property may generate the basis for a scenario that may lead to a concern about human issues.

(Palmer, 1995) reports that a similar problem was detected during simulation. 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. We gave SMV no specific chain of events to analyse nor did we check explicitly for the absence of altitude busts. The analysis revolved around a simple generic use case concerned with altitude capture. It was the tool that pointed to a particular sequence of events that could lead to this hazardous situation. This could have been achieved as an automated verification process based only on a pen and paper scenario of an aircraft in its early design stages.

Finding a problem is just a trigger for further analysis and discussion. The designers and human-factors experts can be called upon to clarify the full consequences of the counterexample. 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 will undoubtedly allow for a better/safer design from the start. It will also reduce downstream costs of failing to discover these problems until too late.

6. Related Work

The use of automated reasoning tools for software verification has attracted considerable interest in recent years. In this paper we have considered model checking interactive systems' specifications for the verification of interactive systems designs. This work is comparable with a number of recent papers.

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. This can lead to more readable specifications. The possibility of performing the analysis in an automated manner however 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 notation (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).

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. 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 we are convinced that the methodology can also be applied to the analysis of other issues. For example, task related properties, lock-in and interlock issues, or awareness can be analysed in this way. 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 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 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 also be used downstream of these abstract models.

SMV is also used by (Chan et al., 1998) for the verification of requirements specifications. They do this by analysing RSML (Requirements State Machine Language) specifications. Here the translation to SMV is not necessarily semantics-preserving hence SMV models may be generated whose semantics differ 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 rather than the specific behaviour of the system. While it is obvious that the system must behave correctly this is clearly not enough. It is also necessary that system and user interact effectively. Our work can be seen as complementary with work in requirements verification.

SCR and RSML have been used with success for the specification of safety critical systems. MAL as a specification language provides deontic operators for permission and obligation that allows the specification of more complex behaviour patterns while retaining a degree of readability and ease of use. Experience has shown that the behaviour of MAL based interactor models are mostly based on the notion of action. Permission axioms assert the conditions for the actions to be valid. Modal axioms assert the effect of the action on the state. These axioms are consistent with the traditional style of specifying a system using pre- and post-conditions for the possible actions.

MAL based models can be translated into SMV in a fully automated manner. In all the other approaches there is some degree of manual intervention. Automated translation is crucial to avoid undetected translation errors introduced through human intervention.

Chan et al. (1998) discuss 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. 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 task analysis output. The translation process from a LOTOS specification to a finite state machine implies that information will be lost. Conditional guards are systematically removed in this process which causes the checkable version of the specification to admit more traces of behaviour than the original LOTOS version. Approaches have been proposed that involve using particular styles of specification to avoid these difficulties (Paternò, 1995) or a manual translation of the specification (Palanque et al., 1996).

Properties are here expressed in an action based notation ACTL (Nicola et al., 1993). We believe CTL is a better choice because actions can be encoded as state attributes quite simply but encoding state information as actions is rather complicated and not amenable to automation. LOTOS specifications are architectural descriptions of the user interface that derive from the task analysis which makes it more difficult to reason about the relation between interface and system behaviour. Our iterative approach to verification allows simpler models. The attributes that are chosen to be modelled reflect user concerns, focusing on actions and display attributes that are relevant to the user. Properties of these simpler models thereby reflect user concerns to some degree. The models and properties presume a minimum about user processes. Consequences of the model viewed from a human factors perspective may however be of concern and may involve some analysis of human processes by those experts. A task based approach can be used to generate properties for verification but there is danger in such an approach that the task will over prescribe what the operator does. Humans do not follow procedures instruction by instruction in general and problems often arise as a result of deviations from normative behaviour.

Bumbulis et al. (1996) reports on the use of the HOL theorem prover for interactive systems verification. The approach deals with properties of the interface at the device level and is rather restrictive in the properties that can be verified. Only safety properties of the relationships between the devices present at the interface level are considered. There is no mention of the underlying functionality, nor of the user. The analysis that we describe occurs earlier 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 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

We have looked at the automated verification of early specifications of interactive systems. Interactive systems are complex systems which pose difficult challenges for 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.

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 a model of the artefact was built. We used CTL formulae to capture use cases about the operation of the artefact. Issues were raised about the behaviour of the system as a result of the process of verifying the formulae. Scenarios were found where the system did not behave as expected. The analysis of these scenarios acted as a focus for further interdisciplinary discussion regarding the meaning of the scenarios 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.

As a conclusion to the paper we shall briefly address a number of common objections to the approach as well as some restrictions that are on the agenda for future work.

– As pointed out in Section 5.2 the property eventually verified was weaker than the initial assumption about the pilot's expectations introduced in Section 4.2. The CTL formula in the example specifies only the initial and final states of the scenario being considered while the textual version (implicitly) mentions the intermediate states of behaviour. Hence it is fair to point out that the CTL formula could under some circumstances check even though the altitude problem did occur. This would happen in a situation where the aircraft would go above the altitude set in the altitude dial and then come back to the appropriate altitude and stop at it. This raises two points. First of all it is important to

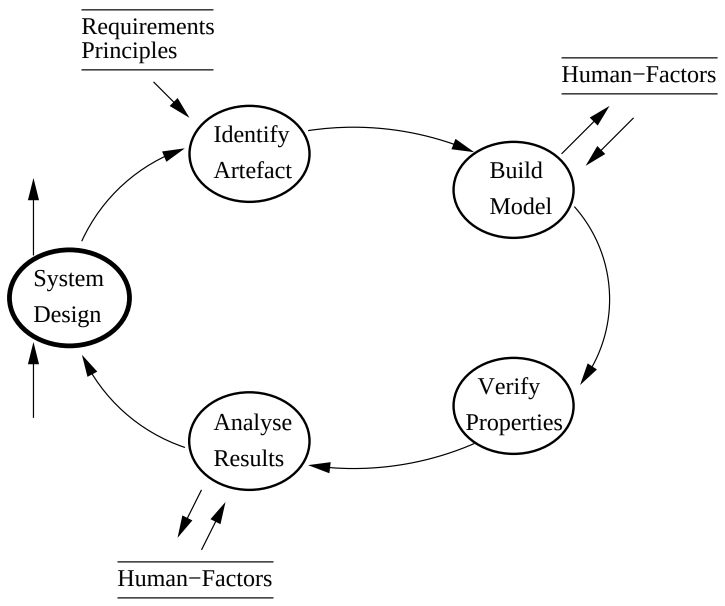


Figure 11. The verification process revisited

clearly understand what is being verified. We were not expressly looking for an "altitude bust" situation. Rather we were checking the system against a use case (that the aircraft would stop at the desired altitude) in order to detect unforeseen consequences of the design. Second, the obligation operator (obl) raises an obligation for an action to occur but imposes no restrictions on how long that might take. Alternative semantics where an obligation had to be fulfilled immediately have been found too restrictive. One possible way around this is to introduce axioms of shape obl(ac)$\rightarrow$cond. What this type of axiom allows us to express is that once action ac becomes obligatory it must happen before $\neg$cond. It is then possible to express that an obligation to enter altitude capture pitch mode must be fulfilled before the aircraft goes past the target altitude (obl(enterAC)$\rightarrow$ALTDial.needle<plane.altitude). Use of this type of axiom must be exercised with care since they impose very strong restrictions on the behaviour of the system.

– One problem in relation to model checking is to find a model that is sufficiently expressive while consisting of a manageable 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 and therefore a goal of a partial model should be to restrict the size of the model to be manageable and yet accurate in describing the appropriate features of the system.

- The use of interactors and SMV gives us a degree of freedom and expressive power that comes with some cost. CTL, although allowing for the expression of possibility, raises fairness concerns. In the case study above we could have a situation where the pilot repeatedly sets the climb rate of the aircraft to zero thereby preventing it from reaching the altitude set in the capture. Situations of this kind can be solved either by: altering the property; using fairness constraints on the system; or reworking the specification with the particular context of analysis in mind.

- Another potential problem with model checking is the size of the counter examples that are generated by the tool. Our experience has shown the counter examples to be small (tens of states). We believe this is due to the use of partial models of systems. Partial models of course have their own problems: are we using the appropriate scenarios and abstractions? This is not just a problem of partial models, 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 it would be up to us to decide what questions to ask of that specification. We would always 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 reason about such issues rigorously, and to prove formally whether the criteria are met or not early in the design cycle.

- A 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. There are even situations where discrepancies between different models can be used to detect problems in the design.

- Another question that can be asked is whether it will always be feasible to encode mode surprises as invariants over the states of the model. Our aim has not been to verify the system *exclusively* for mode surprises. We are mainly concerned with the interaction between user and system, and it is this interaction process that we wish to analyse. Given a system (interface) design and a typical use case we want to investigate the interaction process between user and system in terms of the use case. During this process it becomes possible to detect problems in the interaction. As it has been shown, a mode surprise is one such type of problem which can be detected. In conclusion, the point is that we do not really want to model "mode confusion" we want to be able to detect it. We hope to have shown that use cases give us the possibility of doing this by providing counter examples that can form the basis for analysis by designer and human factors experts.

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

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

Cheaney, E.: 1991, 'ASRS Introduces...'. *ASRS Directline* (1). http://asrs.arc.nasa.gov/directline.htm.

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: W.-P. de Roever, H. Langmaack, and A. Pnueli (eds.): *Compositionality: The Significant Difference*, Vol. 1536 of *Lecture Notes in Computer Science*. Springer, pp. 1–22.

Doherty, G., J. C. Campos, and M. D. Harrison: 2000, 'Representational Reasoning and Verification'. *Formal Aspects of Computing* **12**(4), 260–277.

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.

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.

Stallman, R.: 1998, *GNU Emacs Manual.* Free Software Foundation, 13th edition.

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. The compiler is line oriented hence each expression must be fully contained in a single line. Line breaks can be escaped with the backslash character allowing for 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
attributes
  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.altitude<ALTDial.needle) -> \
                                       plane.climbRate=1
  (pitchMode=ALT_CAP & plane.altitude>ALTDial.needle) -> \
                                     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 Creissac Campos
Universidade do Minho, Departamento de Informática
Campus de Gualtar
4710-057 Braga, Portugal
e-mail: jose.campos@di.uminho.pt
fax: +351 253 60 4471