Automated Deduction and Usability Reasoning

José Francisco Creissac Freitas de Campos

Submitted for the degree of Doctor of Philosophy

The University of York
Human-Computer Interaction Group
Department of Computer Science

1999

Abstract

This thesis addresses the applicability of automated deduction techniques in performing usability reasoning. Usability relates to how easily users will be able to operate a system. Hence, much of usability reasoning is done by placing users in front of the system and analysing how they perform. (Formal) automated reasoning, on the other hand, is based on mathematics, and its aim is to enable precise reasoning about the world, from a model of that world. Automated reasoning is therefore without direct observation. The possibility of performing usability reasoning from early in the development process will contribute to a better understanding of the design, and reduce the problems found in later stages of development. The use of formality, will enable proof that certain desirable system properties are present. The use of automation will contribute to the quality of the analysis, and allow the analysis of more complex systems.

The thesis proposes an approach to verification which allows for a closer integration of verification into the interactive systems development life-cycle. This approach is based on a move towards a feature centric approach to verification. By promoting an integration of user oriented concerns into the verification process, the approach enables reasoning about human-factors related issues in a primarily software engineering oriented context.

A framework for usability related properties is also proposed. This framework is used to understand what is involved in reasoning about usability properties. A tool is developed which enables the verification of MAL-based interactor specifications in the SMV model checker. Examples are given of the application of both model checking and theorem proving to the analysis of usability properties, in the context of the proposed approach. These examples include analysis of how interface layout will support the users tasks, of whether interface design principles such as predictability are followed, and of mode complexity issues.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Professor Michael Harrison, whose constant guidance and support have made this thesis possible. I feel very lucky to have been his student.

I had the opportunity to discuss my work with many people during my time at York. Special thanks are due to Gavin Doherty, Karsten Loer, Nick Merriam, Bob Fields, and James Willans. I would also like to thank everyone in the HCI group for a stimulating and friendly working environment.

Thanks are also due to my family and friends, both old and new, for helping me to feel home away from home. Of the new friends, special thanks go to Eraldo, Paulo, and also Ronaldo (for the Jazz).

I must also thank the Fundação para a Ciência e a Tecnologia (formerly Junta Nacional de Investigação Científica e Tecnológica) for financial support under grant PRAXIS XXI/BD/9562/96, and the University of Minho (especially everyone at the SIM group) for the three years leave which enabled me to come to York. Additionally, I would like to thank the Fundação Caloust Gulbenkian for financing my attendance to COMPOS'97.

Finally, I owe a great debt to my future wife, Júlia, for all her patience, support and love.

Author's declaration

Much of the work presented in this thesis has already been published elsewhere, co-authored with Michael Harrison (Campos & Harrison 1997, Campos & Harrison 1998, Campos & Harrison 1999*a*, Campos & Harrison 1999*b*, Campos & Harrison n.d.), and also with Gavin Doherty and Michael Harrison (Doherty et al. 1998, Doherty et al. 2000). In all cases, I have presented only those aspects of the work which are directly attributable to me.

To Júlia

"A Estrela Polar és Tu,
A Estrela segura, Tu"
(Estrela Polar, author unknown)

Chapter 1

Introduction

"Only science can keep technology in some sort of moral order"

Edgar Friedenberg quoted in (Thimbleby 1990)

This thesis is about the use of automated deduction techniques in usability reasoning. How feasible is it to use mathematically based techniques to analyse how a system will be used? How can this be done from the early stages of design? The thesis is therefore taking a software engineering perspective on Human-Computer Interaction (HCI). The point of intersection between software engineering and studies of human behaviour is a challenging one. This chapter deals with explaining why this is an interesting area of research, and introduces the main concepts that will be needed throughout the thesis.

1.1 The need to (formally) reason about usability

It has become commonplace, whenever talking about Human-Computer Interaction, to mention the pervasiveness of computers in all areas of society, as well as the great share of development and maintenance costs which is attributable to the development of the user interface (see, for example, Preece et al. 1994, Newman & Lamming 1995). (Preece et al. 1994, Chapter 1) gives several examples of the impact that HCI considerations (or lack thereof) will have in the development and deployment of interactive systems.

However, development and maintenance costs cannot be measured simply in money. Interactive systems have growing application in safety critical environments, aircraft cockpits and nuclear power plants, for example. In the case of safety critical systems, bad quality can cause unacceptable damage up to the loss of human life. This raises the need to get design right from the start.

It is also true that design is mostly led by technology. The current rate of technological development means that the actual implications of the introduction of new technology are not always completely understood before the technology is put into practical use[1]. In many cases, when an accident or near miss happens, the blame ends up attached to the operator of the system, the user. As Woods et al. (1994) point out, this is not always so clear cut. Many times the *user error* is a natural consequence of bad system design. There is clearly a need to be able to assess and guarantee the quality of the designed software.

Reasoning about design, in the context of interactive systems, implies reasoning about how useful and easy to use those systems will be (or, at least, are), i.e. evaluating their *Usability*. Usability evaluation methods can be divided into two groups (see Newman & Lamming 1995, Part III):

- Empirical — these rely on building prototypes of the system being developed and testing them using real users under controlled conditions.
- Analytic — these rely on confronting models of the system with how users are expected to behave.

Factors influencing the usability of a system range from pure software engineering concerns to psychological, sociological, or organisational concerns. Hence, in most situations, only actual deployment of the system will give a final answer towards its usability. It is somewhat paradoxical, therefore, that in some situations, namely where

[1] The explosion of Arianne 5 during its maiden launch was due, in part, to a misconception in the application of redundancy to circumvent system failure. The notion of redundancy used embodied a hardware-biased point of view (if a part breaks its function must be performed by a backup part) which proved inappropriate for a software system (replacing a *faulty* routine by a backup version of the same routine did not solve the underlying problem. Based on the same input the *new* routine produced the same output which was found faulty to start with).

safety-critical systems are involved, empirical evidence might not be enough to support claims about the usability (and safety) of a system. Moreover, empirical methods are expensive to apply, and require a lot of organising and time. Additionally they are difficult to apply in the early stages of design when most decisions have to be made. Analytic methods fill this gap. By being based on models of the system, and not requiring a prototype to be built and placed in a plausible interaction context, analytic methods have the potential to allow reasoning about the usability of a system to be carried out early in the design process. Their drawback is that assumptions have to be made about user behaviour, and the results can only be as good as the assumptions that are made.

Traditional analytic methods tend to be carried out manually and more or less informally (Newman & Lamming 1995). This can work in areas where the technology is well known, and the design of the system is not too complex. However, as systems (hence, models) become more complex, this type of approach becomes less likely to deliver. Formal methods have long been proposed as a means of achieving a degree of leverage over complexity. It is therefore only natural that the use of formal methods for interactive systems design has become an area of active research (Harrison & Thimbleby 1990, Abowd 1991, Harrison & Duke 1995)

The use of formal (mathematical) models during development can impact system design at two levels:

- the use of rigorous mathematical concepts and notations can help in the organisation and communication of ideas;
- mathematical models can allow rigorous reasoning about properties of the system being designed.

Although the modelling process itself can provide valuable insight into the system being designed, only the possibility of formally reasoning about the models can bring the use of formal models into full fruition. Hence, in recent years formal verification of interactive systems has, too, become an active area of research (see, for example, Markopoulos & Johnson 1998). The main goal of this research has been to enable formal reasoning about properties of interactive systems specifications. Of particular

interest in this area is the use of automated deduction tools to assist the reasoning process. Since the systems under consideration are interactive systems, this reasoning should also address usability issues. This, then, is the central proposition of the thesis:

> Automated reasoning tools can be used in the usability analysis of interactive system designs.

The remainder of this chapter will introduce the main concepts involved in the proposed area of research. Section 1.2 clarifies what is meant by the term formal verification. Section 1.3 explains the challenges faced when attempting to reason about usability. Section 1.4 discusses the different roles involved in interactive systems design and analysis. Section 1.5 introduces the main approaches to automated reasoning. Finally, Section 1.6 presents an overview of the thesis proper.

1.2 Formal Verification

The development of a software system usually starts with the identification of a problem to be solved. For this problem a solution will be devised, and from this solution a model of a system which implements it will be generated. This model can be called the specification of the system (for a discussion on the role of models see Fields et al. 1997). The type of specifications used during design can vary from an informal textual description, up to a formal specification in some mathematical notation. Whatever type of specification is used, it should capture the essential aspects of the system in a precise, clear and concise way. As the level of complexity grows, so does the demand put upon the expressive power of the specification notations. In this context, essential features of specification notations are their capability of abstraction and of structuring the specification into tractable meaningful units. It is clear that for complex systems, informal textual descriptions will not be appropriate, and it is commonly argued that formal mathematical notations should be used (see, for example, Jones 1980).

Once a satisfactory specification has been achieved, the process then proceeds to generate a software system which implements the specification. Hence, the issue of ensuring that a software system adequately solves the original problem, encompasses two basic questions:

- does the specification adequately represent the solution?
- does the software system adequately represent the specification?

Proving that a program is correct against its specification is, in principle, a decidable problem. The specification acts as an absolute measure of *quality*[2]. Proving that the specification is a correct representation of the (intended) solution, on the other hand, is not mathematically possible (Jones 1992, page 21). This springs from the fact that in this case there is no such measure of quality. To have it, some representation of the solution against which to compare the specification would be needed. But the specification is *the* representation of the solution.

Not being able to prove the correctness of the specification is, nevertheless, different from not being able to reason about its quality. In fact, the position that *fitness for purpose* cannot be addressed seems rather extreme. A third question can be raised:

- how good is the proposed solution?

Assuming the specification adequately represents the solution, this can be reinterpreted as the question: how good is the specification? Again, an absolute answer cannot be achieved. However, it is still possible to identify desirable properties of the system (measures of quality), and challenge the specification with theorems expressing those properties. If the theorems can be proved, then it has been verified that the specification has those properties. If the specification and proofs are done in some formal, mathematically based notation, then the verification is said to be formal. Obviously this begs the question of whether such properties adequately represent the purpose of the system, so the problem has not gone away. What has been gained is that it has been circumscribed to a set of properties, instead of the system as a whole.

This process of exploring the specification with theorems representing properties to be verified is called (formal) specification verification (or validation). This is clearly different from formal program verification, the process of formally proving that a given system satisfies a specification, which is the traditional area of verification (see Loeckx & Sieber 1984, Jones 1992).

[2] In this context quality means the conformance of the program to its specification.

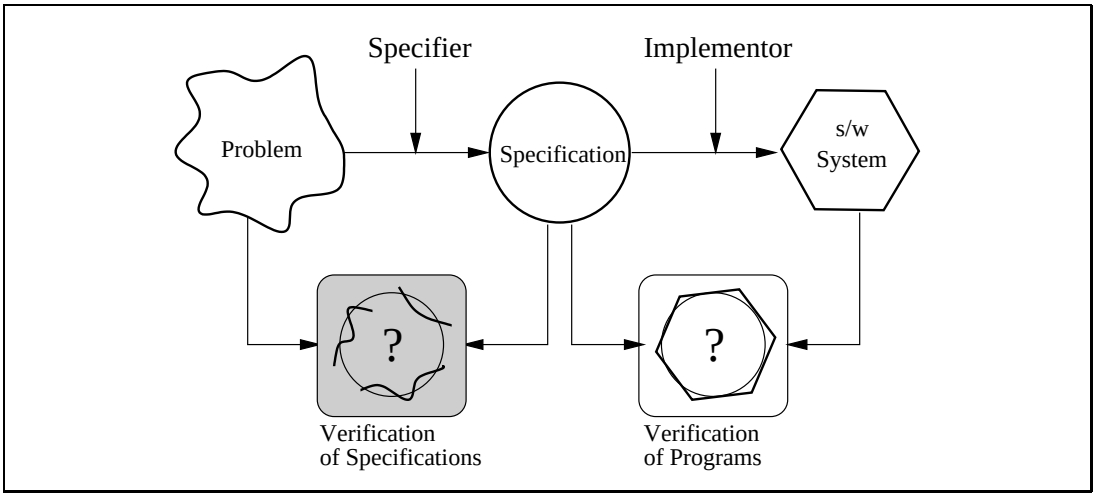

Figure 1.1: Software Development Model

At this point it should also be clear that formal verification is different from testing. Formal verification establishes the validity of a property in a given specification in an absolute manner. Testing, in all but the simplest specifications, can only establish the validity of a property in a subset of the specification. While an appropriate choice of test cases can give a high degree of confidence, that confidence is never absolute.

Figure 1.1 shows a schematic representation of the software development process described above. This is not meant to be a comprehensive model of the software development life cycle, simply an aid to identify the main concepts that are of interest in the present context. The step from problem to specification, in particular, could be further divided into at least two sub-steps: requirements gathering, and specification derivation.

This thesis deals primarily with the verification of specifications — Figure 1.1 shows the corresponding component of the development model in gray. Nevertheless, since usability relates to the interaction between system and users, and users interact with the actual system that has been built, some issues regarding the relationship between models and implementations of interactive system will also be addressed. In any case, references to verification, or formal verification, will mean primarily formal verification of specifications.

1.3 Interactive systems and formal verification

The point about proving theorems during development of a system, is to improve the development process so that better systems can be obtained. Talking of better systems and of reasoning about systems quality implies, as seen above, some measure of that quality. In other areas of computer science, such as hardware design or computer communications, measures of quality are relatively easy to establish. Two examples of "hard" requirements which can usually be verified of a system or design are: deadlines must be kept, and there can be no information loss. In fact, formal methods and verification techniques have been successfully applied in the development of such systems (Clarke et al. 1996). Interactive systems, however, are different from such systems in that the area of analysis is not confined to the system being designed, but must also consider the user. This means that usability criteria are seldom amenable to being determined by some calculation.

A field where verification techniques have been used with some success is the verification of reactive systems (Manna & Pnueli 1995). In order to best highlight the specific problems that interactive systems pose, a comparison between the two types of system will be made.

1.3.1 Reactive systems

A reactive system can be thought of as a system that is immersed in, and communicates with, some environment (see Figure 1.2). Hence, its behaviour is non deterministic, as it is conditioned by the environment. In the present context, this system will be some software system and the environment will be composed of other systems which might themselves be software systems or real world artifacts.

An example of such a system is an `http` (Hyper Text Transfer Protocol[3]) server in the internet accepting requests from other servers or from browsers. In this case the server is the reactive system and the environment is composed of all the other servers and browsers (the clients of the reactive system).

[3] `http://www.w3.org/Protocols/` (last accessed on the 20th of September, 1999).

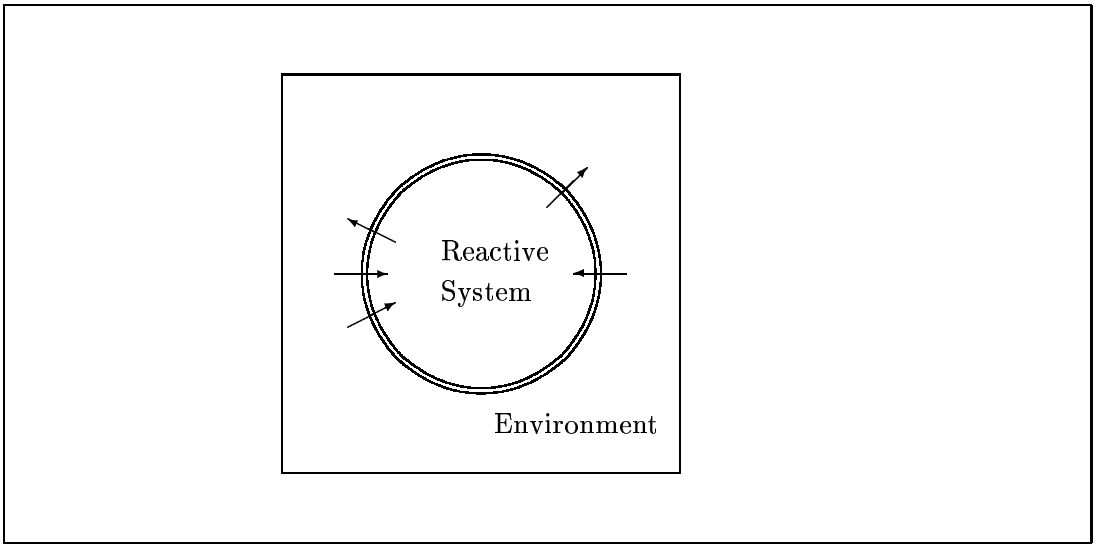

Figure 1.2: Reactive System

Such systems have a communication protocol (in this case `http`) that determines how the functionality of the system can be accessed. The communication protocol will define how information can be transfered (typically by means of discrete events and/or status phenomena) and possibly some rules that specify the ways in which those events or status phenomena can be combined (in a linguistic model of interaction these would be called the lexical and syntactical levels of the dialogue). It is assumed that the clients of the reactive system know the communication protocol, and are capable of using it. Typically the system will not respond, or respond with an error message, to incorrect requests. It is also assumed that it is up to the clients to be able to interpret the reply of the reactive system. The key point to retain is that reactive systems are built to provide a service through a protocol and the concerns of its designers end in that protocol. Typically, the reactive systems designer's knowledge about the clients is only that they will be designed to communicate using the appropriate protocol.

1.3.2 Interactive systems

Interactive systems can be seen as special cases of reactive systems. They too are immersed in some environment and have to respond to requests from that environment. Considering interactive systems as a special case of reactive systems, it should be possible to use the same tools and techniques with similar results. A significant

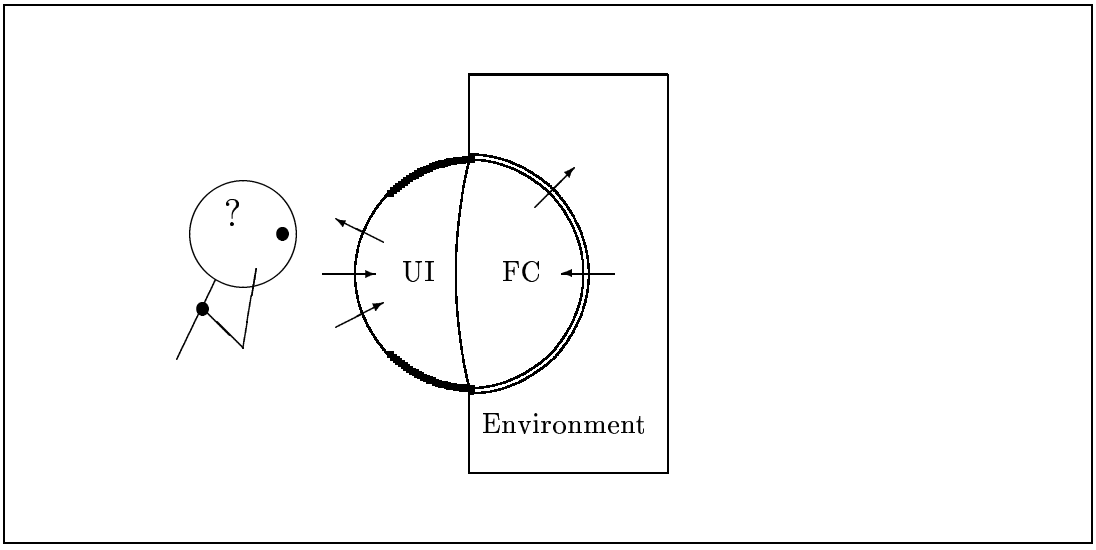

Figure 1.3: Interactive System

difference exists, however: part of the environment of interactive systems is composed of human users (see Figure 1.3).

Consider a web browser. It can also be thought of as a reactive system. It receives, and handles, requests from the user to get pages and requests from the http server to show pages. The difference between this system and the http server is manifested at two levels:

- the complexity of the interaction that it is maintained with the user;
- assumptions that must be made about users' capabilities.

Regarding the former, user interface technology keeps evolving, and the means by which user and system can interact are far more complex that what is common between any two computer systems. Hence, the specification of the dialogue between user and system will be potentially more complex. On the other hand, humans are intrinsically heterogeneous and their cognitive capabilities are limited. The communication protocol has to be developed with that in mind also. In particular, it cannot be simply assumed that the user will be able to interpret the interface. In fact, the aim of usability reasoning is to analyse how well users will be able to interact with the system. So, while developing an interactive system, knowledge about the clients of the system (the users) is used to inform and shape the design. It could be argued that this is also the

case in traditional reactive systems. Humans, however, are far more complex and more unpredictable than software, and considerations about them have a greater impact on design.

This situation is also reflected in the verification process. While for traditional reactive systems, analysing the correctness of the system encompasses an objective analysis of the behaviour of the system, when considering interactive systems there is also concern with a more subjective analysis of the effectiveness of the communication protocol. For example, when developing a reactive system it might be possible to rely on the underlying network to ensure that a message that is sent is correctly received. When developing an interactive system, however, there will be a need to analyse how the message is presented to the user in order to assess if it will be correctly perceived. Basically, there is a need to assess how effectively the user will be able to use the interface. The problem here, is that a detailed specification of how the user *works* is not available.

From the above it should now be clear that reasoning about usability involves more than what is traditional for other types of systems. Aspects related to the user must also be considered. In fact, some authors argue that, in the design of interactive systems, the word system should be applied to the pair user/interactive artifact (Butterworth et al. 1998*a*, Duke et al. 1998). Or, in cases like Computer Supported Cooperative Work (CSCW), to collections of users and artifacts.

1.4 The need for an integrated approach

It is the need to take the user into consideration which makes HCI a multidisciplinary area. (Preece et al. 1994) identifies eleven disciplines which contribute to HCI, ranging from Computer Science, to Social Sciences such as Philosophy or Anthropology. It is clear that designing an interactive system poses a set of questions which cannot be solved by software engineering alone. It is not reasonable to expect software engineers to have expertise in all of those areas. Design must become a cooperative process between experts from the several disciplines involved. When it comes to the use of formal methods to perform analysis, although increased application of this type of

method can be witnessed in specific areas, such as hardware or protocol design, its use if far from widespread. Hence, there might be the need to resort to a formal methods expert. From the above, three basic roles can be identified regarding the formal analysis of a design: the designer, the (formal) analyst, the human-factors expert[4]. The designer is the person/team carrying out the design of the system. The (formal) analyst is the person/team which will be performing formal analysis of the design. The human-factors expert is the person/team with the expertise to decide what factors will influence usability. The problem now is how best to integrate these different contributions into a coherent design process.

Bellotti et al. (1996) report on a case-study where designers presented a number of teams of formal methods and human-factors experts, with a design and some requests for analysis. As a result of the several analyses carried out, a list of problems with the proposed design was produced, together with some tentative design alternatives for illustration purposes. While this approach has enabled the identification of a number of problems with the proposed design, a closer integration between the different participants would be beneficial. As it stands, the process requires some sort of *feature freeze* which is then submitted for analysis. When the analysis results are received, they must be integrated into the design. This makes analysis almost an outside process regarding design, and has the potential to create problems in communication between analysts and designers. Furthermore, the analysis initiative rests on the designers, while the analysis expertise rests on the analysts. A closer integration of the different roles should allow for a quicker turnaround time, and for the expertise of each of the participants to be better explored.

Formal methods is part of software engineering, hence ideally the software engineers designing the system should acquire formal methods expertise. This would eliminate the need for an external formal methods analyst, since the designer would be able to perform the analysis. The interaction between software engineering and human-factors, on the other hand, must be accomplished at the level of the process. A way is needed to integrate human-factors concerns into the software engineering process of

[4]From now on *human-factors* will be used as an umbrella term for all disciplines that pertain primarily to the user.

designing the system. In particular, human-factors concerns should guide the analysis towards those issues that are user relevant

1.5 Automated techniques for formal verification

So far the chapter has highlighted the motivations behind research on the use of formal verification for usability reasoning, and the main problems it faces. The use of automated reasoning techniques to aid in the verification process has also been mentioned. This section introduces the two main classes of automated techniques: theorem proving and model checking.

The process of reasoning about formal models of systems has long been an object of study (Jones 1992). For a long time, however, formal proofs tended to stay in the area of envisaged benefits that were never actually completely fulfilled. It would be said that formal proofs could be done, but little was shown about how to actually prove interesting properties of a system. The fact is that formal proof tends to be a delicate, detailed, and time consuming process. As the complexity of models grows, tackling such proofs by hand becomes increasingly harder. This has led to the study of mechanical reasoning techniques as a way to (at least partially) automate the analysis. Two main categories of methods can be identified:

- deductive methods (i.e. theorem proving): these are semi-automated methods where a traditional mathematical proof is performed by a tool under user guidance — examples of theorem provers are PVS (Crow et al. 1995) and the Larch system (Guttag et al. 1993);
- algorithmic methods (i.e. model checking – see Clarke et al. 1986): these are fully automated methods and, given suitable system descriptions and properties, are capable of determining if a property is valid in a system, without human intervention.

1.5.1 Theorem Proving

Theorem provers take a deductive approach to verification. Proofs are performed in the traditional mathematical style, using some formal deductive system (a set of log-

ical axioms, together with a set of deduction rules). Proofs progress by transforming (rewriting) a set of premises into a desired conclusion, using the axioms and the deduction rules. As a small example consider a logical language with sort S, and operators $a, b, c : S$ and $\bowtie$: $S \times S \rightarrow$ Bool. Consider also a deductive system with:

Axioms	Deduction Rules
(1) $\forall_{x,y,z:S} \cdot x \bowtie y \wedge y \bowtie z \rightarrow x \bowtie z$	$\dfrac{P, P \rightarrow Q}{Q}$
(2) $b \bowtie c$	

The proof that if $a \bowtie b$ then $a \bowtie c$, can be carried out in the following way:

1.	$a \bowtie b$	premise
2.	$a \bowtie b, b \bowtie c$	Axiom 2
3.	$a \bowtie b, b \bowtie c, a \bowtie b \wedge b \bowtie c \rightarrow a \bowtie c$	Axiom 1 with $x = a, y = b, z = c$
4.	$a \bowtie c$	deduction rule and 3.
5.	*QED*	

In order for the proof to be considered valid, each proof step (i.e. each transformation of the premises) must be formally justified. This means that proofs can quickly become unwieldy. Theorem provers try to solve this by automating as much as possible of the different types of proof steps. In the above proof, for example, the transition from 3 to 4 could be done automatically by a theorem prover. The process is not fully automated because user intervention is still needed in order to provide guidance regarding the strategy to follow. Hence, in Step 3 above, the user would ask the prover to try applying the appropriate deduction rule.

The philosophy of different systems, regarding user intervention, varies. Some adopt a CISC type of approach and try to prove the formula using a set of powerful tactics, trying to minimize the intervention of the user. Others, using a RISC approach, apply little more than the re-writing rules, and if unable to prove the formula, ask the user for a proof method to apply. Usually this latter kind of system is programmable so that it can be made to emulate a CISC-like system.

Theorem provers provide a degree of automation. Nevertheless, it is still up to the verifier to decide the proof strategy but for the simplest proofs. Additionally,

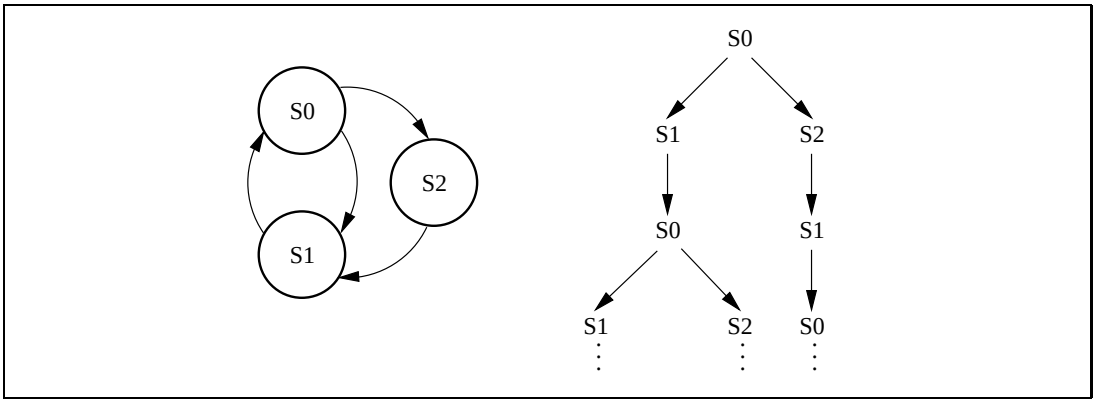

Figure 1.4: Execution paths of a finite states machine (adapted from Clarke et al. 1986)

automated proofs need to be much more detailed than normal hand produced proofs, since no leaps in reasoning can be made. The prover must be "convinced" of the correctness of every step in the proof. So, automated theorem proving can be difficult to use. Learning curves, in particular, can be quite steep. Besides this, traditional theorem provers are based on some sort of equational logic, which means that they are suited to reason about equality, but not to reason about change (see Chapter 5). This might become a problem since in many situations the properties of interest relate to the behaviour of the system. Given these limitations, research in the area is continuing and alternative approaches have been sought.

1.5.2 Model Checking

Model checking was originally proposed as an alternative to the use of theorem provers in concurrent program verification in (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 1.4). In this context, the possible execution paths are interpreted as alternative futures.

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

- the analysis is fully automated (as opposed to theorem provers' high reliance on

the skills of its users);

- the validity of a property is always decidable (as opposed to theorem provers' undecidability problems).

A main drawback of model checking has to do with the size of the finite state machine needed to specify a given system: useful specifications may generate state spaces so large that it becomes impractical to analyse the entire state space. Hence, theoretically decidable systems may become undecidable in practice. The use of Symbolic Model Checking (Burch et al. 1990) somewhat diminishes this problem. Avoiding the explicit representation of states, state spaces as big as 10^{20} states may be analysed. The problem remains, however, that some systems might not be specifiable by a finite state machine. Work in this area has led to some techniques that can be applied to check some properties even when the state machine is infinite. For a discussion of the applicability of these techniques in the context of interactive systems see (Mezzanotte & Paternò 1996).

The kind of analysis that model checking is concerned with has basically to do with testing the *universality*, *inevitability* or *possibility* of given properties expressed in some temporal logic. For example, whether it can be guaranteed that the system will not be in a deadlock situation, or that users will have to save their work before quitting.

1.5.3 Other approaches

Model checkers and theorem provers have complementary capabilities. An area of research has been concerned with the combination of the benefits from both techniques. This has lead to the development of a number of tools combining model checking and theorem proving, for example, TLP (Engberg 1994, Engberg 1995), PVS (Rajan et al. 1995, Owre et al. 1997), or STEP (Bjørner et al. 1996, Manna & Pnueli 1995). These tools are further discussed in Chapter 5.

1.6 Thesis Overview

In this chapter the main concepts that play a role in this thesis have been identified. Section 1.1 has motivated the need for usability reasoning from early stages of design. As systems grow in complexity, formal (mathematical) models can help in controlling the problems raised by complexity. Section 1.2 has made it clear what is meant by formal verification in the context of the thesis. Section 1.3 has then highlighted what makes the verification of interactive systems a challenging problem, and Section 1.4 has explained the need to integrate human-factors concerns into design and verification. Finally, Section 1.5 has described the two main classes of techniques for reasoning about formal models in an automated way: theorem proving, and model checking.

The object of this thesis can now be more precisely stated as the use of automated deduction techniques in the formal verification of specifications in the context of the development of interactive system.

At this stage it should be noted that automated deduction will not be proposed as a complete alternative to other approaches. As it will become clear during the thesis, automated reasoning can be one more tool which can be used, in many cases with advantage, alongside the traditional approaches to usability reasoning.

The thesis will now progress as follows:

- Chapter 2 will review the current state of affairs concerning the use of automated reasoning in interactive systems development. This will set the foundations for the remaining work.

- Chapter 3, based on the results of the review, will discuss the role which automated deduction might play during interactive systems development. A new approach to the integration of automated deduction into development is proposed, as well as a framework for reasoning about interactive systems properties.

- Chapter 4 addresses the use of model checking in the context of the approach proposed in Chapter 3. A tool to enable the analysis of MAL based interactor models in SMV is introduced.

- Chapter 5 addresses the use of theorem proving. The use of the framework for reasoning about interactive system properties is also addressed in the chapter.
- Chapter 6 describes a less elementary case study, where the use of both model checking and theorem proving is brought together.
- Lastly, Chapter 7 reviews and enumerates the results achieved during the thesis, compares the proposed approach to other work in the area, and presents directions for further work.

Chapter 2

Literature Review

The previous chapter has motivated the use of automated reasoning techniques during interactive systems development. The chapter has shown how interactive systems pose a set of specific problems which need to be addressed if the application of formal verification to their development is to be successful, and has also described the two main classes of automated verification techniques available. This chapter will describe the current attempts at applying available verification tools to the field.

A shorter version of the material presented in this chapter has previously appeared as (Campos & Harrison 1997).

2.1 Introduction

This chapter covers the main work to date on the application of automated reasoning techniques to the analysis of interactive systems specifications. Automated reasoning assumes the use of some formal notation to write specifications, and some tool to perform the analysis. Thus, approaches which, although using formality, do not consider the use of automated reasoning tools, fall outside the scope of the thesis. Having a formal approach does not necessarily translate into having a tool to reason about the specifications. Nevertheless, relevant work will be referenced where appropriate.

Since interactive systems can be seen as special cases of reactive systems (see Chapter 1), it is natural that the first approaches to the use of automated reasoning tools

in the analysis of interactive systems designs were done using model checking. Model checking is a technology that was developed for the analysis of reactive systems. Hence, proceeding chronologically, the chapter starts with the approaches that use model checking. Since each approach focus on the use of a particular tool, they will be identified by the tool used. Three main approaches are described:

- Abowd et al. (1995) use SMV (see Section 2.2.1);
- Paternò (1995) uses the Lite tool set (see Section 2.2.2);
- d'Ausbourg et al. (1996) use Lesar (see Section 2.2.3).

Regarding theorem proving, one approach is described:

- Bumbulis et al. (1996) use HOL (see Section 2.3.1).

Some recent work is mentioned in Section 2.4. Finally, some conclusions are drawn from the exercise, and some work that might be used in overcoming pitfalls that are identified is referenced (see Section 2.6).

2.2 Model Checking

As said, the first approaches to formal verification of interactive systems were based on model checking technology.

The three above mentioned approaches will now be described. Abowd et al. (1995) specify the user interface in a Propositional Production Systems style using the Action Simulator tool, and then use SMV to analyse the specification. Paternò (1995) uses LOTOS Interactors to make a hierarchical specification of the user interface, based on the task analysis output, and then uses the Lite tool set to analyse that specification. The last group of authors does not use a traditional model checker. Instead, they use Lustre, a data flow language, and the associated verification tool Lesar. In order to describe, and afterwards compare, the three approaches, three main aspects will be considered:

- how the user interface is specified,

- how that specification translates into some kind of finite state machine,
- how the resulting finite state machine description can be analysed.

2.2.1 Using SMV

Abowd et al. (1995) (see also Wang & Abowd 1994) combine the simplicity of Action Simulator (Monk & Curry 1994) with the power of the Symbolic Model Verifier (SMV) tool (McMillan 1993) to perform the analysis of interactive systems specifications. The user interface is specified using Action Simulator and then translated into the SMV input language. The specification is then analysed in SMV using Computational Tree Logic (CTL) formulae (Clarke et al. 1986).

The Dialogue Specification

Action Simulator (AS) is a spreadsheet package that allows for the specification of dialogues in a tabular fashion, using a Propositional Production System (PPS) style of approach. Additionally, the tool has dialogue simulation capabilities: the specification can be executed allowing the designer to observe its behaviour.

To specify an interactive system with a PPS, first a set of fields defining the system state must be identified. Each field represents some information on the system state and at each state a field can only have one value. Afterwards, the set of available actions to manipulate that state (the events) is also identified.

Figure 2.1 shows the specification of a very simple photocopier: rows correspond to events and columns to fields, the first row is the state of the system. The dialogue is specified by associating with each event pre- and post-condition pairs. The pre-condition of an event defines the combination of field values that makes the event enabled (events marked with "`********`" in the example are enabled). The post-condition of an event defines which will be the values of the fields after the event takes place. Pre-conditions are written on the top side of the rows, and post-conditions on the bottom side. Blank fields in pre-/post-conditions mean, respectively, don't care/don't change values. So, in the example, event `Req.>1copies` is enabled, as its pre-condition – field `Copying` set to `FALSE` and field `One Copy` set to `TRUE` – is verified

	Conditions	No. conds = 3	
	Copying	One Copy	Normal Toner
State	FALSE	TRUE	TRUE
Req. >1 copies ********	FALSE	TRUE FALSE	
Reset Copies	FALSE	FALSE TRUE	
Darker ********			TRUE FALSE
Cancel darker			FALSE TRUE
Copy ********	FALSE TRUE		
finished copy	TRUE FALSE		

Figure 2.1: PPS specification using Action Simulator (adapted from Monk & Curry 1994)

by the present state. If the user chooses to generate this event, the field `One Copy` will be set to `FALSE` and the other fields will be left unchanged (see post-condition of `Req.>1copies`).

Note that as there can be more than one pre-/post-condition pair associated with each event, the specification can be nondeterministic. This can be useful to abstract away parts of the control of the system being specified. Note also that there is no mechanism to prevent deadlock situations.

A PPS specification can be seen as a tuple PPS $= (A, \Sigma, T, P)$ where:

- A is a finite set of event labels;
- Σ is a finite set of dialogue states;
- T is a binary relation where $T \subseteq A \times (\Sigma \rightarrow \Sigma)$, in which each member specifies a rule;
- $P : \Sigma \rightarrow \mathcal{V}^F$ assigns to each state a partial function mapping fields (F) to their values in that state ($\mathcal{V}$ is the set of all possible field values)

It is clear from this definition that a PPS defines a labelled finite state machine. Σ and P define the states, while T associates transitions between states to the events in A.

The Finite State Machine

In order to be analysed, the PPS specification must first be translated into the SMV input language. This input language describes the transition relation of a finite state machine, which can then be analysed in SMV using CTL formulae. In the context of SMV, the finite state machine is called a CTL machine. A CTL machine can be described by the triple CTL $= (S, R, P)$ where:

- S is a finite set of states;
- $R \subseteq S \times S$ gives the possible transitions between states, and must be a total relation;
- $P : S \rightarrow 2^{AP}$ assigns to each state the set of atomic propositions (AP) true in that state.

Hence, the CTL machine is also a finite state machine. Comparing the definitions of PPS and CTL, however, it can be seen that there is a mismatch. In the PPS transitions are labelled; while in the CTL machine they are not. The labels in the PPS are important, however, as they describe which event caused which transition. A way of performing the translation between both tools is needed that preserves this information.

This problem was overcome by including the events as state information. More specifically, each CTL state represents the state of the system, as it was described in the PPS, and also a possible next event in the dialogue. Hence, each state in the PPS specification is represented by n states in the CTL machine, one for each of the n possible events in the original PPS state. This situation is exemplified in Figure 2.2. This means that the notion of state in the PPS is different from the same notion in CTL, so care must be taken when talking about PPS states during verification in SMV.

Additionally, as R must be total, dialogues with deadlocks cannot be represented in CTL. This problem is solved by including in the PPS specification a special event *Stuck* that will be enabled when no other event is, and that will be associated with the identity transition.

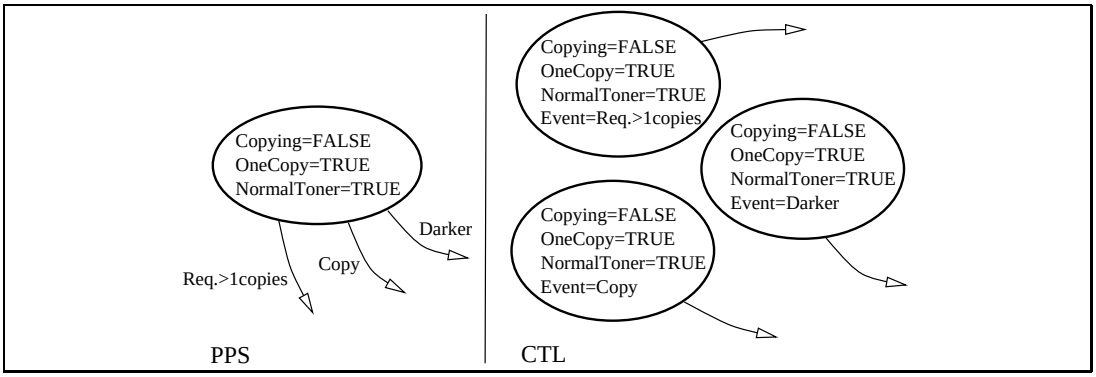

Figure 2.2: From one PPS state to 3 CTL states

In (Wang & Abowd 1994) an algorithm for the automatic translation from PPS to CTL is presented.

Checking the Specification

Once the PPS specification is translated into a CTL machine, the SMV tool can be used to check the validity of CTL formulae in the machine. As has been seen, the CTL machine is a finite state machine and CTL formulae allow questions to be asked over the execution paths of that machine. For a complete description of the CTL syntax see (Clarke et al. 1986). Besides the usual propositional logic connectives, CTL allows for operators over paths:

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

and over states in a path:

- G – for all states in the path (universal quantifier over states in a path);
- F – for some state in the path (existential quantifier over states in a path);
- X – for the next state in the path;
- U – some property will hold in the path until some other property holds.

Propositional predicates over Σ (the states) are called state formulas. The operators over states can be applied to state formulas to obtain path formulas. Applying an operator over paths to a path formula yields a state formula. So, if p is a state formula, some of the valid combinations are:

- $AG(p)$ – p is universal (for all paths, in all states, p holds);
- $AF(p)$ – p is inevitable (for all paths, for some state along the path, p holds);
- $EF(p)$ – p is possible (for some path, for some state along that path, p holds);
- $AX(p)$ – p holds in the next state;
- $EX(p)$ – p might hold in the next state;
- $A[pUq]$ – p holds until some other property q holds.

All these formulas are related to the initial state of the CTL machine.

As the CTL specification is state based, dialogue properties will be expressed in terms of the atomic propositions that describe states. It must be noted that any given combination of properties does not necessarily identify, uniquely, one and only one state, but a set of states that satisfy these properties.

In (Abowd et al. 1995) a set of templates for testing relevant properties of interactive systems is proposed. They can be divided into three categories: state, event, and task (where a task is some target state to be reached). The state category includes the templates (with `s_stmt` a predicate on states, and `ev_stmt` a predicate on events):

- deadlock freedom – `!EF(event=Stuck)` – it is impossible to reach a state where the next event is `Stuck`;
- state inevitability – `AG(s_stmt1 -> AF(s_stmt2))` – from the states defined by `s_stmt1` the user always has to go through some state in `s_stmt2`;
- state floatability – `AG(s_stmt1 & !s_stmt2 & !ev_stmt1 & ... -> E[!(s_stmt2) U s_stmt3])` – from a state in `s_stmt1` the user can go to a state in `s_stmt3` without passing through `s_stmt2` (the negations in the definitions of the initial state ensure that the user is not already in the undesired state and that the next action will not lead to that state);

the event category includes the templates:

- rule set connectedness – `EF(Event=Action & preAction)` – action `Action` can be executed;
- reversibility – `AG(Event=Action & s_stmt -> EX EF(s_stmt))` – the effect of an action can be undone.

The task category includes the templates:

- weak task completeness – `EF(s_stmt)` – it is possible to reach a state that is identified with the completion of a task;
- strong task completeness – `AF(s_stmt)` – it is inevitable that a state will be reached which is identified with the completion of a task;
- weak task connectedness – `AG(s_stmt1 -> EF(s_stmt2))` – from a given state set, whatever is the next action, the user can always reach another designated state set;
- strong task connectedness – `AG(s_stmt1 -> EF(ev_stmt & AX(s_stmt2)))` – from a given state set, whatever is the next action, the user can always reach another designated state set using a designated last action.

Some additional properties are presented in (Wang & Abowd 1994). Overall, the properties that can be checked are about what states can or cannot be reached from some other states, which is not surprising since that is what CTL allows for.

Discussion

The power of a verification approach is a function of both the expressive power of the specification language, and the analytic power of the verification technique used. Regarding how Abowd et al. (1995) specify interaction, it can be seen that all input is represented by events, while all output is represented by state fields. Thus, it is only possible to think of interaction using actions for input and status phenomena for output. Although it is possible to devise ways of representing status phenomena using

events and vice-versa, this is clearly awkward. It has been argued by Dix & Abowd (1996) that both interaction techniques should be equally addressed in specification methods. Further, there is no obvious way of distinguishing user events from functional core (or system) events. In the example a convention was adopted of marking system events with asterisks, but that is only a convention. Similarly it is not possible to distinguish events that only affect the interface from events that affect the functional core. This arises from the fact that there is no distinction made between user interface state and underlying system state. The system is looked at as a whole and the fields that define the state information are supposed to represent the interface of the system as well as its underlying state. Although this can be so for small simple systems, in complex systems it will not be feasible (or even desirable) to show every thing at every time.

Additionally, there is almost no provision for user considerations in the specification. Only tasks are considered, and even so, only as a target state to be reached. This is a very simplistic view of a task and, again, can only be truly useful in very simple cases.

At the technological level, besides the already stated over-simplicity of the specification, some problems may occur due to the inclusion of events in the states of the CTL machine. The first and more obvious one is a problem of state explosion. Figure 2.2 was an example of this.

A more subtle problem might happen when questions like,

```
EX(Copying)
```

are asked. In this case what is being checked is if, from the initial states of the system, it is possible to start copying with only one action (there exists a next state where copying is true). Looking at the specification in Figure 2.1 (page 44), this seems to be true. All that has to be done is to select event `Copy`, which is enabled. If the formula is presented to the model checker, however, the answer is:

```
-- specification EX(Copying) is false
```

Something seems wrong. Since the failure of the existential quantifier provides no counter example, a double check is tried by asking:

```
AX(!Copying)
```

which is the opposite property (it is not possible to start copying with one action). The answer is also negative:

```
-- specification AX(!Copying) is false
```

and a counter example shows that using the event `Copy` it is possible to have a next state with `Copying` set to true.

Thus the model checker seems to be saying that from the initial states it is not possible to set `Copying` to true with one action. At the same time it is saying that it is possible to do it using the event `Copy`. Apparently there is a contradiction.

Looking back at Figure 2.2 in page 46 it is possible to see what is happening. While in the PPS there is only one initial state, in the CTL machine there are three initial states. One for each possible next event. In this context, the first property is interpreted as *for all* initial states `EX(Copying)` is true, and clearly this property is only true for the state where the next event is `Copy`. So the property is indeed false. On the other hand, the second property is also false because, in fact, there is a state in the set of initial states which contradicts it.

So, the problem arises from the different notions of state that are used in the PPS and in SMV. What is more, the original property cannot be checked. This is due to the fact that, at the SMV level, there is no way of identifying states without their associated events. What can be asked is: "from the initial states of the system, *and whatever action is taken*, is it possible to start copying with only one action?". This is considerably stronger than the intended formulation: "from the initial states of the system, is it possible to start copying with only one action?".

This example shows that model checking, although providing answers in an automatic way, does not necessarily give a profound insight into the PPS specification. Were it not a so simple property the initial answer might have been accepted as a

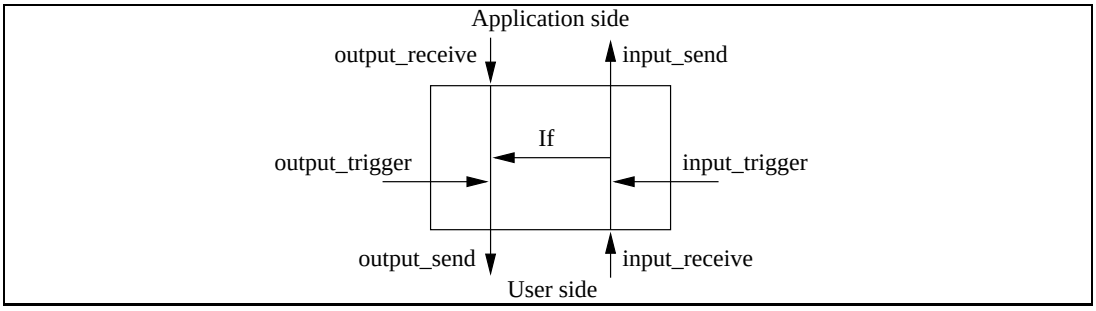

Figure 2.3: LOTOS Interactor Architecture (adapted from Paternò 1995)

correct answer to the question that supposedly was being asked. So, although the verification step is automatic, human intervention is needed to interpret an analyse the results. This example also shows that the formulation of the predicates expressing the desired properties is not a trivial matter. In this case the problems arose from difficulties with the translation between tools. Complex specifications, however, are bound to introduce problems of their own making.

2.2.2 Using the Lite tool set

The main problem with the previous approach has to do with the expressive power of the specifications used, which is very limited. This section shows how a more powerful specification notion has been used.

In (Paternò 1995) an approach based around the Lite tool environment (Mañas et al. 1992) is proposed. Interactor based specifications written in LOTOS (Bolognesi & Brinksma 1987)[1] are translated into a finite state machine, and then analysed using Logic Checker, a model checker that checks an action based version of CTL (ACTL — Nicola & Vaandrager 1990, Nicola et al. 1993).

The Dialogue Specification

In this approach the specification of the user interface is based on the interactor architecture presented in Figure 2.3.

The notion of interactor is defined as a pair $I = (FI, FO)$:

[1] For an easy and practical introduction to LOTOS see (Drayton et al. 1992).

$$FI : (input\_receive \times input\_trigger \times T) \rightarrow (input\_send \cup \Phi) \times If$$
$$FO : (output\_receive \times If \times output\_trigger \times T) \rightarrow (output\_send \cup \Phi)$$

where T stands for time and Φ for null information.

Basically, an interactor is a black box that conveys information from the user side to the application side through channels *input_receive* and *input_send* (the internal dialogue – *FI*), and from the application side to the user side through channels *output_receive* and *output_send* (the external dialogue – *FO*). Note that *input_receive*, *input_send*, *output_receive*, and *output_send* represent sets of channels. The triggers are boolean gates that determine when information is conveyed. To allow for feedback, there is also a flow of information from the internal dialogue to the external dialogue (*If*). This flow, however, is internal and not directly perceivable from the outside. Interactors can be composed to construct a hierarchy, allowing for a modular specification of the interface.

The specification of the user interface is obtained by transformation of a task description of the system into a hierarchy of interactors. For a full description of the process see (Paternò 1995). After defining the architecture of the user interface using interactors, each interactor is specified in LOTOS. This specification is done writing a LOTOS process for each interactor. This process will specify the behaviour of the input and output channels of the interactor.

As an example, a very summary specification of the photocopier in the previous section (see Figure 2.1 in page 44) could be done using a single interactor. Figure 2.4 shows a graphical representation of this interactor. In the figure it can be seen that there is a separation between the interface and the functional core (or application). Further, because the channels can transmit parameters, this interactor specifies the interface in more detail than the corresponding PPS. Event `Req.>1copies`, for instance, has been replaced by event `in_req_n_copies` which allows the discrimination of the desired number of copies. This is made clear when the interactor is fully specified in LOTOS. The full specification of the interactor is too big to be presented here, so just an abstract is presented in Figure 2.5.

As the example shows, the behaviour of the interactor is described by writing expressions that define the effect of each of the inputs. This example is simplistic, but

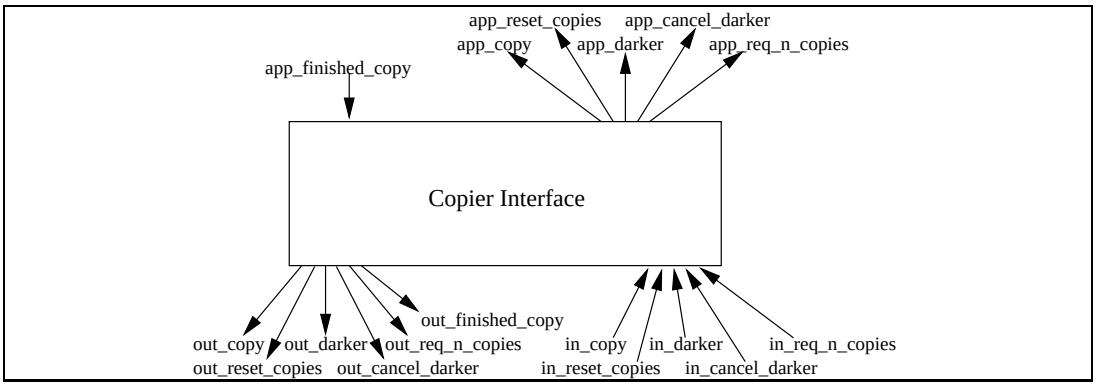

Figure 2.4: LOTOS Interactor for the copier interface

```
process Copier_Interface [in_copy, ..., in_req_n_copies,
                          app_copy, ..., app_req_n_copies,
                          app_finished_copy,
                          out_copy, ..., out_req_n_copies, out_finished_copy
                         ] (copying, one_copy, normal_toner: Bool): noexit :=
[copying = false] -> in_copy; app_copy;
                     out_copy;
                     Copier_Interface(true, one_copy, normal_toner)
[]
...
[]
[copying = false ∧ one_copy = true] ->
                     in_req_n_copies ?n: nat;
                     app_req_n_copies !n;
                     out_req_n_copies !n;
                     Copier_Interface(copying, false, normal_toner)
[]
[copying = true] -> app_finished_copy;
                     out_finished_copy;
                     Copier_Interface(false, one_copy, normal_toner)
```

Figure 2.5: The LOTOS specification of the copier interface

clearly this specification notation has more expressive power than the PPS above. It allows for a modular style of specification and supports the distinction between user, user interface, and application. An example of a complex system analysed using this approach is MATIS, the Multi-model Air Travel Information System (Duke et al. 1995).

The Finite State Machine

So that it can be analysed, the LOTOS specification must be translated into a Finite State Machine (FSM). This translation is done automatically by the Auto tool (Mañas et al. 1992), but with some limitations.

In order for the translation to be possible, the LOTOS specification must first be translated from Full LOTOS to Basic LOTOS. This translation process implies that data type information and parameters will be lost, as well as the boolean guards used to constrain behaviour. In the example above, this means two things. First, the parameters in the events relative to requesting more than one copy will be lost. Second, and more importantly, the guards that specify the preconditions for the events are also lost. This means that in the Basic LOTOS version of the specification all events are always enabled regardless of the interactor state.

The loss of data type information does not seem to be such a serious problem. Different gates can be automatically created for each data type thus preserving necessary distinctions, and the approach is not concerned with the system response to particular data values, but with overall behaviour. It does mean, however, that there will be an explosion on the number of gates. Additionally, the introduction of *artificial* gates might make verification results harder to interpret.

The loss of conditional guards, on the other hand, will cause the Basic LOTOS version of the specification to admit more traces of behaviour than the Full LOTOS version. This means the FSM will have traces of behaviour that are not present in the initial specification, so at least the reachability properties of the specification will be affected. In (Paternò 1995) it is suggested that boolean guards should be avoided and that process synchronisation should be used whenever possible. This, however, means that the natural way to describe the system is no longer possible. In (Palanque

et al. 1996) it is shown that a manual translation from Full to Basic LOTOS might further obviate this problem, but at the cost of losing the automation.

Checking the Specification

Once the specification has been translated into a FSM, the Logic Checker tool is used to analyse it. Logic Checker checks ACTL formulae. ACTL is a branching time temporal logic that allows reasoning about the actions that a system can take. ACTL formulae are interpreted over a Labelled Transition System defined as a tuple $(Q, Ac \cup \{\tau\}, \rightarrow, Q_0)$, where:

- Q is a set of states;
- Ac is a finite, non-empty set of visible actions;
- τ represents the internal, not visible, actions;
- $\rightarrow \subseteq Q \times (Ac \cup \{\tau\}) \times Q$ is the transition relation;
- Q_0 is the initial state.

Looking at the definition of the $\rightarrow$ relation it can be seen that the transitions are labeled by the actions. This is different from CTL where there was no notion of labels (see Section 2.2.1, page 45). Nevertheless, it has been proved that they have the same expressive power (Nicola & Vaandrager 1990). In fact, ACTL is an offspring of CTL developed specifically with the intent of allowing reasoning about actions.

Note that Ac represents not only the user actions, but also the responses from the system, and events at the application level. So, at this level, the distinction between user actions and application actions is not so clear. Note also that in ACTL states do not have an associated value.

The syntax of ACTL is somewhat clumsier than that of CTL. Now, besides state and path formulas, it is also possible to have action formulas (predicates over Ac). As has been said, there is no notion of state values so state formulas can only be obtained from path formulas, not from states themselves (as they are *empty*). Following the

syntax used in (Paternò 1995) the same operators as in CTL are used but with some differences.

The quantifiers over paths (A and E) and over states in a path (F and G) work in the same way. The X and U operators over states in a path are now indexed by action formulas:

- $X\{ac\}st$ – the next action in the path will satisfy ac and the state resulting from that action will satisfy st;
- $st_1\{ac_1\}U\{ac_2\}st_2$ – the path has states satisfying st_1 and actions satisfying ac_1 until an action satisfying ac_2 leads to a state satisfying st_2.

Finally, state formulas can also be built the following way:

- $<ac>\ st$ – from the initial state there is one computation in which no action is observed (only τ happens) until an action satisfying ac is observed and a state satisfying st reached;
- $[ac]st$ – from the initial state if an action satisfying ac occurs then a state satisfying st is reached (note that before ac internal actions might happen).

Note that in ACTL the emphasis is on the actions observed more than on the states reached. For instance, in ACTL one does not ask "*is it possible, in the future, to have the copier in single copy mode?*" but "*is it possible, in the future, to perform the* Reset Copies *action?*" In fact, since there is no notion of state fields, most of the state formulas are just the constant `true`.

A number of property templates are proposed for checking the specification. These are divided into interactor, system integrity and user interface properties. The interactor properties are general properties of the Basic LOTOS specification of the interactors, making it possible to check if these specifications have the intended behaviour. The system integrity properties have to do with the system architecture, like checking there is at least one path leading from a user connected gate to an application connected gate. These properties are more directly connected with technical aspects

of the specification, as such, and its consistency, than they are with properties of the user interface that is being specified, so they will be discussed no further.

Regarding user interface properties, templates are proposed for a number of properties. The properties can be classified in the following broad classes:

- reachability;
- visibility;
- continuous feedback;
- task related properties.

Reachability is defined as: "... given a user action, (...) it is possible to reach an effect which is described by a specific action." (Paternò 1995). The template is:

Template 2.2.1 (Reachability)

$AG([user\_action_x]E[true\{true\}U\{reachable\_effect\}true])$

Visibility is defined in the same way, the action associated with the desired reachable effect being an action in the output port of the Interactor.

Continuous feedback is similar to visibility but stronger. It states that after a user action a visible effect must be produced before another user action is accepted. The template is:

Template 2.2.2 (Continuous feedback)

$AG([user\_action_x]A[true\{\neg\ user\_action\}U\{user\_interface\_modification\}true])$

where *user_action* is the conjunction of all the user actions.

In the context of tasks, three formulae that classify various types of error are presented. These formulae allow the analysis of the impact of a given user action on a predetermined task, where a task is some target action. Erroneous actions are those that are not needed to perform the current task and are classified as minimal, recoverable and unrecoverable errors. Minimal errors are those actions after which it is possible to perform an action that is useful for the task.

Template 2.2.3 (Minimal errors)

$AG([minimal\_error](EX\{useful\_action\_for\_task\}true\ \&\ EF\ <task\_performance>\ true))$

Recoverable errors are those after which several actions must be performed before a useful action for the task is possible.

Template 2.2.4 (Recoverable errors)

$AG([recoverable\_error](EF\ <useful\_action\_for\_task>\ true\ \&\ EF\ <task\_performance>\ true))$

Unrecoverable error are those after which it is not possible to perform the task anymore.

Template 2.2.5 (Unrecoverable errors)

$AG([unrecoverable\_error]\neg\ (EF\ <task\_performance>\ true))$

The concept of 'task useful action' is not defined. This kind of information must be obtained elsewhere, possibly in the task specification. Note that this is a first example of where interaction with the human-factors community might happen.

A notion of task reversibility is also defined. The property expresses that once a task is initiated, an action can be performed that cancels the previous effects so that the task can be performed again.

Template 2.2.6 (Task reversibility)

$$
\begin{array}{l}
AG([user\_action\_request]EF\ <task\_performance>\ true]) \\
\& \\
AG([user\_action\_request] \\
\quad E[true\{\neg\ task\_performance\} \\
\qquad U \\
\qquad (\{action\_cancelling\_previous\_effects\} \\
\qquad\ E[true\{\neg\ task\_performance\}U\{user\_action\_request\}EF\ <task\_performance>\ true])])
\end{array}
$$

The notion of 'cancelling previous effects' is not defined and is not clear whether it refers to the whole Interactive System, or just to the fact that the task can be initiated again. This seems to be a consequence of the impossibility to characterise states: as states cannot be characterised, it is not possible to express that the system returns to the previous state. This, in turn, means that it is necessary to rely on this vague notion of action that cancels effects without really formalising what the action means. It must also be noted that in order to test task reversibility, both what actions were executed and which actions cancel their effects have to be known. So, the property has to be tested for very specific situations.

The property that a task can be performed from any state in the dialogue is expressed by the template

Template 2.2.7 (Task from any state)

AGEF(<*task_performance*> *true*)

Finally, templates to express the CARE properties (Duke et al. 1995) are also given. The CARE properties (Complementarity, Assignment, Redundancy and Equivalence) are a framework to evaluate the usability of Multi-Modal interactive systems. In the context of the approach alternative modalities are informally defined by considering that different channels represent different modalities.

Discussion

Unlike the previous approach, Paternò (1995) separates the user interface from the underlying system. This enables the verification of properties relating to visibility issues, and the analysis of dialogue control. It also allows for some base level assumptions about the user to be made, at the level of dialogue specification and analysis. These relate mainly to the notion of task. However, the separation between the two layers is such that there is no way to reason about the underlying system's state: the underlying system's specification is not part of the interactive system specification. Hence, it is not possible to reason about the relation between interface and system behaviour.

The approach is heavily based on the notion of event. Even tasks are defined only as a target event to be executed. So, achieving a task is performing the target event, independently of the strategy that leads to it. Like in the previous approach this is an oversimplified notion of task. And in fact, it is easy to devise two sequences of actions with the same final action but corresponding to different tasks, for example: `<select_landing_mode, engage_auto_pilot>` (to land an aircraft) and `<select_go-around_mode, engage_auto_pilot>` (to abort a landing). The solution here would be to *create* artificial actions, each corresponding to a task. These actions would need to be associated with specific sequences of events. Unfortunately, a growing number of actions will make the system harder to verify.

2.2.3 Using Lesar

The last approach using a model checking related technique is reported in (d'Ausbourg et al. 1996) (see also d'Ausbourg 1998, d'Ausbourg et al. 1998). In this case a model checker is not explicitly used. Rather, an automaton is obtained from a program in Lustre (a data flow language) and the Lesar tool is used to analyse the output flows of the automaton.

The Dialogue Specification

This approach is similar to that of Paternò (1995) in that it uses the notion of interactor to model the user interface. In this case, however, the starting point is not a task description of the system, but a description of the actual interface implementation in UIL (User Interface Language — see Heller & Ferguson 1994). Interactors are derived from these UIL descriptions, and modelled in Lustre. Lustre (Halbwachs et al. 1991) is a declarative data flow language for reactive systems. A Lustre program is a collection of nodes describing functional relations between input and output flows. As an example, the program

```
node edge (b : bool) returns (edge : bool);
let edge = false → b and not pre b; tel
```

defines a node named `edge` that receives a flow of booleans (`b`) and produces another flow of booleans (`edge`); The output flow is defined as `false` followed by the result of the expression `b and not pre b`, which at each instant is true if the present value in flow `b` is true and its previous value (`pre b`) was false. So, this node detects rising edges in the input flow.

The translation from UIL description into a Lustre hierarchy of nodes is done using a notion of interactor similar to the LOTOS interactor. A set of interactors are identified in the UIL description (corresponding to the widgets in the interface) and then nodes are written to model each interactor. A push button can be modelled by the following node (definition taken from d'Ausbourg et al. 1996):

```
node PushButton(Push, Release, On : bool)
  returns(Activation, Revvideo, Modif_pres, Selected : bool);
```

```
let
   Selected = Push and On;
   Activation = false → pre Selected and Release;
   Revvideo = from_to(selected, not selected);
   Modif_pres = edge(Revvideo) or edge(not Revvideo);
tel;
```

where `from_to` and `edge` are other Lustre nodes.

The nodes are then composed in a hierarchy matching the hierarchy of interface description in UIL. At this point some control nodes will be added to govern the interaction between the different *interactors*.

The Finite State Machine

In this approach, no finite state machine is generated prior to the verification process. In fact, as will be seen in the next section, for each property checked a new finite state machine is generated.

Checking the Specification

Properties to be checked are expressed in the same formalism as the interface. In fact, Lustre can be considered a subset of temporal logic. A property can be expressed as a Lustre node, the property will be true if the output flow of the node is true. The following node, for instance, tests if a change in flow `A` implies that flow `B` becomes true.

```
node Reactive(A,B : bool) returns (r : bool);
let r = implies(edge(A) or edge(not A), B);
```

In order to verify a property, the relevant output flows of the specification of the interface are fed into the node specifying the property. If the output flows of the resulting Lustre program are always true then the specification verifies the property. The expression to test if the push button above is reactive is:

```
node Verify_Reactive_PushButton(Push, Release, On : bool)
   returns (r : bool);
   varActivation, Revvideo, Modif_pres, Selected : bool;
let
   (Activation, Revvideo, Modif_pres, Selected) = PushButton(Push, Release, On);
   r = Reactive(Selected, Modif_pres);
tel;
```

The output flows of a Lustre program can be analysed by compiling the program, and using the Lesar tool to traverse the automaton generated by the compiler.

Discussion

The aim of this approach is to develop a software environment for assisting the formal validation of interactive systems. The approach suffers from the same problem as the previous ones in that it is restricted to the user interface layer. In fact, from the examples presented by the authors, it seems that the approach is targeted more to low level analysis of the interactions between the different components of the interface than with more high level questions about the interaction with the user. There is, for example, no mention of user related issues such as task.

While the use of the same language to model both the system and its properties seems to solve some of the problem of translation between LOTOS and FSM in Paternò's (1995) approach, this is done at the cost of using a less expressive language. There is, for instance, no mention of data types, and only boolean flows seem to be considered. The verification process is also less powerful, since besides safety properties, only a restricted form of liveness can be checked. An advantage of this proof process is that the automaton that is generated contains only the flows that are relevant for the property being checked. This way, the model checking process does not operate over the whole representation of the system, only over the result of combining the system with the property. This helps to avoid problems of state explosion.

2.2.4 Conclusions on Model Checking

The main difference between the approaches comes from the specification notations used. Abowd et al. (1995) adopted a simple and easy to use approach with the advantage of having tool support (Action Simulator). The approach might be too simple, however. In fact, for the verification to be useful it must be done at an appropriate level of detail, and Action Simulator was designed for very high level abstract specifications (Monk & Curry 1994). At this level the properties that are worth investigating might not be present. It is doubtful that so much should be given up for ease of use when the verification process, in itself, might be a complex task.

Paternò (1995) avoids this problem by using a more powerful specification notation. By using interactors, that are composed to build a complete specification of the user interface, he separates user interface information from the underlying system's information and is able to talk about properties specific to the user interface, like visibility. Unfortunately, information about the state of the underlying system is only available indirectly through events. Despite the use of a better specification notation, the verification has still to be done at the model checking level, and the translation of the interactors specification to a finite state machine might mean the version of the specification being analysed has more behaviour (allows more traces of events) than the original one. This seems to be a problem that will affect all the attempts to use powerful specification notations, as the specification will always have to be translated into a finite state machine in order to be model checked, and the expressive power of those is limited.

The use of Lustre somehow avoids this problem by taking the opposite direction and specifying the properties at the specification level. The language used, however, is very biased towards reactive systems, and does not seem to have enough expressive power to deal with richness of a user interface. In the next section it will be shown how Bumbulis et al. (1996) use a more powerful specification notation.

At the verification level, the main difference to be noted between the approaches have to do with the temporal logics that each approach uses. CTL focuses on the states and the transition between states, ACTL focuses on events and the sequences of events that can be generated, and Lustre focuses in the flow of information through the system. These different foci allow for different styles of analysis. ACTL formulae have to do with analysing the future to determine if some event will or will not happen under certain conditions (related to other events happening or not). CTL formulae, on the other hand, have to do with analysing the future to see if a system state will or will not be reached under certain conditions (this time related to other states being reached or not); however, as Abowd et al. (1995) encode events as state information, the former analysis can also be done. Lustre seems to be less expressive than the previous two, as only safety properties can be directly expressed. A restricted form of liveness can be expressed by negating safety properties (d'Ausbourg et al. 1998).

Different foci also raise different problems. ACTL has problems when trying to express properties that have to do with the state the dialogue is in (undo for example). On the other hand, CTL has some problems when trying to express that "*something is possible from a state*": as every state in the original PPS generates a set of similar states (one for each event that can come next), what can actually be expressed is that "*something is possible from a state* whatever action is taken" (which is stronger). This problem can be overcome by explicit elimination of undesired actions.

Although Paternò's (1995) approach is the more expressive, in the sense that the separation of the user interface from the underlying system allows for a better reasoning about properties specific of the former, the separation seems to be excessive. In fact, as the underlying system is not made explicit in the specification, it is impossible to reason about it directly and how it relates to the user interface.

2.3 Theorem Proving

Having seen how model checking is being used in the formal verification of user interface specifications, the alternative approach to system verification, theorem proving, will now be considered.

2.3.1 HOL

Bumbulis et al. (1996) use HOL (a Higher Order Logic theorem prover) in the verification of user interface specifications (see also, Bumbulis 1996). Their approach is based on a language of interconnected components (IL — the Interconnection Language). IL enables the specification of user interfaces by composition of the different components available. By creating implementations of each component in a toolkit, and models of each component in the HOL logic, these specifications can be translated into toolkit code (creating an implementation of the system), as well as to the higher order logic of the HOL system for formal verification.

An immediately obvious advantage of this approach is that the formalism used to perform the analysis, Higher Order Logic, is the same which is used to define the semantics of the IL language. In fact the IL models will be expressed directly in the

```
component Window(width,height)
component Dial(parent,x,y,width,height) set< changed>
component Slider(parent,x,y,width,height) set< changed>
component Main {
      f:Window(170,220)
      d:Dial(f,5,5,160,160)
      s:Slider(f,5,165,160,160)
      s.changed->d.set
      d.changed->s.set
}
```

Figure 2.6: IL description of window with dial and slider (taken from Bumbulis et al. 1996)

HOL logic. So, it can be anticipated that the problems arising from the use of different languages for specification and for verification in both Paternò's (1995) and Abowd et al.'s (1995) approaches will not happen. This should also avoid the problems of lack of expressiveness as found in (d'Ausbourg et al. 1996).

The Dialogue Specification

As said above, user interfaces are specified as sets of connected components using IL. The notion of component in IL is similar to that of interactor (especially the LOTOS version) although is has been developed to more closely resemble that of widget in a toolkit, in order to allow for an easy implementation of the specification. A component is defined as having a set of ports (input and output) and observers. Different components can be connected through their input and output ports, much in the same way as a widget's methods and call-backs are connected. Input ports are also the mechanisms by which users manipulate the components. The authors do not show how output to the user can be specified. Intuitively, observers should be used for this, however, their use seems to be limited: observers can only be mentioned when specifying the connections between ports.

A IL description of a window with a dial and a slider is shown in Figure 2.6. Four components are specified: "Window", "Dial", "Slider", and "Main". The first three are simple components. Each declaration includes the name of the component, its parameters, and its ports and observers (no component declares observers, in

$$
\begin{array}{l}
\mathsf{Slider}\ i\ c\ s\ q\ e\ \mathit{parent}\ x\ y\ \mathit{width}\ \mathit{height}\ \mathit{set}\ \mathit{changed} = \\
\quad (\mathit{set} = (\lambda\, v.\mathsf{if}(q(\lambda\, n.\neg\ (n = v)))(\mathsf{assign}(s(\lambda\, n.v))\ {}^{\circ}_{9}\ \mathit{changed}\ v))) \\
\quad \wedge\ (i = 0) \\
\quad \wedge\ (c = {}^{\exists} v.\mathsf{atomic}(\mathsf{numEv}\ e\ v)(\mathit{set}\ v)))
\end{array}
$$

Figure 2.7: HOL predicate defining a Slider (taken from Bumbulis et al. 1996)

this case). Ports marked with "<" are input ports, and ports marked with ">" are output ports (observers are marked with "@"). Component "Main" is a composite component, defined in terms of the other, simpler, components. In particular it specifies the connections between the ports of its constituents. Figure 2.6 specifies how components should be connected, but says nothing about their behaviour. For that, a command language based on (Nelson 1989) is used. The semantics of the language constructs is defined by HOL predicates over sets of runs. Here a run is a sequence of event/resulting state pairs representing possible behaviours of a component. These predicates will consist of a series of conjuncts specifying the behaviour of each of the ports and observers. Instead of defining a syntax for the command language, components are defined directly by HOL predicates. The predicate for modeling "Slider" is shown in Figure 2.7. For a full explanation of this predicate (and of the IL language in general) see (Bumbulis 1996). Briefly, i is the initial state, and c the behaviour of an instance of Slider. The predicate is the conjunction of three sub-predicates. The first defines the behaviour of channel "set": if the value received through "set" is different from the current value, then the new value is stored and propagated through "changed". Note that s is used to apply assignments to the state, while q is used to calculate predicates on the state. As the predicate for "Main" will show, this allows the state of the overall interface to be defined as the product of the states of individual components. The second sub-predicate defines the initial state of the component. The final sub-predicate defines the behaviour of an instance of "Slider": it can repeatedly receive "set" events.

The predicates for composite components, like Main, besides the usual conjuncts of simple components, also have conjuncts for each of its constituent instances. The predicate for component "Main" is shown in Figure 2.8. As stated in the IL description, "Main" is composed of a "Window", a "Slider" and a "Dial". Since "Main" has no ports or observers, only sub-predicates defining how constituent instances are

$$
\begin{array}{l}
\mathsf{Main}\ i\ c\ s\ q\ e = \exists\ i_1\ c_1\ i_2\ set_2\ i_3\ c_3\ set_3. \\
\quad \mathsf{Window}\ i_1\ c_1\ (\lambda f.s(\lambda(v_1,v_2,v_3).(fv_1,v_2,v_3)))\ (\lambda P.q(\lambda(v_1,v_2,v_3).Pv_1)) \\
\qquad (\mathsf{CONS}\ 1\ e)\ 170\ 220 \\
\quad \wedge\ \mathsf{Slider}\ i_2\ c_2\ (\lambda f.s(\lambda(v_1,v_2,v_3).(v_1,fv_2,v_3)))\ (\lambda P.q(\lambda(v_1,v_2,v_3).Pv_2)) \\
\qquad (\mathsf{CONS}\ 2\ e)\ \mathsf{VOID}\ 5\ 5\ 160\ 160\ set_2\ set_3 \\
\quad \wedge\ \mathsf{Dial}\ i_3\ c_3\ (\lambda f.s(\lambda(v_1,v_2,v_3).(v_1,v_2,fv_3)))\ (\lambda P.q(\lambda(v_1,v_2,v_3).Pv_3)) \\
\qquad (\mathsf{CONS}\ 3\ e)\ \mathsf{VOID}\ 45\ 165\ 160\ 160\ set_3\ set_2 \\
\quad \wedge\ (i = (i_1,i_2,i_3)) \\
\quad \wedge\ (c = c_1 [\!] c_2 [\!] c_3)
\end{array}
$$

Figure 2.8: HOL predicate for Main (taken from Bumbulis et al. 1996)

connected, the initial state, and the behaviour of an instance, are present. Its state is a tuple made of the states of its constituents ($i = (i_1, i_2, i_3)$). Note how the third and fourth parameters (corresponding to s and q) are used to select the appropriate state component. Behaviour is defined by the parallel combination of the behaviours of the constituents ($c = c_1 [\!] c_2 [\!] c_3$).

Verifying the Specification

Given the definition of all the components properties must then be verified. Properties to be verified are expressed as predicates over sets of runs. Verifying that a model has a given property P amounts to proving the following theorem:

$$\forall i\ c.\ \mathsf{Main}\ i\ c\ (\lambda f.f)\ (\lambda P.P)\ [\,] \Rightarrow \mathsf{P}((\lambda s.s = i) \rightarrow \mathsf{do\_od}\ c) \tag{2.1}$$

What the formula expresses is that for every possible initial state (i), and for every possible behaviour (c), of Main, P is a valid property of the execution of the behaviour from the initial state (do_od is a command that repeatedly activates c until no longer possible). That is, P is a universal property of the dialogue. It should be noticed that this is a safety property, and as such can, in general, be proved by a model checker if there are a finite number of states in the model.

In this approach, in order to perform proofs, a logic that allows reasoning about the properties of interest must be mechanised. Hence, a logic is presented to reason about this type of property of the overall behaviour of the specification. The authors then show how it can be proved that the slider and the dial will always be synchronised. This proof takes 20 steps and relies on the previous proof of a lemma. Although it is a

fact that by doing the proof a greater insight is acquired about the specification, such a level of effort seems somewhat exaggerated given the system that is being analysed, and the fact that this type of property could be proved automatically in a model checker.

Discussion

Besides this problem with complexity of use, the approach also seems limited in the type of analysis it provides. This seems to spring from two main factors. At the specification level, only the interface is considered, there is no mention of the underlying system's state or of user related concerns. Although it can be imagined that some special component could be developed to model the underlying system, this is not shown. Further, what is specified is not so much the interaction between the users and the interface, but the user interface architecture and how the different components communicate with each other. Output to the user seems to be defined implicitly by the states of the components, so this approach suffers from the same problems as Abowd et al.'s (1995) and properties related to how the user perceives the interface do not seem to be verifiable.

At the verification level, not using a logic that can capture temporal properties limits the scope of analysis to invariants of the user interface. So, reasoning about behaviour can also be a problem: only safety properties can be verified.

2.3.2 Conclusions on Theorem Proving

In conclusion, although the approach uses a powerful verification environment, it has two main drawbacks. The specification style and the logic used do not allow reasoning about some of the important aspects of interaction, additionally the verification process is quite complex. The expressive power of the theorem proving approach does not seem to be explored fully. One problem with the approach above might be that a rather simplistic, too software engineering biased, view of interaction is being taken. In order to analyse interaction between a user and a system, more than just the user interface, device level, description must be taken into account. Chapter 3 will address this.

2.4 Recent work

Recently, some interesting work has been reported on which addresses the verification of user related issue in safety critical systems. Doherty et al. (1999) use HyTech (Henzinger et al. 1997), a tool for reachability analysis of hybrid automata, to analyse systems with a continuous component (as opposed to the discrete nature of, for example, SMV). Rushby (1999) uses Murφ (Dill 1996), another state exploration tool, to reason about automation surprise in the context of an aircraft cockpit. Work is also being carried out at NASA Langley Research Center regarding the analysis of mode confusion in digital flight decks. This work concerns the use of model checking (Lüttgen & Carreño 1999), as well as theorem proving (Miller & Potts 1999). Since this is very recent work, it will be addressed during the thesis, where appropriate, and also more specifically in Chapter 7.

2.5 Considering the user

One of the distinguishing points about the recent approaches above is the inclusion of considerations about the user. As the chapter has shown, this was one of the weaknesses of all the reviewed approaches. One possibility for accounting for the user would be resorting to user modelling (cf. PUMA Blandford et al. 1997). This type of approach, however, is mainly concerned with user issues. Additionally, user modelling falls in the area of expertise of human-factors disciplines like psychology. Hence, the inverse problem would potentially happen: not enough consideration of system issues.

A different approach is to merge system and user models into a single model. Duke et al. (1998) use interactors to model both the system and an ICS based model of the user (what they call Syndetic Modelling). ICS (Interacting Cognitive Subsystems — Barnard & May 1995) is an architecture containing a set of functionally distinct subsystems, aimed at representing human information processing as a system of distributed cognitive resources. Expressing the (syndetic) model using a formal logic, allows for mathematical proofs about the model to be carried out. Degani (1996) divides a model into a number of sub-models: Environment, User-Tasks, Interface, Control Mechanism, and Plant. Each of these models is specified using Statecharts

(Harel 1987). In this case, the task model plays the role of the user. Statemate (a tool for Statecharts) is used to develop and analyse the specification. However, Statemate does not provide (yet) a true (formal) verification environment. Rushby (1999) and Doherty et al. (1999) also consider the use of a model of the user. In (Rushby 1999) a model of the system is first developed, this model is afterwards *enriched* with a user model. As in (Degani 1996) the user model is built from the user tasks. It is also shown how *forgetfulness* could be modelled using nondeterminism. In (Doherty et al. 1999) both models are built separately, and then combined for analysis. In this case, the model presented is very simple and consists of the actions the user might engage in at the interface.

While it is debatable whether a single user model will ever be able to adequately capture all the user issues which are relevant for usability verification, it is clear that the user models utilised in these last three approaches (which all use some form of automated support) are far from the ICS based model in (Duke et al. 1998) (which has the strongest cognitive psychology grounding). This can be attributed to two main factors:

- expertise — the developers in the last three approaches are primarily computer scientists, user modelling requires expertise in areas such as cognitive psychology which computer scientists and software engineers will usually not possess;
- complexity — any attempt to produce realistic user models will result in complexity, This poses problems when trying to manipulate and verify such models (remember that what must be verified is the composition of user and system model, the system model might itself be quite complex).

Ideally then, a software engineering approach to usability verification should try to perform reasoning without resorting to an explicit user model.

Regarding the approaches above, two points should be noted. Firstly, assumptions are made about the user in order to build the user model. Secondly, the analysis is mainly concerned with the system, and with those properties of the system that potentiate a good interaction process with the user (from a cognitive psychology point of view, this might not be the case). Hence, given a number of assumptions about

the user materialised as a user model, the system will be considered satisfactory if a number of properties about the interaction are satisfied. Since the assumptions can be considered as fixed, the success of the verification depends only on the properties of the system. Clearly, it would be possible to avoid developing a user model by directly checking those system properties that will impact usability. Obviously input from human-factors disciplines would still be necessary, since assumptions about the user must be used to identify those properties that are relevant for usability. However, the role of human-factors experts would now be clearly delimited, and all the modelling and analysis could be carried out in a software engineering context. This is the style of approach which will be pursued in this thesis. (Doherty et al. 1999) also goes somewhat in this direction when it encodes information about user tasks in the properties to be verified, thus circumventing the simplicity of the user model.

2.6 Conclusions

This chapter has described the main approaches that have been so far proposed regarding the use of automated reasoning in the development of interactive systems. Only approaches that resort to the use of automated reasoning were considered, since that is the focus of the thesis. Although the chapter focuses on automated reasoning only, it should be noted that this type of approach should not be seen as some type of definitive answer regarding usability evaluation. As pointed out in Chapter 1, a final proof of usability can only be obtained when users actually use the system. This, however, leaves usability evaluation to very late in the development life-cycle. The use of automated reasoning aims at allowing for usability issues to be addressed early in development.

Some conclusions can be drawn from the review. A first conclusion drawn from Abowd et al.'s (1995) work is that a clear distinction must be drawn between the user interface and the underlying system. The acknowledgment of the need for this separation is not new, going back to the Seeheim Model (Tanner & Buxton 1985). From the other approaches it can be seen that, although necessary, this separation must not be done in such a way that information is lost about the underlying system. It is

also self evident that in order for reasoning about interesting aspects of an interactive system to be viable, sufficiently expressive notations must be used.

The interaction mechanisms by which communication between the user interface and the users is achieved have not been addressed thoroughly. In fact, all the approaches are heavily based on the atomic notions of either event or status phenomena, little or no attention being payed to other concepts such as task or mode. Further, it is important to take user concerns into account in order to enable the analysis of the interactive system against the users' needs and capabilities. Although some recent work has started to address this issue, this is clearly an area where further work is needed. One possibility would be to consider the use of models of the user. Besides greatly increasing the complexity of the final models to be verified, this type of approach draws heavily on human-factors expertise. In this thesis a software engineering approach is sought. So, the emphasis will be placed on system properties that impact usability.

The next chapter will further discuss the merits and shortcomings of the approaches described herein. In doing so, it will identify how to better explore the potential of automated reasoning in this area.

Chapter 3

The Role of Formal Verification

The previous chapter has covered the main work performed in the area of interactive systems automated verification. In order to understand better the relative merits and pitfalls of the work described therein, this chapter addresses the issue of what role formal verification should play in the development of interactive systems. A framework for the comparison of the different approaches is proposed. The integration of verification into the development life-cycle of interactive systems is also discussed, and a new approach proposed. The chapter ends with the description of the interactor language which will be used in the remainder of the thesis.

Some of the material presented in this chapter has previously appeared in (Campos & Harrison 1997, Campos & Harrison 1998, Campos & Harrison n.d.).

3.1 Introduction

To improve on the work described in the previous chapter, it is first necessary to understand what the strengths and weaknesses of that work are. This can only be done if an understanding of what should be expected from the application of formal (automated) reasoning to the field of interactive systems development exists. This understanding will not only help in analysing the available approaches, but will also be useful in guiding further efforts in the area.

Regarding interactive systems, the main concern is to assure the quality of the system in terms of its interaction with the user. The problem then is how to measure

the quality of the user interface. Traditionally, quality will be measured against a set of properties that the system/artifact must exhibit. Trying to devise a meaningful set of properties, that should be true of an interactive system in order to guarantee its quality, poses a problem: there is no magic recipe for easy interactive systems building. Although there are a number of human-factors oriented studies which resulted in design guidelines and rules, these cannot be turned directly into a set of properties that all systems must obey. Guidelines are (or, at least, should be) of a qualitative and high level nature, which means that they are not easy to verify in a *mechanical* way, and are not suitable for such an approach. They must first be turned into concrete properties. A typical guideline might be: "*Reduce cognitive load*" (Preece et al. 1994). Design rules, on the other hand, are about very specific interface features, which means that formal verification might not be, in many cases, the best approach. A typical design rule might be: "*A menu should not have more than seven entries*".

In the field of software engineering, lists of properties have also emerged (see, for example, Sufrin & He 1990, Harrison & Duke 1995). However, these tend to be governed mainly by the specific style of specification being used, and what that style allows to be expressed. Their relevance towards usability is not always completely clear. In (Abowd et al. 1995) and in (Paternò 1995), sets of properties of interactive systems that can be checked automatically are proposed. Again these seem driven more by the capabilities of the verification formalism used than by pure methodological concerns.

One of the main reasons for the problem above is that interactive systems form a heterogeneous class of systems. In fact, the only common requirement is that the system interacts with a human user effectively. This accounts for systems from airplane cockpits, or control rooms of nuclear power plants, to text editors or spread sheets. If a specific type of system was chosen, the task of identifying relevant properties would become more feasible. However, the aim of the present thesis is to address the problem of automated deduction for usability reasoning in broad terms. The starting point then is simply that the system will have a user interface. So, instead of trying to establish a list of properties, this chapter tries to identify some specific issues related to interactive systems design, about which the designer might wish to be assured that the system is satisfactory. This will enable the definition of broad classes

of properties of interest. These classes are defined not in relation to the underlying verification framework, but in relation to the domain of analysis. This is different from the lists of properties proposed by (Abowd et al. 1995) and (Paternò 1995). It is also different from, for example, Manna & Pnueli's (1995) classification of temporal formulae to specify program properties. Their taxonomy has no direct connection with the domain of analysis as it is based on the temporal operators that are used to express the property.

Once a framework identifying the domain of analysis is established, the chapter progresses by looking at the reviewed approaches in the light of the framework. The chapter then discusses how the verification of properties that relate to these issues raised by the framework might integrate with the overall process of interactive systems development.

3.2 A framework for theorems on usability

As noted above, a framework identifying the major issues that should be taken into account when reasoning about usability is needed (both to better compare the reviewed approaches, and to guide further work). This section deals with developing such a framework.

The proposed framework relates to the issues which should be considered when reasoning about interactive systems. Initially it will be used to compare the approaches reviewed in Chapter 2. Afterwards it will also help guide the analysis of interactive systems designs (see Chapter 6). It should be stressed that the framework developed in this section is primarily concerned with what is involved when reasoning about usability. In particular, it is not intended as a theory of interactive systems. The relevant issue is what should be analysed of a design, not how the design should be specified. In fact, this thesis is primarily about the interaction between a user and a system, not about how systems are specified. Regarding the use of formal methods in the specification of interactive systems see, for example, (Harrison & Thimbleby 1990, Palanque & Paternò 1998, Doherty 1998).

3.2.1 Entities involved in interaction

As pointed out in (Doherty & Harrison 1997, Doherty 1998), the process of building the user interface of a system can be seen as a reification of the functional level of the same system. Reification (see Jones 1986) encompasses two steps: data reification (i.e., the development of an alternative, less abstract, representation of the state of the system), and operation reification (i.e., the development of operations on the new representation which are equivalent to the operations in the original representation). Interactive systems development, then, can be seen as the process of reifying the system state to construct a presentation for that system (data reification), and additionally providing user interface actions that match the functional core level operations of the system (operation reification).

Typically, the state of a system can be modelled by a set of attributes, and presentations modelled by a set of percepts (Duke & Harrison 1994*b*). In the present context the term *percept* means a presentation element that the users can perceive through their senses. Note that this is slightly different from the use of the term in cognitive psychology literature. The issue of emergent features[1], in particular, is not addressed. Calling ρ to the mapping from system state to presentation, then:

$$\rho : S \rightarrow P \qquad \text{with } S = \mathbb{P}Attributes$$
$$P = \mathbb{P}Percepts$$

Assuming a deterministic system (see Dix 1990, for a discussion on the role of non determinism in interactive systems), operations will map system states to system states, and interface actions will map presentations to presentations. Hence, as proposed by (Doherty 1998), Jones's (1986) *Function Modelling* diagram (Figure 8.3, page 213, in Jones 1986) can be adapted to produce Figure 3.1. Here, ρ is shown building a user interface presentation for a given system state ($\text{state}_{\text{system}}$), and R is the retrieve function which performs the opposite transformation, i.e., R maps a presentation into the system state that has generated it.

[1] Emergent features: features that "may emerge from a presentation due to the way that the human cognitive system collects and organises information" (Duke & Harrison 1994*b*).

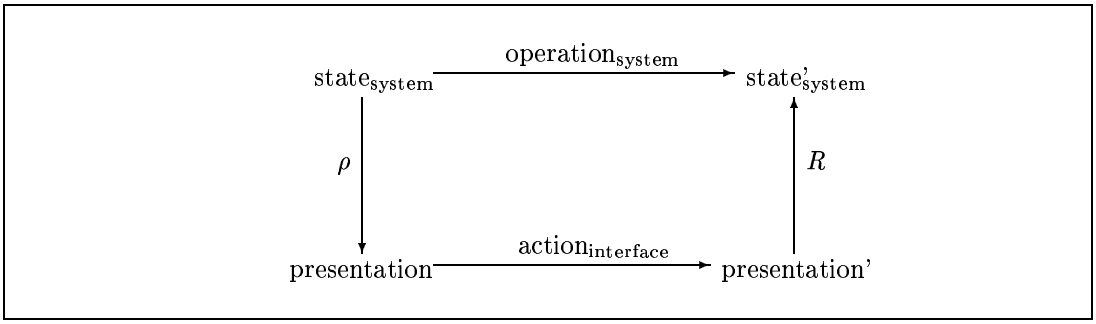

Figure 3.1: User Interface as a Reification

Note that both ρ and R are being considered one-to-one. This is assuming that each system state will have a distinct presentation, and that each different presentation will correspond to a different system state. Obviously this is a simplification. In many situations this will not be possible, or even desirable. This issue will be further addressed in Section 3.2.3. For now, this simplified situation will be enough to highlight the main concerns involved in reasoning about interaction.

As in (Jones 1986), the diagram in Figure 3.1 identifies the relation that must hold between a hypothetical operation ($\text{operation}_{\text{system}}$), causing a state change from $\text{state}_{\text{system}}$ to $\text{state}'_{\text{system}}$, and a hypothetical user interface action ($\text{action}_{\text{interface}}$), causing a presentation change from presentation to presentation$'$. This relation must hold if the user interface action is to be an appropriate reification of the system's operation. The above requirement can be expressed as (cf. Jones 1986):

$$R \circ \text{action}_{\text{interface}} \circ \rho(\text{state}_{\text{system}}) = \text{operation}_{\text{system}}(\text{state}_{\text{system}}) \tag{3.1}$$

Note that this equation represents the situation in Figure 3.1. Hence, it is referring to a specific system state, operation, and interface action. Obviously the equation could be generalised over states, operations and/or actions. At this stage, however, the relevant issue is not developing actual properties, rather identifying the kind of properties which will be relevant of the relationships between the entities involved in interaction. The equation above suffices to capture (by example) the intended idea of correctness.

Following from the tradition of formal systems development (cf. Jones 1986), Equation 3.1 defines the correctness of the reification in terms of the system state. However,

regarding interactive systems, the interesting questions are related to how the user perceives that state. In particular, the user interface should enable the user to build a correct understanding of the system's state. Hence, the correctness of the interactive system is dependent on how it conveys information about the state of the system to the user, and not only on how it reacts to user action. Correctness should be defined in terms of the interface, not the system state. An initial attempt at this would be to rewrite Equation 3.1 to:

$$\text{action}_{\text{interface}} \circ \rho(\text{state}_{\text{system}}) = \rho \circ \text{operation}_{\text{system}}(\text{state}_{\text{system}}) \tag{3.2}$$

Correctness is now defined in terms of the interface presentations (what the user perceives). How/when can it be said that two presentations are the same? This is obviously dependent on the particular use that the interface is being put to, not to mention the perceptual capabilities of the user.

In fact, the presentation describes the physical structure of the interface only. It says nothing about how such structure will be perceived and interpreted by a user. This last issue is the subject area of cognitive psychology, which is an active field of research. Approaches such as, for example, GOMS (Card et al. 1983), Programmable User Modelling (PUM — Young et al. 1989), or Interacting Cognitive Subsystems (ICS — Barnard & May 1995) attempt to develop models of human behaviour (*user models*) which can be used to reason about how users will interact with systems. From a software engineering perspective, these models have the problem of being completely separate from the system models which are under analysis. This has lead to the study of approaches which integrate both system and user models into a single model (Duke et al. 1998). As explained in Chapter 2, in this thesis the development of explicit user models is avoided and a different approach is used: incorporating assumptions about the user into the verification stage. Hence, as already hinted above, two issues are relevant at this stage: perception and interpretation. Perception deals with how users will perceive the presentation of the system. Assumptions about what the users will perceive can be used to determine what to analyse of the system. For now the relevant point is that perception must be considered in building a framework for interactive

systems reasoning.

Regarding interpretation, the interest lies in the models users build of the perceived world: their *mental models*. For a discussion on mental models see, for example, (Newman & Lamming 1995, Chapter 13) or (Preece et al. 1994, Chapter 6). Assuming a propositional representation view, a mental model can be seen as a set of predicates expressing knowledge the user has about the system state. Hence, the decision as to whether two presentations are the same should be made in the context of the mental models which they enable the user to construct, and the relevant retrieve function becomes the mapping from the interface to the user mental model of the system (i.e., perception).

The introduction of mental models into the framework, enables the consideration of a further concept: *goals*. When users interact with a system, they do it with a purpose (i.e., a goal). This goal can be expressed as a target mental model which they want the system to conform to given the current mental model of the situation. Hence, the concept of goal can be interpreted as a mapping from current mental models to target mental models. User interface actions relate to goals much in the same way that they relate to system operations, the interface must provide actions that match the users' goals when using the system.

When these new concepts are introduced into the diagram in Figure 3.1, the result is the diagram in Figure 3.2, which takes the user into consideration. Note that this diagram corresponds roughly to the "Stages of Execution" in (Norman 1988).

It is now possible to identify the three basic entities involved in the interaction:

- functional core — the system state and its operations;
- user interface — the presentation and possible user actions;
- user — the person using the interactive system (building a mental model of the system and trying to fulfill goals by issuing commands).

Additionally, the diagram in Figure 3.2 has introduced two additional mappings:

- perception — this mapping captures what is assumed the user will perceive of the user interface presentation. It can be seen as a filter which the user applies to the

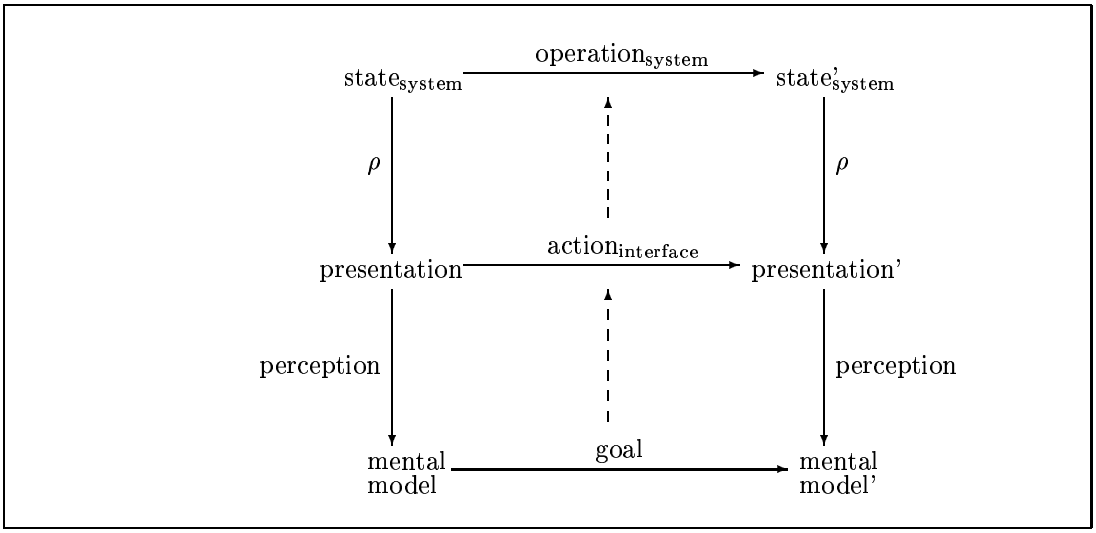

Figure 3.2: The different models of an Interactive System

information presented at the interface, in order to construct a mental model of the system state. How users will perceive the user interface will depend on the actual system being analysed. The type of property and user being considered also has a bearing on what perception relation to consider. The perceptual capabilities of the user are very much a human-factors issue. Hence, defining the appropriate perception relation, for a specific system, becomes part of a process of discussion between software engineers and human-factors experts, over the design proposed for that system.

- goal — this mapping captures the user objectives when using the system. Again, the definition of what the user goals will be can become part of the process of interdisciplinary discussion between software engineering and human-factors experts. At this stage a simple one-to-one relation between goals and interface actions is being assumed. This is further developed in Section 3.2.3

The framework developed so far, although a simple one, can already be useful in looking at the interaction process. In the case study in Chapter 6, for example, the interface of a digital audio-video communications system is analysed, regarding the effectiveness of the display in capturing the access capabilities that are allowed to users. There, system states are sets of attributes modelling how the information pertaining to users' accessibility and availability levels are stored internally by the

system (see Figure 6.2, on page 197). Presentations are sets of percepts, in this case the identification of the user selected as callee, the door state of that user, and the buttons corresponding to the different types of connections (see Figure 6.3 on page 198, and also Figure 6.1, on page 193). The mapping between the system state and the percepts is expressed by the axioms in Figure 6.3.

The users' mental models being considered deal with what users believe are their capabilities in connecting to other users, and preventing connections from other users to themselves. As noted previously, mental models are not explicitly constructed. Instead, they are embodied in the properties which are selected for verification. For example, the property checked on page 203, corresponds to the mental model of a user which believes that, if the callee's door is open, than it is always possible to establish a connection to the callee.

The next sections deal with identifying interesting relationships between the entities identified above, and also with adding some more structure to the framework. The framework is presented in Section 3.2.4.

3.2.2 Properties of the interaction

In this Section, some equations are developed that enable a characterisation of the main types of properties which are of interest when reasoning about interactive systems. The equations put forward should not be interpreted as actual properties to verify but simply as templates expression interesting types of questions.

In purely representational terms, an interface is said to be correct if its presentation enables the user to build an accurate model of the underlying system state. For the example in Figure 3.2[2] this can be expressed as:

$$\text{assumptions}_{\text{user}}(\text{perception} \circ \rho(\text{state}_{\text{system}})) = \text{assumptions}_{\text{system}}(\text{state}_{\text{system}}) \tag{3.3}$$

That is, the user's mental model of the system matches the system state, in so far as the model relates to assumptions about what is considered relevant of the interaction

[2] Unless stated otherwise, all the equations in this section refer to the illustrative example in Figure 3.2.

process. Note that this equation corresponds to the model proposed in (Doherty & Harrison 1997, Doherty 1998) for representational reasoning.

The equation above considers only the vertical axis in Figure 3.2, and relates to what Norman calls the Gulf of Evaluation: "the amount of effort that the person must exert to interpret the physical state of the system" (Norman 1988). Introducing the horizontal axis allows reasoning about Norman's (1988) Gulf of Execution, which can be performed using the following type of theorem:

$$\text{goal} \circ \text{perception} \circ \rho(\text{state}_{\text{system}}) = \text{perception} \circ \text{action}_{\text{interface}} \circ \rho(\text{state}_{\text{system}}) \tag{3.4}$$

The equation compares the result of executing an interface action with the goal the user had in mind.

Equation 3.4 considers the user side of the interaction only (user goals, and interface actions). Reasoning about whether interface actions and functional core operations are consistent can be performed by proving that (cf. Equation 3.2):

$$\begin{aligned}\text{perception} \circ \text{action}_{\text{interface}} \circ \rho(\text{state}_{\text{system}}) \\ = \text{perception} \circ \rho \circ \text{operation}_{\text{system}}(\text{state}_{\text{system}})\end{aligned} \tag{3.5}$$

This equation expresses the need to verify that related system operations and interface actions are equivalent in terms of the user's perception of their effect on the system.

It is also possible to reason about the effect of specific actions/operations by relating the user's mental models prior to and after the action/operation:

$$\text{perception} \circ \rho(\text{state}_{\text{system}}) \bowtie \text{perception} \circ \rho \circ \text{operation}_{\text{system}}(\text{state}_{\text{system}}) \tag{3.6}$$

Where $\bowtie$ is a relation on user mental models, capturing the change in presentation which the user should perceive as result of operation $\text{operation}_{\text{system}}$. If, for example, the system being analysed is a speed dial for an airplane (see Section 5.3), and the property being analysed deals with the indicated airspeed of the airplane, then $\bowtie$ could be the "greater than" relation ($>$) testing whether the user would perceive an increase

in the indicated airspeed after some operation $operation_{system}$.

These equations cover all of the graph in Figure 3.2. They illustrate the basic types of properties that can be asked of an interactive system design regarding its quality:

- Perception (or Evaluation) — the relation between the user's view of the system and the actual system state. The issue of perception is primarily a human-factors one. Hence, identifying desirable perception type properties will be part of a process of discussion between software engineering and human-factors experts. Perception concerns what is shown at the user interface, how it is shown, and how the users perceives it. Perception includes questions like: "*do actions/operations have appropriate feedback?*" (cf. Equation 3.6), or "*will the user correctly perceive the displayed information?*" (cf. Equation 3.3).

- Execution — the relation between the user's goals and the interface actions provided. Execution type properties deal with what can be done at the user interface, how it can be done, and how the way it can be done relates to the user's way of doing it. This type of property will help develop an understanding of how users will use the system, and how well the system fits the users needs. Again, this prompts for discussion with the human-factors side of interactive systems design. Execution includes questions like: "*can the effects of actions be undone?*" or "*how do the actions at the user interface match the user's goals?*" (cf. Equation 3.4).

- Reliability — the relation between the interface actions and the system operations. Reliability type properties deal with how the user interface and the underlying system work together. This is the more software engineering oriented type of properties. Properties of this type help build an understanding of how well the interface layer fits the underlying functionality of the system. Reliability includes questions like: "*does the same action/operation always have the same effect?*" (cf. Equation 3.5) or "*does some predicate on the state of the system always hold?*".

As stated initially, this list is not a collection of desirable properties. Instead, it highlights the relationships between the entities involved in interaction. The equations,

in particular, act only as examples of what issues actual properties should address, not as rigid templates. The use of quantification, for example, enables greater expressive power. Determining whether there is an action which allows the user to achieve some specific goal, could be done with (cf. Equation 3.4):

$$\exists_{a:\mathrm{Action}} \bullet \mathrm{goal} \circ \mathrm{perception} \circ \rho(\mathrm{state}_{\mathrm{system}}) = \mathrm{perception} \circ a \circ \rho(\mathrm{state}_{\mathrm{system}}) \tag{3.7}$$

3.2.3 Adding structure

The framework above, however, is still rather simplistic. In particular, it assumes a one-to-one mapping between user goals, interface actions, and system operations. In reality things are usually not that simple. Additionally, only a general notion of action has been mentioned, and nothing was said about how perception occurs. The actual mechanisms through which interaction is performed need addressing.

Normally, dialogue between user and user interface proceeds through the exchange of information. At the lowest level, information is conveyed either by discrete exchanges of data (*Events*) or by continuous flow of data (*Status Phenomena*) (Dix & Abowd 1996). The classification of a given interaction mechanism will depend greatly on the chosen level of abstraction. A video clip, for instance, might be seen as a continuous flow of information, or as a sequence of discrete values.

Over these atomic interaction mechanisms, several layers of structure may be identified. First of all, the interaction mechanisms themselves have different characteristics, conveying information through the different senses of the user. Even the simplest of interfaces will usually have visual (the screen), audio (at least the beep) and, more subtly, tactile (the keys in the keyboard) feedback. The type of the interaction mechanism in relation to the sense that it addresses is called its *Modality*.

Another level of structure is introduced by the mappings between goals, actions, and operations. From a traditional reification point of view, the reification of an operation might result in several actions, and vice-versa. From the point of view of the user, it is not realistic to expect all user goals to be achievable with single actions. Usually goals (which can be thought of as desired system states) will be achieved by

following an appropriate dialogue strategy. A goal with a set of possible strategies to achieve it is called a *Task*. It is important that the system supports the tasks that the user wants to perform. Furthermore it is also important that the strategies the system provides to achieve a given goal are made clear, as the user might not know them in advance.

Another important factor in the interaction between users and interactive systems is that the cognitive capability of humans is limited. This impacts the interface in two ways:

- the interface cannot present the whole system state to the user at once — except for the simpler cases this would mean too much information would be presented, and the user would be unable to process such an amount of information in a useful time. Instead, the interface has to provide the user with access to appropriate subsets of the system state;
- the interface cannot give the user access to all of the system functionality at once — again this would be too much information to be processed by the user. Instead, the interface has to provide access to different subsets of the functional core at different moments in time.

The fact that there are different subsets of state/functionality being accessed through the same interface means that the meaning of the interface components will depend on the current subset being used. The association of the interface components with a given subset of functionality is called *Mode*. While modes are useful to reduce clutter at the interface, they may also pose problems. For example, the user might misinterpret the current mode and be confused by the behaviour of the system.

Tasks and modes have to be considered during verification. Reasoning about tasks might imply reasoning about sequences of events. Here task is seen as the method the user has to perform in order to achieve some goal. Similar types of questions can be asked as before, but now they will not relate to a single action/operation but to a sequence of actions/operations. Determining whether there will be some sequence of user actions which will allow the user to fulfill a goal can be done by proving a relation

that has the following structure (cf. Equation 3.7):

$$\exists_{s:\text{Seq Action}_{\text{interface}}} \bullet \text{goal} \circ \text{perception} \circ \rho(\text{state}_{\text{system}}) = \text{perception} \circ \circ_{/s} \circ \rho(\text{state}_{\text{system}}) \tag{3.8}$$

where $\circ_{/s}(state)$ calculates the result of executing the sequence of actions in s starting from $state$.

This type of equation starts becoming interesting when it is considered that the system interacts, not only with the user, but also with the rest of the environment. The system is thereby engaging in other operations than those directly resulting from user action. This will have an impact on task analysis since, for example, the system might autonomously change mode while the user is engaged in accomplishing a task. A question is raised about what the consequences of the mode change will be in relation to the successful completion of the task. Temporal logics become particularly useful here, since they allow the expression of concepts such as possibility, inevitability, etc. As an example, in Paternò's approach (see Chapter 2), the property above could be expressed as:

$$E[true\{true\}U\{\text{action}_{\text{goal}}\}true] \tag{3.9}$$

The equation states that a sequence of actions will exist which ends with an action ($\text{action}_{\text{goal}}$) asserting the fulfillment of the goal. Note that this equation does not directly relate the state of the presentation before and after the sequence of actions. Since in ACTL it is not possible to talk about state structure (attributes), an action was introduced to assert the goal (as is done in Paternò 1995). This, however, is a rather artificial way of expressing the goal. If the structure of the state is important, an alternative is using the CTL logic. But then the actions would be lost. Chapter 4 addresses this issue.

A different type of task is that of obtaining some piece of information from the system. This involves the perceptual and analytic capabilities of users. They will have to be able to interpret and analyse the information provided about the system state through the user interface. Hence, this type of analysis falls under perception.

Modes identify specific subsets of the underlying system that become available to the user. They can be seen as partitioning the user interface into a number of equivalence classes which determine the meaning of specific elements. Modes then have impact on the mapping that is made by ρ, and also on the association between user actions and system operations. Typically the current mode will be defined by the state of the system/interface. If such information is built into the models, it then becomes possible to verify, for example, whether the user will correctly perceive the current mode, whether some action will achieve the desired goal in a given mode, or whether an action has the same effect in all modes.

3.2.4 The Framework

All the elements identified above can now be put together to form a framework which can be used when reasoning about the application of verification to interactive systems design. The three basic entities involved in the interaction are the user, the user interface, and the underlying system. Interaction progresses through a number of interaction mechanisms that are made available to the user by the user interface. Events and status phenomena can be thought of as classes of atomic interaction elements. Modes describe the structure of the connection of these elements with the functional core. Tasks describe the structure of the use that can be given to the same elements. Modalities define the structure of the elements themselves. Finally three classes of properties have been identified: Perception, Execution, and Reliability. Table 3.1 summarises the framework.

3.3 Problems with current approaches

The framework in Table 3.1 can act as a checklist against the approaches described in Chapter 2. Table 3.2 summarises the results of such an analysis. Each row on the table relates to one of the approaches reviewed, identified by the tool used. As discussed in the review (Chapter 2), all the approaches mentioned there focus almost exclusively on the user interface. This has an impact on how they are able to deal with the interaction mechanisms, and also with what types of properties can be addressed.

Entities		User
		User Interface
		Underlying System
Interaction Mechanisms	Atomic	Event
		Status phenomena
	Structure	Modality
		Task
		Mode
Properties		Perception
		Execution
		Reliability

Table 3.1: Framework for reasoning about interactive systems

		SMV	Lite	Lesar	HOL
Entities	Users	×	×	×	×
	User Interface	$\sim^a$	✓	✓	✓
	Underlying System	$\sim^a$	×	×	×
Interaction Mechanisms	Events	$\sim^b$	✓	✓	$\sim^b$
	Status phenomena	$\sim^c$	×	✓	$\sim^c$
	Modality	×	✓	×	×
	Task	$\sim^d$	$\sim^d$	×	×
	Mode	×	×	×	×
Properties	Perception	×	✓	✓	×
	Execution	✓	✓	✓	×
	Reliability	✓	$\sim^e$	$\sim^e$	$\sim^e$

[a] Specified together. [b] Just input. [c] Just output. [d] Just as a target action or state. [e] Just of the user interface.

Table 3.2: Summary of the review

All of the approaches deal with interaction rather poorly. Mode is never considered, and task is only dealt with in a very simplified manner. Of modelling tasks as target states/actions, it can be said that goals are being modeled, not the tasks to achieve those goals. Modalities are only considered by Paternò (1995) (Lite). Except for d'Ausbourg et al. (1996) (Lesar), no other approach fully covers both events and status phenomena. However, d'Ausbourg et al. (1996) do not deal with either of the more advanced structuring mechanisms (Modality, Task, and Mode).

Regarding properties, the results must be read bearing in mind that none of the approaches adequately covers the three entities being considered in the definition of

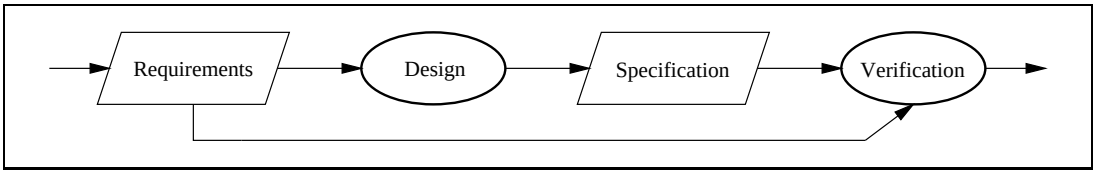

Figure 3.3: Verification as a final step in development

the property classes. In this way, the properties that can be verified by each approach are somewhat crippled, in the sense that they concern mainly the user interface, and not the relation between user interface and user or system. This is specially so in the case of reliability.

In this light, the results presented in Table 3.2 do not seem very promising. Anyone trying to improve on these approaches is faced with two problems:

- where to find the expressive power needed to model all the relevant aspects of an interactive system — it has been seen that these range from system behaviour to user perception;

- where to find the analytic power needed to analyse all the relevant types of properties — it has been seen that these range from temporal properties to state based properties.

These problems stem from the particular perspective that all the approaches have taken on the application of automated reasoning to interactive systems verification. Whatever the level of abstraction and tool used, all the approaches mentioned above tend to see verification as a final step in validating a specification against a set of desirable properties (see figure 3.3), and they all propose using a single tool to perform such a step. This represents what Henzinger (1996) calls "*Myth 1A*" of formal verification: that formal methods should be applied to complete designs. This type of approach raises a number of problems when attempting to put verification of complex systems into practice. This section points out what some of those problems are, and possible solutions.

3.3.1 On the role of models

Given the complex and semantically rich nature of interactive systems, it becomes impossible to make sure, at the outset, that all relevant details of a system have been tied down in a specification. It has been argued (Fields et al. 1997) that no available single specification notation is powerful enough to capture all relevant details of a given system. Instead, in order to validate a system design against a heterogeneous set of requirements, a number of specialised models will be needed. Each model shall take an appropriate viewpoint on the system, vis-à-vis the requirements being considered. Furthermore, as pointed out by Clarke et al. (1996), it should be possible to perform verification of partial specifications, of selected aspects of the system, in order to support evolutionary systems development.

So, instead of a single, monolithic, model, a number of smaller models should be used. The rationale here is that no single verification approach will be capable of addressing all relevant issues regarding interactive systems design and development. Instead, a range of approaches should be available, each supporting a specific type of analysis.

3.3.2 On the role of properties

As mentioned in Section 3.1 above, some authors have proposed templates for interesting properties that can be verified with their approaches (see Abowd et al. 1995, Paternò 1995). Abowd et al.'s (1995) templates deal mainly with what user interface states can be reached. On the other hand, Paternò's (1995) templates deal mainly with what user interface actions can be executed. Whether states or actions are used, an attempt is made in both approaches to map these states/actions to significant concepts at the user interface level: tasks, modes, visibility, etc. On the whole, these sets of templates tend to explore what model checking will allow (what meaning can be attributed to checkable properties), rather than being driven by the requirements needs.

By defining a set of property templates for a given class of systems, the authors try to pinpoint what properties are relevant and/or impact the quality of those systems. Regarding interactive systems, a number of factors complicate this purpose.

The properties that a system should exhibit are dependent on its requirements. These will vary from system to system. It is the case, however, that if the class of *Interactive Systems* can be identified, then there must be some common ground between all the system in that class. In this case the common ground is that all interactive systems communicate with a human user. And the single most abstract property that the design should ensure is that the system will be *easy to use*. Having identified this goal, it remains to be identified what makes an interface easy to use, so that it can be verified if a given interactive system model has the required characteristics. This poses two problems.

First of all, there is no global theory of interaction to act as a guide in design and as a global measure of quality. The lack of such global measures, means that *every system is a theory* and it is hard to make generalisations. So, the interesting properties will be dependent on the system being developed and it becomes hard to define a set of templates that are generically useful and applicable.

Additionally, given the complexity of interactive systems, and the number of different perspectives on them, it becomes difficult to make design decisions based on prescriptive theories alone. The ability to check the impact of possible choices in different aspects of the system could be very helpful in this. The validation process could be used as a guide to the process of choice as it provides insight into the problem that is being tackled.

A further problem with defining a set of templates is that the properties of interest will depend on the particular level at which the analysis is being performed. Consider, for instance, a template stating that it should be possible to execute a task from every state in the context of a hypothetical teller machine specification. The applicability of the template changes drastically with the level at which the analysis is done. If only individual interaction sessions between the machine and its user are considered, then the template should not be used: in each session, the user should be able to perform only one task[3]. However, if a broader look at the system is taken, to consider the successive interactions of different users, then the property is relevant as the user

[3]Consider a teller machine which does not present the user with the possibility of performing additional operations.

should be able to insert the card again to perform a new task. Thus, the role that properties play is dependent, not only on the system under consideration, but also on the particular level of abstraction at which the analysis is performed.

Not having a set of predefined properties, means that the verification process should be flexible enough to accommodate different styles of reasoning. Requirements, not the verification technology, should lead verification. Again, one way of achieving this is having different models and verification techniques, each capable of a specific type of analysis.

3.3.3 On what exactly is being proved

Seeing verification as the last step in design might lead to thinking of the verification process as an absolute measure of quality. However, the aim of formal verification is not proving a system correct, rather finding errors in the design (Clarke et al. 1996, Henzinger 1996). Correctness assumes some absolute measure of quality against which the specification can be verified. Trying to define it, one is faced with the problem of its own correctness. Furthermore, in HCI many of the quality considerations (ultimately relating to usability) are of a subjective nature and/or based in heuristics. This type of consideration is hard to write down in the form of properties to be checked.

The concept of usability is itself dependent on the domain of application. A cockpit notion of usability is very different from a spreadsheet notion of usability. Except, maybe, in that both interfaces should allow users to do what they want (or are supposed) to do *easily*. It seems then that formal verification should be seen as part of a wider process of *designing for quality*. Formal verification will enable the designer to, from very early stages of design, filter out flawed design, and compare alternatives against specific quality criteria. It can be said that the best verifications are those that fail, as they are the ones that allow a deeper understanding of the specification. Selecting the best design will probably mean considering the use of alternative approaches, based also on heuristics and experimenting.

Furthermore, seeing the verification step as a final step in the development process, and trying to use *off the shelf* properties, might also lead the designer to end up looking

at properties that are more relevant of the specification than of the system. Although this is not, in itself, negative (it is desirable to make sure that the specification is consistent and has no mistakes) care must be taken not to confuse the two types of properties. Consider again the property that every task is possible from every state. If the specification is done with finite state machines, and tasks are defined as some target state, this proves that those target states are always reachable. How much does it say about the system? It is quite different to need ten actions or one hundred actions to achieve task completion, but no information about this is obtained from verifying the property.

3.3.4 On the applicability of the techniques

The number of different perspectives that might be taken on interaction means that aiming at a global unified specification, encompassing all the relevant information, will yield a very complex model. This is especially problematic as comprehensive enough tools are still not available which enable the verification of such complex systems. As has been pointed out in (Rushby 1995), model checkers can be used to reason about control, but lack more generic expressive power, while theorem provers are good at dealing with rich data but lack the ability to reason easily about control. While some attempts have been made to combine both techniques (see, for example, Rajan et al. 1995, Owre et al. 1997), further work is needed in this area.

Even if powerful enough tools were available, it turns out that it is still hard to derive and prove the relevant properties from the system specification. On the one hand, general purpose specifications might not have all the information required for a specific property. On the other hand, a general purpose specification might have too much unnecessary detail (so far as a specific property is concerned) making the proof more difficult, if not impossible.

At a different level, it is also true that the style of specification drastically influences what and how analysis can be done. Consider, for instance, model checking. If the specification is not reducible to a finite state machine, model checking simply cannot be used on it. Although work is being done that uses model checking for non-finite state machines (see Mezzanotte & Paternò 1996, for an example in the context of

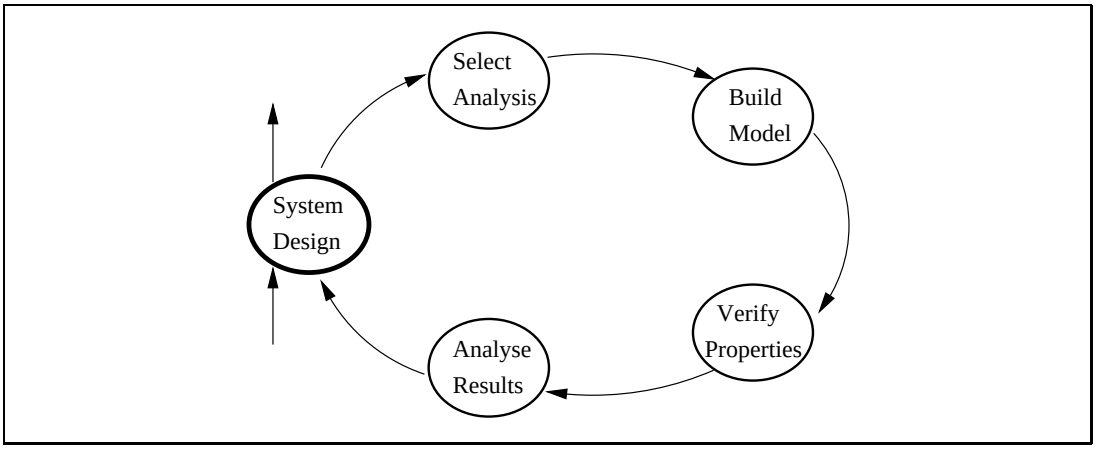

Figure 3.4: A new verification process

interactive systems) in the end it always involves some degree of uncertainty. If a finite state machine model is used, then the use of theorem proving is hardly justifiable. Generally, model checkers are better at analysing finite state machines. In the case of theorem provers, the situation can become even more complex. The fact that a theorem prover can be used to verify a specification, does not mean necessarily that it will be easy to do so. In fact, different styles of specification can have a drastic impact on how easy (feasible) it is to prove one thing or another, and different properties can require different styles of reasoning. All this means that the specification should be written with the type of intended proof in mind.

3.4 A new verification process

It is now clear that a shift in focus is needed. Instead of focusing verification on the specification, it must be focused on the system itself and its properties. This way, design and verification become part of the same development process, where verification is used to inform design decisions. This type of approach is called *verify-while-develop* by de Roever (1998). It can be achieved by using, not a monolithic specification of the system which tries to encompass all of the system (at its level of abstraction), but a set of models (cf. Fields et al. 1997) each taking a particular viewpoint on the system. Four main steps can be identified for this type of approach (see Figure 3.4):

- selecting what to verify;

- building an appropriate model;
- performing the verification;
- analysing the results.

The next sections look at each of these steps.

3.4.1 Selection

An instance of the process starts with the selection of a particular feature or aspect of the system which is considered relevant enough to deserve a rigorous verification of its properties. At the same time, those properties which should be verified are also identified.

Section 3.2 has discussed the main concepts that come into play regarding the identification of desirable properties of interactive systems. As was then pointed out, how these concepts crystallise in concrete properties that a system should exhibit is dependent on the domain. Hence, only in the context of a concrete design situation can concrete verification objectives be formalised. The selection of these objectives can be triggered in a number of ways. Three main possibilities are:

- Specific requirements that the design must meet — in the case of, for instance, safety critical systems, a number of predefined critical properties might be part of the requirements specification. Formal verification can be used to certify that the requirements are met.
- The need to make a specific design decision — the designer might resort to formal analysis when faced with a choice between different design alternatives. In this way, designers can make better informed decisions, avoid carrying out a full field study, or having to wait until the prototyping stage in order to evaluate the alternatives.
- Design principles and guidelines — the design might be assessed against specific design principles and guidelines in order to ascertain its quality.

By moving the decision of what to verify inside the design process the concerns mentioned in Sections 3.3.2 and 3.3.3 are addressed. It becomes easier to identify what properties are truly relevant to the design, and also it becomes possible to use the results of verification to inform design decision early in development.

3.4.2 Modelling

Having identified the subject of verification, and what properties need to be analysed, the next step is building a model of the selected artifact(s) which enables the desired analysis to be carried out. A first consideration here is what type of analysis will be needed. As discussed in Section 3.3.4, this will have influence on what type of model should be built. Since it is unlikely that properties will be either purely state related, or purely behaviour related, some compromise will have to be made at this stage.

An important point regarding the model has to do with its completeness. The model should only include information which is relevant to the property being analysed. This can pose a problem: exactly what is relevant regarding the property might not be known until the analysis is actually performed. In particular, how can it be known that something will not have an unexpected interference/effect regarding the property under consideration? The solution to this problem lies in the assumption-commitment paradigm (see de Roever 1998). When the model is built, assumptions will have to be made about what is relevant and what is not. These assumptions can themselves be seen as properties of the system that should be verified. Hence, they can be the subject of analysis in order to prove their validity. This also shows how this type of approach potentiates a compositional style of verification (de Roever et al. 1998), a feature which is important when dealing with complex systems.

Obviously, the possibility of formal analysis is just one of the benefits of formal modelling. Another, is the immediate insight which is gained by the modelling process itself. This can give immediate feedback regarding the design, and can trigger discussion regarding specific design issues even before the analysis is actually started.

3.4.3 Verification

Having built a model, the appropriate tool can now be applied to its verification. Firstly, however, the properties that are to be verified must be expressed in a logic appropriate to the tool being used. Care must be taken regarding this formalisation. This is specially true of properties involving Perception or Execution, since the user side needs to be considered. As discussed in Chapter 2 (Section 2.5), a software engineering view of development is being taken so no user model is explicitly built. Instead, user considerations are moved to the stage of selecting what to analyse. This has an impact on what properties will be selected for analysis, but also on how those properties, which will include (*soft*) psychology oriented concerns, might be formally expressed in the context of a (*hard*) formal language. As Chapters 4 to 6 will show, this can be achieved by somewhat sidestepping the problem. Assumptions about the user can be made. These might be assumptions about the user's perceptual, cognitive, or motor capabilities. These assumptions can then be used to simplify the expression of the properties in a formal setting. Any proof of a property that was expressed thus, will only be valid if the assumptions are valid. Since these are assumptions about the user, deciding whether, or not, it is reasonable to assume they hold becomes a matter for discussion with psychologists and other human-factors experts.

What is being done is breaking the reasoning into two stages: a more system oriented stage, and a more user oriented stage. The results of the user oriented analysis are used to inform the more formal system oriented analysis. This is another instance of an assumption-commitment style of reasoning, and helps solve the problem of folding user concerns into a software engineering context.

3.4.4 Analysis

Finally the results of the verification need to be analysed. From an analysis point of view, the most interesting situations are those where the verification fails. In model checking a counter example might be provided by the tool, showing a behaviour where the property does not hold. In theorem proving some sequent will be reached which cannot be solved. In both cases the feed back provided by the tools gives hints as to what is wrong with the model.

Again, care must be exercised when analysing the results. In particular, two question must be asked:

- Is the problem that was found an authentic problem of the system, or does it relate only to the way in which the model was developed? It might even be that the model does not include all the relevant information. Chapter 5 will give an example of this.
- How real is the problem? It might happen that the problem which was detected, although theoretically possible, has no real meaning in practice. Again, this prompts for further discussion with human-factors experts and further analysis of the system.

Once a conclusion is reached, the results of the analysis can be fed back to inform the design. Obviously, this process will be used iteratively.

3.4.5 Benefits of the approach

As the discussion above has shown, the verification loop in Figure 3.4 is not a self contained process. In selecting what to analyse, the designer might resort to design guidelines and principles, or to specific requirements of the system. The process of building a model and formalising the properties, besides providing immediate insight into the design, will also raise issues related, for instance, to assumptions about how the system will be used or the perceptual capabilities the users might need. This will prompt discussion with human-factors experts, in order to assess the validity of those assumptions. Finally, the analysis of the results will prompt further interactions with psychologists, in order to fully understand the consequences of the results. Figure 3.5 shows a new version of the process where all these interconnections are made explicit. As can be seen, the analysis stage becomes the focal point of the process.

The iterative process just described has a number of advantages, when compared with the use of verification late in the design life-cycle:

- immediate use of results — since verification is used during design, not after a complete specification has been developed, the incorporation of results from

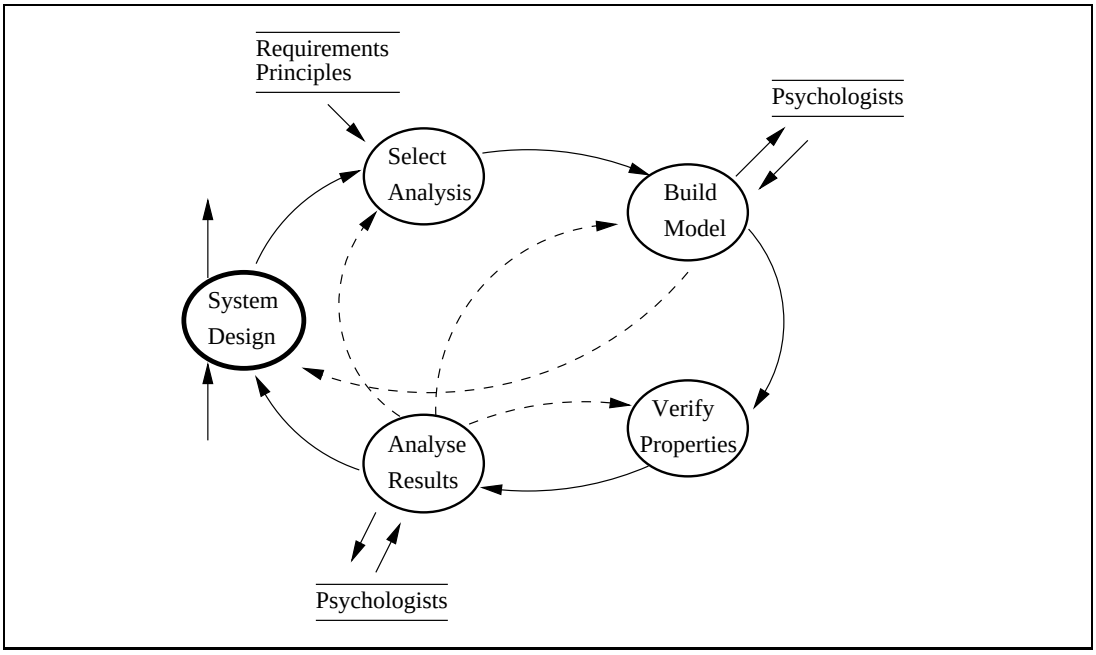

Figure 3.5: Revised verification process

the verification process into the design becomes much easier (the results become available when decisions have to be made);

- relevance of the analysis — since the process focuses on system features and properties, it uses verification to validate the choices that are made in relation to what is important of the system, not its specification;

- technique fit — since a number of models are used, not just a single specification, the process allows the selection of the most appropriate verification technique to each situation;

- model fit — each model can be developed in the most suitable formalism for the tool that will be used;

- complexity control — since the models will focus on specific system features and properties, they will be simpler and easier to manage and verify, enabling the verification of properties that otherwise would be too difficult to check;

- reuse — because of the focus on specific features/properties, it will be possible to reuse proofs/property formalisations when the same property is being verified

in different models (which might correspond to different design alternatives for the same system, or to different systems), or even reuse models when similar features are being analysed of different systems;

- compositionality — the use of an assumption-commitment style of reasoning allows for the use of compositionality in the proof process, this helps in further controlling complexity in the proofs, and also in folding user concerns into verification.

As pointed out in (Fields et al. 1997), the use of a number of models to represent different aspects of the same reality does raise some problems. In particular, problems with consistency between models, and with the veracity of the models. As the authors point out these issues are not necessarily disadvantages or even specific to this type of approach, and, in the end, these concerns have more to do with the designer than with the tool. The veracity of a model is a problem which is formally (and philosophically) undecidable. Some criteria for veracity would have to be devised, but then the problem would remain of showing any given criterion correct.

Regarding consistency, each model can be seen as a fish-eyed image of the system: some aspects of the system are represented in great detail and the remainder in a more abstract manner. These models will overlap to some extent. Consistency is guaranteed by making sure that models agree where overlap occurs. As Chapter 5 will show, it is even possible to analyse consistency between different models, representing different perspectives on the system, in order to detect inconsistencies on the design itself. Chapter 6 illustrates the use of different models to analyse different aspects of a system.

3.5 The Interactor language

Having identified the main goal of verification, and how it should be applied, the problem now remains of how to build the models. Traditional specification languages are usually concerned with the state and operations of the system, and user interface

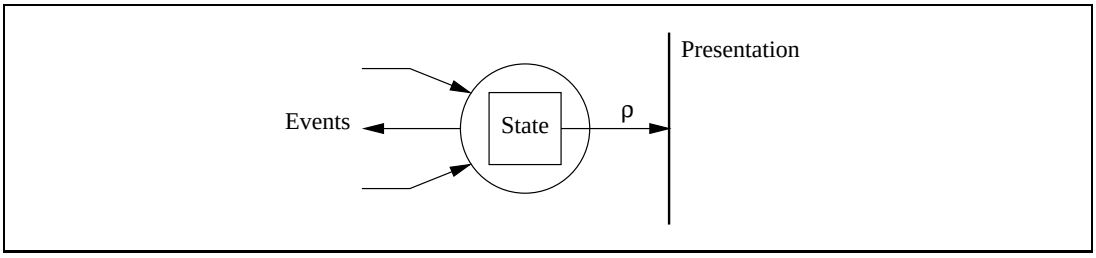

Figure 3.6: York Interactor

considerations are left to the implementation stage of the development process. Interactors (Faconti & Paternò 1990, Duke & Harrison 1993) have been proposed as a means of structuring specifications with the user interface in mind.

The interactor model, as developed by the York group (Duke & Harrison 1993, see Figure 3.6), is that of an object which, besides interacting with the environment through events, is capable of rendering (part of) its state into some presentation medium. The model does not prescribe a specific specification notation for the description of interactor state and behaviour. Instead, it acts as a mechanism for structuring the use of standard specification techniques in the context of interactive systems specification. Several different formalisms have been used to specify interactors. These include Z (Duke & Harrison 1993), modal action logic (MAL) (Duke, Barnard, May & Duce 1995), and VDM (Harrison et al. 1996).

In this thesis, interactors will be used to build the models. Structured MAL (Ryan et al. 1991, Fiadeiro & Maibaum 1991), a (deontic) modal logic, will be used to specify interactor behaviour, following its adaptation to interactor specification by Duke (see, for instance Duke, Barnard, May & Duce 1995, Duke et al. 1999).

The interactor language to be used will be introduced by means of an example. The photocopier example introduced in Chapter 2 (see Figure 2.1 on page 44) will be used for this. For brevity, not all actions in the original specification will be modelled. Initially, only two actions will be considered: *copy* and *finishedcopy*. Other features of the original specification will be occasionally considered to illustrate specific points.

Looking at Figure 3.6, it can be seen that an interactor has three main components:

- state;

- behaviour (the effect of the events in the state);
- presentation (ρ).

The definition of an interactor starts with a declaration containing the keyword **interactor** and an identifier, the interactor name. Supposing the interactor should be called *Copier*, the declaration is:

interactor *Copier*

Next, the interactor state must be defined.

3.5.1 State

The state of an interactor is defined by a collection of attributes. In this case, of the three original attributes in Figure 2.1, only attribute "Copying" is needed. The other attributes have no effect, and are not affected, by the actions being considered. Attributes are declared in the **attributes** clause, together with their types. The interactor becomes:

interactor *Copier*
attributes
  copying : *bool*

The usual types will be considered as given: boolean (*bool*), natural numbers (*nat*), etc. In this case, type *bool* is being used. It is also possible to define new types, by enumeration, in a global clause: **types**. Supposing the toner control was also to be modelled, a more general attribute than the original "Normal Toner" could be modelled using the enumerated type:

types
  $TonerIntensity = \{light, normal, dark\}$

3.5.2 Behaviour

Once the state is defined, the actions that correspond to the events must be declared. This is done using clause **actions**. In this case there are two actions:

interactor *Copier*
attributes
copying : *bool*
actions
copy finishedCopy

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

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

A major difference between the logic used herein and Structured MAL is the treatment of the modal operator. There the modal operator is applied to whole propositions. This means that there is no way to relate *old* and *new* values of attributes directly. In Structured MAL this is usually done by the introduction of auxiliary variables. Suppose that an attribute was being considered (*ncopies*) to indicate the number of copies that should be made. Suppose also an action (*incr*) which increments the value of that attribute. In Structured MAL the effect of the action on the attribute would be specified using the axiom:

$$(Aux = ncopies) \rightarrow [incr](ncopies = Aux + 1) \tag{3.10}$$

where Aux is an auxiliary variable introduced to *carry* the value of *ncopies* into the next state (after *incr*). This makes axioms harder to read:

- an auxiliary variable must be introduced for each attribute which is defined in terms of its past value;

- what is calculated in relation to what might become ambiguous — in the expression $(ncopies = Aux + 1)$, *ncopies*, being an attribute, is affected by the modal operator while Aux, being a variable, is not affected.

To avoid these auxiliary variables, the definition of modal operator from (Fiadeiro & Maibaum 1991) will be used. The operator is only applied to attributes. This allows for the axiom to be written as:

$$([incr]ncopies) = ncopies + 1 \tag{3.11}$$

Which reads: the value of *ncopies* after *incr* equals the current value of *ncopies* plus one. To further simplify more complex axioms where the modal operator is applied to more than one attribute, the operator can be factored out and priming used to indicate those attributes which are affected by it. Hence, the axiom above becomes:

$$[incr](ncopies' = ncopies + 1) \tag{3.12}$$

The parentheses will be omitted whenever the scope of the modal operator can be inferred.

The effect of the two interactor actions in the state can now be expressed by the following axioms clause:

axioms
(1) $[copy]\ copying'$
(2) $[finishedCopy] \neg\ copying'$

These two axioms express the effect of the actions on the state, however, they say nothing about when the actions are permitted or required to happen. For this the permission and obligation operators must be used.

As in (Ryan et al. 1991), only the assertion of permissions and the denial of obligations will be considered. That is, axioms of the form:

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

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

As can be seen in Figure 2.1, the machine can only start copying if it is not already copying, and it must be copying in order for *finishCopy* to be allowed. These two conditions can be expressed using the axioms:

(3) $per(copy) \rightarrow \neg\ copying$
(4) $per(finishedCopy) \rightarrow copying$

Finally, an axiom can be written to express the fact that once the machine starts copying, it will eventually finish copying:

(5) $copying \rightarrow obl(finishedCopy)$

3.5.3 Presentation

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

The final version of the interactor is shown in Figure 3.7. In this case only the annotation $\boxed{\text{vis}}$ is being used. This annotation asserts the fact that the parameter/action is visible. Note that action *copy* is annotated with a modality, while action *finishedCopy* is not. This indicates that the first action can be invoked by the user, while the second is internal to the system.

3.5.4 Multiple Interactors

The model in Figure 3.7 has only one interactor. Usually a model will comprise of a number of interactors, each modelling a different component of the system. It is

interactor *Copier*
attributes
  vis *copying* : *bool*
actions
  vis *copy*
    finishedCopy
axioms
  (1) [*copy*] *copying'*
  (2) [*finishedCopy*] ¬ *copying'*
  (3) *per*(*copy*) → ¬ *copying*
  (4) *per*(*finishedCopy*) → *copying*
  (5) *copying* → *obl*(*finishedCopy*)

Figure 3.7: Simplified interactor version of the photocopier

possible, for example, to have one interactor modelling the functional core of the system, and another modelling the user interface. Complex interactors can be built from simpler ones either by composition or by inheritance.

Composition

The composition of interactors is done via inclusion as in (Ryan et al. 1991). It will be assumed that all actions and attributes of an interactor are always accessible to other interactors that include it. The use of interactor *Copier* in some other interactor is expressed thus:

interactor *Office*
includes
  Copier **via** *officeCopier*
...
axioms
  (1) [] ¬ *officeCopier.copying*

where *officeCopier* becomes the name of a particular instance of *Copier* in the context of interactor *Office*.

As the example shows a parent can access its children state with the notation: *child.attribute*. The example also illustrates the use of the initialisation operator. In this case Axiom (1) states that initially the copier will not be copying.

Inheritance

By importing one interactor into another, it is possible to use a limited form of inheritance. By doing this, all the declarations of the imported interactor become declarations of the interactor doing the importing. Consider that a model of a copier that included the *ncopies* attribute mentioned above is needed. This new model could be obtained from interactor *Copier* by writing:

interactor *newCopier*
importing *Copier*[*working*/*copying*]
attributes
  vis *ncopies* : nat_0
axioms
  (1) [*copy*] $ncopies' = ncopies - 1$
  (2) $per(copy) \rightarrow ncopies > 0$
  (3) [*finishedCopy*] $ncopies' = 1$
  (4) $per(finishedCopy) \rightarrow ncopies = 0$

Interactor *newCopier* has all the declarations of interactor *Copier*, plus the attribute *ncopies*. Additionally, attribute *copying* has been renamed *working*. Obviously, new actions and axioms will be needed, the example show four axioms. Axioms (1) and (2) assert that action *copy* does not change the value of *ncopies*, and is only permitted when *ncopies* is greater than zero. Axioms (3) and (4) assert that action *finishedCopy* resets *ncopies* to 1, and is only permitted when *ncopies* is zero.

A last feature of the language is parameterisation. So far all actions used were atomic. Actions can also have parameters to allow information to be passed with them. Consider interactor *newCopier* above, if an action to set the value of *ncopies* was needed, the interactor could be rewritten to:

interactor *newCopier*
importing *Copier*[*working*/*copying*]
attributes
  vis *ncopies* : nat_0
actions
  vis $reqncopies(nat_0)$
axioms
  (1) [*copy*] $ncopies' = ncopies - 1$
  (2) $per(copy) \rightarrow ncopies > 0$
  (3) [*finishedCopy*] $ncopies' = 1$
  (4) $per(finishedCopy) \rightarrow ncopies = 0$
  (5) $[reqncopies(n)]\ ncopies' = n$

This new version of the interactor shows how parameterised actions are declared: indicating the type(s) of the parameter(s), between parentheses, after the action name; and also how they are used: in this case n is a universally quantified variable representing the value which is passed when the action happens.

Interactors can also be parameterised. This allows for generic interactors to be written which can later be instantiated with appropriate types. Interactor parameterisation is done as it would be expected: a parameter is added, between parenthesis, to the name of the interactor being declared. Inside the interactor this parameter is treated as a type. Later, when that interactor is included or imported, a actual type must be provided. See Chapter 4 for examples of the use of this feature.

3.6 Conclusions

This chapter has argued that using verification in (interactive) systems development is more than just checking whether the specification of the system has all the required properties. Establishing a list of checkable properties is not possible, unless a very specific type of system is defined as the subject of interest. It is still the case, however, that interactive systems have specific considerations that need to be taken into account when deciding what should be subject to analysis. This chapter has put forward a framework for reasoning about interactive systems that embodies these concerns. It has done this by identifying some of the key concepts that must be taken into account.

Global specifications tend to be too complex for verification, and different types of properties ask for different proof styles/techniques (hence, different specification styles). The chapter proposed that a number of partial models/specifications of the system should be built, allowing for the most appropriate verification technique to be used in each case.

Changing the focus from a global specification into partial, property oriented, specifications can also give a number of additional advantages: greater confidence in that properties which are relevant to the system are being checked (not only of its specification); the specifications to verify become simpler; proofs may be reused during

development, or on systems with similar requirements. Furthermore, the important properties of the system must be considered during design. The development and verification of partial models can then be used as an aid to decision making.

Finally the interactor language that will be used in the remainder of the thesis was also introduced. This language uses a modal logic based on Structured MAL.

This chapter has set the scene for the remainder of the thesis. A model for user interface related properties, and a closer integration between verification and design were proposed. In order for this type of approach to be feasible, different verification techniques will be needed to tackle the different types of properties/models. Chapters 4 and 5 address the issues related to using model checking and theorem proving during interactive system development in the context of the proposed verification process.

Chapter 4

Model Checking

This chapter discusses the use of model checking for the automated analysis of interactive systems models. SMV is used to perform the analysis. A tool is described which enables the translation of interactor models into SMV code for verification. Two examples of application are also presented. The first example deals with perception related issues. The second illustrates how reasoning about mode complexity can be performed using model checking.

Some of the material presented in this chapter has previously appeared in (Campos & Harrison 1998, Campos & Harrison 1999*a*).

4.1 Introduction

The previous chapter has proposed a general approach to the use of verification during interactive systems development. The next step is to choose appropriate logics and verification tools, and assess their applicability in the context of the proposed approach. At this stage generic tools are favored to more specific ones.

As seen in Chapter 3, temporal logics can be useful at expressing behavioural properties of interactive systems. An advantage of using such logics is that model checking (Clarke et al. 1986) can be used to perform automated verification of temporal logic formulae. A prerequisite for model checking is that the system under analysis be modelled as a finite state machine. The temporal logic will express properties over the behaviour of such machines.

4.1.1 Choosing a logic

Labelled Transition Systems and Kripke Structures are the two basic types of structure which can be used when expressing finite state machines. Logics for Labelled Transition Systems include ACTL (Nicola & Vaandrager 1990), and are primarily concerned with the actions that the system can engage in. As seen in Section 3.2.3 this can pose problems when attempting to reason about the effect actions have on the state. Logics for Kripke Structures include CTL (Clarke et al. 1986), and allow the expression of properties related to the state structure. Unfortunately, Kripke structures have no notion of action, so direct reference to the cause of each state change is not possible.

An ideal situation would allow both actions, and state information to be referenced simultaneously. Two approaches can be taken (Nicola & Vaandrager 1990). One is to use an action based logic like ACTL and encode information about relevant state changes as special actions in the system. This is the approach taken in (Paternò 1995) to reference task completion for example. This solution, however, is primarily a manual one, requiring the skill of the developer. The relevant state changes need to be identified and included in the model each time a model is developed. Additionally, this solution can create a proliferation of actions, making it unclear which actions belong to the system, and which actions are referents for the states.

A second approach is to use a state-based logic like CTL and encode information about the transitions in the state. Each state will hold information about the transition(s) leaving, or, alternatively, arriving at that state. This is the approach taken in (Abowd et al. 1995). This type of approach lends itself to automation, since once an encoding strategy is chosen, an algorithmic solution can be found. Abowd et al.'s (1995) approach, however, deals only with simple Propositional Production System (PPS) based models. An interesting possibility, then, is to see if this type of approach can be extended to more complex interactor based models, and how it integrates into the verification process described in the previous chapter.

4.1.2 Choosing a tool

Several different tools exist for the automated analysis of finite state models. These range from state reachability tools, that can determine whether a specific state can be reached, to proper model checkers which can analyse the model with respect to more general temporal properties. While some tools require the explicit construction of the state space, others can act on symbolic representations of that state space, thereby saving space and time. SMV (McMillan 1993) is a symbolic model checker (SMV stands for Symbolic Model Verifier), and it will be used in this thesis.

SMV was chosen for three reasons. Firstly because it is a proper model checker. Some tools for reachability analysis might provide better capabilities for specific problems, but SMV is a generic tool for temporal property checking. Secondly because it uses CTL as the temporal logic for expressing properties. As explained in the previous section, this was the logic of choice. Finally, because of its symbolic algorithm. SMV uses Binary Decision Diagrams (BDD — Bryant 1986) to perform symbolic manipulation of the state space (Burch et al. 1990). This allows for the verification of larger state spaces than those allowed by non-symbolic tools. The model checker included with the PVS theorem prover, for example, is not symbolic (see Rajan et al. 1995, Owre et al. 1997). The specific version of SMV used is version 2.5.3.1c from CMU[1].

4.2 From Interactors to SMV

For model checking of interactor models in SMV to be possible, first the models must be expressed in the SMV specification language. Unlike Abowd et al.'s (1995) approach, where only a single PPS is used, this implies representing models composed of multiple interactors. For each interactor, its state and behaviour must be expressed in SMV.

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

[1] `http://www.cs.cmu.edu/~modelcheck` (last accessed on the 17th of August, 1999).

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

Figure 4.1: An example SMV module

4.2.1 Overview of the approach

An SMV specification is a collection of modules, much in the same way as interactor specifications are collections of interactors. An SMV module is similar to an interactor in that is also has a state (a collection of attributes), and axioms describing how the state evolves. These similarities raise the possibility of representing interactors as SMV modules.

The interactor language has already been introduced in Chapter 3. Figure 4.1 shows an example SMV module. Attributes are declared in clause `VAR`, clause `INIT` defines the initial state of the module, and clause `TRANS` defines how the state evolves. In this case the attribute will repeatedly toggle between true and false (zero representing false), every time a state change happens. Note the use of `next` to reference the next state: axioms in `TRANS` clauses are written using a temporal logic where `next` is the only temporal operator. The usual propositional operators are also present: `!` stands for logical not, `&`/`|` stand for logical and/or respectively, `->` and `<->` stand for implication and equivalence. Please note that only a subset of the SMV input language will be used/needed in this thesis. Section 4.2.2 further addresses this.

An immediate observation from the description above is the lack of the notion of action in SMV. As described in (Fiadeiro & Maibaum 1991) or (Abowd et al. 1995), this can be solved by encoding information about transitions in the state. Instead of taking Abowd et al.'s (1995) approach of encoding information regarding the next action, it was decided to encode information regarding the previous action (the action corresponding to the last transition that occurred). This avoids the problem of duplicated initial states described in Chapter 2 (see Figure 2.2 in page 46). In this case states are duplicated when they can be reached using different actions (see Figure 4.2).

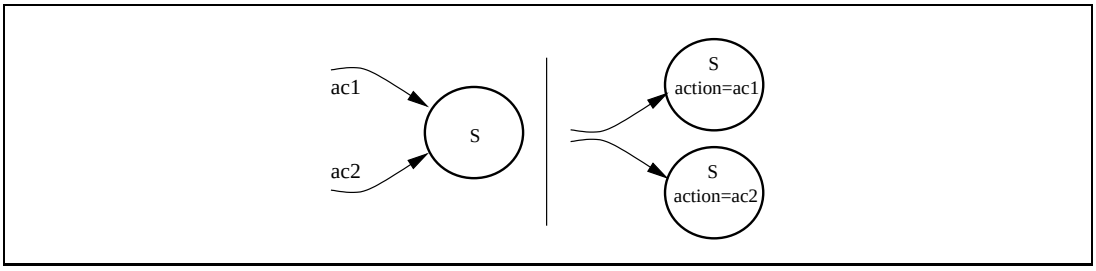

Figure 4.2: State duplication

State attributes in SMV can be declared as booleans or as having an enumerated type. This means that restrictions will have to be enforced in the types used in interactors. If, for example, a variable is defined as having type *nat*, its type will have to be restricted to an appropriate subrange of *nat* before translation to SMV is carried out. Additionally, SMV variables can be declared as arrays. Since is is not possible to use quantification, it is not possible to reason about whole arrays in the axioms, only individual values. Hence, arrays can be seen as collections of individual variables. Arrays will not be considered at this level (see Section 4.2.4).

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

So, with appropriate restrictions it is possible to represent the state of an interactor in the state of an SMV module, and to represent transitions as a special attribute of the state in a style similar to Abowd et al.'s (1995). The major problem remains

of expressing the behaviour of the interactors in SMV. With PPS models this was relatively easy since they define a transition relation on pairs of states, which can be directly expressed in SMV once the problem of encoding actions is solved. The logic used to express interactor behaviour, however, is semantically richer and poses some additional problems. This is specially true of the deontic axioms. The remainder of this section deals mainly with showing how a translation from MAL to SMV can be deduced. This fulfils two objectives: it provides the translation algorithm needed for each type of axiom, and it guarantees the correctness of the translation process. The approach taken follows from (Fiadeiro & Maibaum 1991).

4.2.2 SMV modules

In order to express MAL axioms in SMV, it is only necessary to consider a subset of the SMV input language. The ASSIGN declaration and case expressions, in particular, will not be used. As in (Abowd et al. 1995) case expressions would be of little use since they always select the first condition that evaluates to true.

Figure 4.1 above gives an example of a very simple SMV module. The complete list of declarations used is now presented:

- `VAR` — allows the declaration of the variables that define the module's state. The types associated with the attributes can be either boolean, an enumeration, an array, or another module. The use of arrays will be addressed in Section 4.2.4.

- `INVAR` — allows the specification of invariants over the state. These invariants must be evaluated *inside the state*, so they are written using propositional formulae on the module's attributes (the use of `next` is not allowed).

- `INIT` — allows the definition of the initial state of the module. As above, this is done using propositional formulae.

- `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.

- `FAIRNESS` — allows the specification of fairness constraints. The behaviour of the module will have to obey the fairness constraints infinitely often. Fairness constraints can be temporal formulae over paths (CTL formulae), or simply propositional (*inside the state*) formulae. In practice, only propositional formulae tend to be useful/meaningful. Hence, only those will be considered.
- `SPEC` — allows the definition of a CTL formula to be checked.

4.2.3 Expressing interactors in SMV

The above section has briefly described the syntax of SMV modules. The interactor language was introduced in Section 3.5. This section discusses how interactors can be expressed using these SMV modules. As explained in Section 4.2.1, importing and parameters will not be considered at this level. See Section 4.2.6 for an example of how these features can be eliminated from a model.

Attributes

In the previous section, it was shown that, given appropriate type restrictions, interactor attributes can be modelled by the variables of an SMV module. Hence, it is possible to define a translation rule:

attributes	a_1 : `T`	$\rightsquigarrow$	`VAR a1: T;`

where `T` is a valid SMV type.

Interactor inclusion

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, so translation is straightforward. Instances of included interactors become module variables whose types are the appropriate SMV modules resulting from the translation of the interactors. The translation rule for interactor inclusion is:

includes	i_{name} **via** i_1	$\rightsquigarrow$	`VAR i1: iname;`

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

Axioms

The next stage is to see how interactor axioms can be modelled in SMV. From Section 3.5, five types of interactor axioms can be identified:

- invariants — these are formulae that do not involve any kind of event (i.e., simple propositional formulae). They must hold for all states of the interactor.
- initialisation axioms — these are formulae that involve the reference event ([]). They define the initial state of the interactor.
- modal axioms — these are formulae involving the modal operator. They define the effect of actions in the state of the interactor.
- permission axioms — these are deontic formulae involving the use of *per*. They take the shape $per(ac) \rightarrow cond$: action ac is permitted only when the propositional formula $cond$ holds.
- obligation axioms — these are deontic formulae involving the use of *obl*. They take the shape $cond \rightarrow obl(ac)$: action ac becomes obligatory when the propositional formula $cond$ holds.

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

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

$prop(a_1, .., a_n)$	$\rightsquigarrow$	`INVAR prop(a1,..,an)`

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

$[]prop(a_1, .., a_n)$	$\rightsquigarrow$	`INIT prop(a1,..,an)`

Modal axioms These are axioms that take the shape $p([e]a_1, .., [e]a_g, a_h, .., a_n)$[2]. They specify the effect of actions in the state.

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

First the occurrence operator $>_{action}$ must be introduced. This corresponds to the $>_{\mathrm{T}}$ operator in (Fiadeiro & Maibaum 1991). This operator is used to signal, in a state, that the state has been reached by the occurrence of some specific action. Hence, $>_{action}\ e$ holds in a state when e is the action that causes the transition to that state (cf. Abowd et al. 1995).

Using the occurrence operator it is now possible to relate the two logics. From (Fiadeiro & Maibaum 1991) it is known that[3]:

$$(\rightarrow [e]p) \Rightarrow ((>_{action}\ e) \rightarrow p) \tag{4.1}$$

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

In the equation above, the modal operator is being applied to whole propositions. However, as explained in Section 3.5, in this thesis the modal operator works at the

[2]To simplify the presentation, the shorthand notation introduced in Section 3.5 (priming) is not used.

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

level of the attributes. Hence, Equation 4.1 becomes:

$$(([e]a_1) = a_2) \Rightarrow ((>_{action}\ e) \rightarrow (a_1 = Y(a_2))) \tag{4.2}$$

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

Since in SMV the Y operator does not exist, equation 4.2 must be rewritten using the `next` operator:

$$(([e]a_1) = a_2) \Rightarrow (\mathtt{next}(>_{action}\ e) \rightarrow (\mathtt{next}(a_1) = a_2)) \tag{4.3}$$

To write this equation in SMV it is first necessary to model the occurrence operator. The approach taken by (Abowd et al. 1995) will be followed, and the operator will be modeled by a state attribute ($>_{\mathtt{action}}$) indicating the event for which the operator holds true. Hence, $>_{action}\ e$ becomes $>_{\mathtt{action}}= \mathtt{e}$. The type of this attribute will be an enumeration of all the possible events.

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

$prop([e]a_1, .., [e]a_g, a_h, .., a_n) \rightsquigarrow$ `TRANS next(`$>_{\mathtt{action}}$`)=e ->`
`prop(next(`a_1`),..,next(`a_g`),`a_h`,..,`a_n`)`

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

This translation rule guarantees that all behaviour specified by modal axioms will be present in the SMV model. The opposite is not true (note that Equation 4.3 is an implication, not an equivalence). SMV generates all state transitions that do not infringe the given axioms. Hence, it is possible to have some behaviours that are present, not because they were explicitly specified, but because no axioms were given which state how the system behaves in some particular circumstances. This is actually a useful feature, since it allows for underspecified models.

Permission axioms These are axioms that take the shape $per(e) \rightarrow p(a_1, .., a_n)$. As for modal operators, in SMV there is no notion of permission. However, from (Fiadeiro & Maibaum 1991), it can be deduced that formulae of the form

$$per(e) \rightarrow cond$$

lead to

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

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

$per(e) \rightarrow p(a_1, .., a_n) \ \rightsquigarrow$ `TRANS next(>`$_{\texttt{action}}$`)=e -> p(`$\texttt{a}_1$`,..,`$\texttt{a}_n$`)`

The behaviour of the SMV model is guaranteed not to infringe the SMV axioms. Hence, this translation rule guarantees that all (MAL level) permission conditions, set for some action, will have to be met (in the corresponding SMV model) in order for the state transition associated with that action to take place.

Obligation axioms These are axioms that take the shape $p(a_1, .., a_n) \rightarrow obl(e)$. Once more there is no direct way to express this in SMV, but from (Fiadeiro & Maibaum 1991) it can be deduced that formulae of the form

$$cond \rightarrow obl(e)$$

lead to

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

with F the *sometime in the future* operator.

It is not possible to express this last equation directly as an SMV axiom. The SMV input language allows reference to the current and next state only, and the equation above makes reference to some arbitrary state in the future. The only way to influence future states of the system is by fairness conditions. Since a fairness condition needs to hold infinitely often, if the formula $\texttt{next}(>_{\texttt{action}}) = e$ is placed

as a fairness condition, then eventually the action will happen. This strategy would require formulae to be added to, and removed from, the set of fairness conditions as obligations are successively raised and fulfilled during the execution of the state machine. Unfortunately, fairness is defined by a static set of formulae in the SMV text. One way to get around this is to use a boolean flag signaling when a specific obligation is raised. Hence, instead of adding the formula to the set of fairness conditions, the flag is set to true. By only unsetting the flag when the action happens, and asserting, as a fairness condition, that the flag must infinitely often be false, it is guaranteed that the action will eventually happen once an obligation for it is raised.

It is now possible to enumerate the rules for the translation of an interactor into a SMV module. This is done in Table 4.1. On the left hand side of the table, the various interactor expressions are listed. The right hand side gives the corresponding SMV expressions The last rule gives the translation for obligation axioms, following the reasoning just described.

4.2.4 Some final comments regarding the translation

As stated initially, the discussion above has only considered actions with no parameters. As with parameterised interactors, it is possible to eliminate a parameterised action automatically by substituting it by a set of actions, one for each possible combination of the parameters' values. Parameterised actions can appear in three types of axioms (note that only universally quantified variables are accepted as parameters):

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

Another feature of the interactor language that has not been mentioned is type definition: the possibility of giving names to enumerated types. This is not possible in

Interactor	**SMV Module**
interactor name	MODULE name
attributes $a : \{v_1, .., v_n\}$	VAR a : $\{v_1, .., v_n\}$ VAR $<_{action}$: {ac1, .., acn};
interactor inclusion: **includes** i_{name} **via** i_1	VAR i1: iname;
invariants: $prop(a_1, .., a_n)$	INVAR prop($a_1, .., a_n$)
initialisation axioms: $prop([]a_1, .., []a_n)$	INIT prop($a_1, .., a_n$)
modal axioms: $prop([e]a_1, .., [e]a_g, a_h, .., a_n)$	TRANS next($<_{action}$) = e –> prop(next(a_1), .., next(a_g), a_h, .., a_n)
permission axioms: $per(e) \rightarrow prop(a_1, .., a_n)$	TRANS next($<_{action}$) = e –> prop($a_1, .., a_n$)
obligation axioms: $prop(a_1, .., a_n) \rightarrow obl(e)$	VAR oble : boolean; INIT !oble FAIRNESS !oble TRANS next($<_{action}$)!= e –> next(oble) = (prop($a_1, .., a_n$) | oble) TRANS next($<_{action}$) = e–>!next(oble)

Table 4.1: Translation from interactors to SMV

SMV but, once again, type names can be eliminated by substituting all the occurrences of a type name by its definition.

So far, the need to restrict the interactor language, so that it can be translated into the SMV language, has been highlighted. The opposite can also happen. SMV allows for the definition of arrays, so array types have been introduced into the interactor language. Arrays can be declared and used using the SMV syntax:

- arr : array $i..j$ of type — this declares an array variable with elements arr[i] through arr[j], each of type type (the indices must be integer values);
- arr[k] =x — the element arr[k] of the array is compared with the value x (k must be a constant).

Arrays can be multi-dimensional.

To help make some expressions more compact the following shorthand notation was introduced:

- `arr`[] $=x$ — all the elements in the array are compared with the value x;
- `arr`<`k`>$=x$ — all the elements in the array, except element `arr`[`k`], are compared with the value x (`k` must be a constant);
- **define** *name* = *expression* — this allows the naming of expressions, so that the names can be used instead of the full expressions in the axioms (see Figure 4.7);
- nochg(*attr*) — used in modal axioms to state that an attribute does not change value (more than one attribute can be listed in the same "nochg" expression).

It was also decided to allow the definition of fairness constraints in the interactor language. This is done using a special clause: **fairness**.

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

4.2.5 The tool

A tool to implement the translation just described has been implemented in Perl (see Wall et al. 1996). The Perl script (`i2smv`[4]) works by reading a interactor model, and

[4]The tool is available at: `http://www.cs.york.ac.uk/~jfc/thesis/i2smv.pl`

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. eliminate the shorthands mentioned above;
6. create the stuttering action;
7. generate SMV code according to the translation in Table 4.1.

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

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

would generate a file `model.smv` which could then be model checked using SMV.

The only restriction on the interactor language (besides the usual substitution of `>=` for $\geq$, etc. — `action` is used instead of $>_{\texttt{action}}$) stems from the fact that the compiler is line oriented. Hence, each expression must be fully contained in a single line. However, line breaks can be escaped with the backslash character, thus allowing, for example, multi-line axioms.

An Emacs[5] mode has been written to provide an integrated environment for the development, translation and verification of interactor specifications. Figure 4.3 shows the tool in use. The top pane holds an interactor model, the bottom pane shows the result of model checking the model with SMV. Once a model is written, the menu option i2smv on the menu bar, provides two alternatives:

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

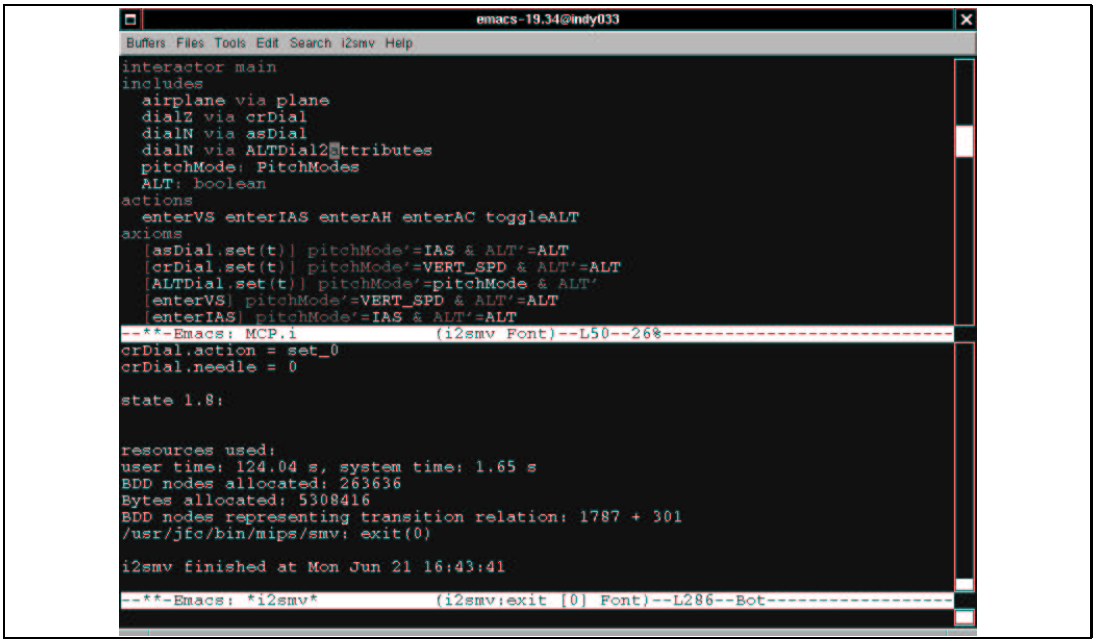

Figure 4.3: The Interactors to SMV compiler

- Compile — the model is compiled to SMV code and the generated file can then be opened in Emacs. SMV has its own mode, so Emacs will change from interactor mode to SMV mode.

- Compile & Verify — this results in what is shown in Figure 4.3. The model is compiled and SMV automatically used on the resulting code. A pane is created to show the result of the verification.

4.2.6 An illustrative translation example

In order to help understand the translation process just described, this section presents a small example of the translation of an interactor model to the corresponding SMV text. The interactor *newCopier* from Section 3.5 will be used. The interactor model is shown again in Figure 4.4. Note that a definition for type nat_0 has been included. Modality annotations have not been included since they are not treated at the SMV level. The name of interactor *newCopier* has been changed to *main*. As for SMV modules, the top level interactor will always have that name. This model covers most of the features in the interactor language: besides invariant, modal, and deontic axioms, it uses types, importing and parameterised actions.

types
$nat_0 = \{0, 1, 2\}$

interactor *Copier*
attributes
$copying : bool$
actions
$copy$ $finishedCopy$
axioms
$[copy]\ copying'$
$per(copy) \rightarrow \neg\ copying$
$[finishedCopy] \neg\ copying'$
$per(finishedCopy) \rightarrow copying$
$copying \rightarrow obl(finishedCopy)$

interactor *main*
importing
$Copier[working/copying]$
attributes
$ncopies : nat_0$
actions
$reqncopies(nat_0)$
axioms
$[]\neg\ working \wedge ncopies = 1$
$[copy]\ ncopies' = ncopies - 1$
$per(copy) \rightarrow ncopies > 0$
$[finishedCopy]\ ncopies' = 1$
$per(finishedCopy) \rightarrow ncopies = 0$
$[reqncopies(n)]\ ncopies' = n$

Figure 4.4: The *newCopier* interactor (revisited)

The steps enumerated in the previous section will now be applied to generate an SMV module. Note that the syntactical transformations which will be described are performed by the tool, not directly on the interactor model text, but on the internal representation used.

Step 1 – eliminate interactor importing

Since SMV modules have no notion of importing, importing must be eliminated from the model. As has been explained, this can be achieved by substituting importing declarations by the actual imported interactors. When this is done, interactor *main* becomes:

interactor *main*
attributes
$ncopies : nat_0$
$working : bool$
actions
$reqncopies(nat_0)$
$copy$ $finishedCopy$
axioms
$[]\neg\ working \wedge ncopies = 1$
$[copy]\ ncopies' = ncopies - 1$
$per(copy) \rightarrow ncopies > 0$
$[finishedCopy]\ ncopies' = 1$
$per(finishedCopy) \rightarrow ncopies = 0$
$[reqncopies(n)]\ ncopies' = n$
$[copy]\ working'$
$per(copy) \rightarrow \neg\ working$
$[finishedCopy]\ \neg\ working'$
$per(finishedCopy) \rightarrow working$
$working \rightarrow obl(finishedCopy)$

Note that *copying* has been substituted by *working*, as requested in the importing clause of interactor *main* (see Figure 4.4).

Step 2 – eliminate type parameters from interactors

Since the model being translated does not use parameterised interactors, this step does nothing. Suppose interactor *newCopier* was not the top level interactor, and had been declared as:

interactor $newCopier(T)$
...
attributes
$ncopies : T$
...

then, for each instantiation of *newCopier* with some type $type_x$, an interactor:

interactor $newCopier\_type_x$
...
attributes
$ncopies : type_x$
...

would be created, and the original parameterised declaration removed.

Step 3 – eliminate parameters from actions

In the current case it is necessary to eliminate the parameter from action *reqncopies*. As explained previously, this is done by creating as many actions as there are values in the type of the parameter (in this case nat_0). Clause **actions** of interactor *main* becomes:

actions
reqncopies_0 reqncopies_1 reqncopies_2
copy finishedCopy

Axioms involving *reqncopies* are also duplicated, with the parameter being replaced by the appropriate value in each instantiation of the axiom. In the current case, axiom $[reqncopies(n)]\ ncopies' = n$ is replaced by the following three axioms:

axioms
...
$[reqncopies\_0]\ ncopies' = 0$
$[reqncopies\_1]\ ncopies' = 1$
$[reqncopies\_2]\ ncopies' = 2$
...

Step 4 – eliminate type names

This amounts to simply substitute type names by their definitions. In this case only the **attributes** clause is affected:

attributes
$ncopies : \{0, 1, 2\}$
$working : bool$

Step 5 – eliminate the shorthand notation

In the current model none of the shorthand notation is used, hence the model remains the same.

Step 6 – create the stuttering action

This step simply introduces the stuttering action into the model. The final model before actual translation to SMV becomes:

interactor *main*
attributes
$ncopies : \{0, 1, 2\}$
$working : bool$
actions
reqncopies_0 reqncopies_1 reqncopies_2
copy finishedCopy
nil
axioms
$[]\neg\ working \wedge ncopies = 1$
$[copy]\ ncopies' = ncopies - 1$
$per(copy) \rightarrow ncopies > 0$
$[finishedCopy]\ ncopies' = 1$
$per(finishedCopy) \rightarrow ncopies = 0$
$[reqncopies\_0]\ ncopies' = 0$
$[reqncopies\_1]\ ncopies' = 1$
$[reqncopies\_2]\ ncopies' = 2$
$[copy]\ working'$
$per(copy) \rightarrow \neg\ working$
$[finishedCopy]\ \neg\ working'$
$per(finishedCopy) \rightarrow working$
$working \rightarrow obl(finishedCopy)$
$[nil]ncopies' = ncopies$
$[nil]working' = working$

Step 7 – generate SMV code

The model is now ready for translation to SMV. Some illustrative examples of the translation of axioms are (cf. Table 4.1):

- axiom $[]\neg\ working \wedge ncopies = 1$ becomes:

  ```
  INIT !working & ncopies=1;
  ```

- axiom $[copy]\ ncopies' = ncopies - 1$ becomes:

  ```
  TRANS next(action)=copy -> (next(ncopies)=ncopies - 1)
  ```

- axiom $per(copy) \rightarrow ncopies > 0$ becomes:

  ```
  TRANS next(action)=copy -> (ncopies>0)
  ```

- axiom $working \rightarrow obl(finishedCopy)$ becomes:

```
MODULE main
VAR
 action:  {copy, finishedCopy, reqncopies_0, reqncopies_1, reqncopies_2, nil};
 working:  boolean;
 oblfinishedCopy:  boolean;
 ncopies:  {0, 1, 2};
TRANS next(action)=nil -> ((next(working)=working))
TRANS next(action)=nil -> ((next(ncopies)=ncopies))
TRANS next(action)=copy -> (next(working))
TRANS next(action)=finishedCopy -> (!next(working))
TRANS next(action)=copy -> (!working)
TRANS next(action)=finishedCopy -> (working)
TRANS next(action)=finishedCopy -> !next(oblfinishedCopy)
TRANS next(action)!=finishedCopy ->
                   next(oblfinishedCopy)=(next(working) | oblfinishedCopy)
TRANS next(action)=copy -> (next(ncopies)=ncopies - 1)
TRANS next(action)=finishedCopy -> (next(ncopies)=1)
TRANS next(action)=reqncopies_0 -> (next(ncopies)=0)
TRANS next(action)=reqncopies_1 -> (next(ncopies)=1)
TRANS next(action)=reqncopies_2 -> (next(ncopies)=2)
TRANS next(action)=copy -> (ncopies>0)
TRANS next(action)=finishedCopy -> (ncopies=0)
INIT action=nil & !oblfinishedCopy
INIT !working & ncopies=1 & action=nil
FAIRNESS
 !oblfinishedCopy
```

Figure 4.5: The `newCopier` SMV module

```
VAR oblfinishedCopy: boolean;

INIT !oblfinishedCopy

FAIRNESS !oblfinishedCopy

TRANS next(action)!=finishedCopy ->

          next(oblfinishedCopy)=((working) | oblfinishedCopy)

TRANS next(action)=finishedCopy -> !next(oblfinishedCopy)
```

Figure 4.5 presents the output generated by the translation tool, when presented with the text in Figure 4.4 at the input.

4.3 An example — Perception

This section shows how the tool described above, together with SMV, can be used to reason about perception related issues. It must be stressed that the focus of the

discussion will be how automated reasoning (in this case model checking) can be used to reason about the interaction between users and interactive systems. The translation algorithm described in the previous section is assumed.

The example that will be used is that of an e-mail client. The basic requirements of the client are that:

- it should be able to receive mail messages;
- it should announce new mail to the user;
- it should allow the user to compose and send mail messages;
- it should work within a windowing environment.

For the purpose of this discussion, the analysis will focus on whether or not the user is made aware of new mail. Following the ideas put forward in Chapter 3, a fish-eyed view of the system will be used, and models will be built that concentrate on what is relevant for the issues at hand. Although the example will be kept very simple, it is still representative at two levels.

- it shows how focusing on a specific view of the system, results in models that are simple but nevertheless relate to features of the overall design;
- it also shows how SMV, in conjunction with the interactors compiler, can be used to analyse multiple interactor specifications, and discuss the relative merits of alternative design options.

Since it is considered that the e-mail client is to be used in a windowing environment, a way is needed to represent the windows on the screen. Because the analysis is only concerned with visibility issues, the models of windows will be built around the following information:

- whether the window is *mapped* on the screen — when a window is mapped it becomes present on the screen until it is unmapped;
- whether the window is *visible* on the screen — a window might be mapped but hidden by another window;

interactor *window*
attributes
vis *mapped* : *bool*
vis *visible* : *bool*
vis *newinfo* : *bool*
actions
map unmap hidew show update seen
axioms
(1) [] ($\neg$ *mapped* $\wedge$ $\neg$ *visible* $\wedge$ $\neg$ *newinfo*)
(2) *per*(*map*) $\rightarrow$ $\neg$ *mapped*
(3) [*map*] *mapped*$'$ $\wedge$ *visible*$'$ $\wedge$ *newinfo*$'$ = *newinfo*
(4) *per*(*unmap*) $\rightarrow$ *mapped*
(5) [*unmap*] $\neg$ *mapped*$'$ $\wedge$ $\neg$ *visible*$'$ $\wedge$ *newinfo*$'$ = *newinfo*
(6) *per*(*hidew*) $\rightarrow$ *mapped* $\wedge$ *visible*
(7) [*hidew*] *mapped*$'$ $\wedge$ $\neg$ *visible*$'$ $\wedge$ *newinfo*$'$ = *newinfo*
(8) *per*(*show*) $\rightarrow$ *mapped* $\wedge$ $\neg$ *visible*
(9) [*show*] *mapped*$'$ $\wedge$ *visible*$'$ $\wedge$ *newinfo*$'$ = *newinfo*
(10) [*update*] *mapped*$'$ = *mapped* $\wedge$ *newinfo*$'$
(11) *per*(*seen*) $\rightarrow$ *newinfo*
(12) [*seen*] *mapped*$'$ = *mapped* $\wedge$ $\neg$ *newinfo*$'$

Figure 4.6: Window interactor

- if the window has *new information* being displayed — a window is considered to have new information from the moment that it is *updated* until the information is *seen* by the user.

This set of attributes was chosen to reflect the concepts at stake in the analysis. For the analysis of other features of the system, the set of attributes to include in the specification will be different. In the context of interest, the interactors representing windows have the following attributes:

attributes
mapped : *boolean*
visible : *boolean*
newinfo : *boolean*

For each of these attributes, actions to set and unset them are introduced:

actions
map unmap hidew show update seen

The complete definition of the "window" interactor can be seen in Figure 4.6. Axiom 1 defines the initial state of the interactor, the remaining axioms define when actions are allowed and what their effect is on the state.

Two mechanisms for announcing new mail will be considered:

interactor *main*
includes
window **via** *mail*
window **via** *others*
define
$mail\_changevis = mail.>'_{\texttt{action}} \in \{map, unmap, show\}$
$others\_changevis = others.>'_{\texttt{action}} \in \{map, unmap, show\}$
axioms
(1) $(\neg\ (mail.mapped \wedge others.mapped)) \rightarrow$
$((mail.visible' = mail.visible \wedge others.visible' = others.visible)$
$\vee\ mail\_changevis \vee others\_changevis)$

Figure 4.7: Simple mail agent window

- announcing new mail in the mail agents own window;
- announcing new mail using a pop up window.

The analysis performed in each case will now be described.

Using a simple mail agent window

The first possibility considered is to put information regarding the new message(s) in the mail client window. This can be done in a number of different ways. For the current analysis, however, there is no need to know exactly how it is done, so the simple window interactor described above will be used.

In order to represent the interplay between the e-mail window and other windows on the screen, the specification considers two windows (see Figure 4.7):

- *mail* — the e-mail client window;
- *others* — represents all other possible windows in the screen.

The only axiom that is needed states that, unless both windows are mapped, visibility can only change explicitly. Once the specification is complete, the analysis can start.

As stated above, the analysis is performed with SMV, using the `i2smv` tool to perform the translation of the interactor model to the SMV input language. The code resulting from the interactor model above is presented in Appendix A.1. The first test is done just to get increased confidence in the soundness of the specification:

EF(mail.mapped & others.mapped)

It simply verifies if it is possible to have both windows mapped (thus verifying that the finite state machine is not empty - note that initially both windows are unmapped). The next property tries to see if after a new message arrives that information is available to the user:

AG(mail.action = update → mail.visible)

The answer to this test is no and the following counter example is provided[6]:

state 1.1 :
mail.newinfo = 0
mail.visible = 0
mail.mapped = 0
others.mapped = 0

state 1.2 :
mail.action = update
mail.newinfo = 1

In the initial state (State 1.1) no window is mapped (hence, visible) and the mail window has no new information. When a "update" happens, "mail" receives new information (State 1.2). However, the window remains unmapped and invisible. This means that a new message has arrived but the user has no way of knowing about this new message. This prompts consideration of the next mechanism that will be analysed: using a pop up window to announce new mail.

Using a pop-up window

Having identified a problem with the previous approach, a new design will be tried: introducing a pop up window to warn about new mail.

A pop up window is defined as a window which is mapped every time it is updated. Thus, the behaviour of the *update* action is redefined:

[6]Note that, at each state, only attributes that have changed are shown. Also, the counterexample was stripped of irrelevant attributes.

interactor *main*
includes
window **via** *mail*
window **via** *others*
popup **via** *alert*
define
$mail\_changevis = mail.\ >'_{\texttt{action}} \in \{map, unmap, show\}$
$others\_changevis = others.\ >'_{\texttt{action}} \in \{map, unmap, show\}$
$alert\_changevis = alert.\ >'_{\texttt{action}} \in \{map, unmap, show\}$
axioms
(1) $mail.>_{\texttt{action}} = update \leftrightarrow alert.>_{\texttt{action}} = update$
(2) $mail.>_{\texttt{action}} = seen \rightarrow \neg\ alert.newinfo$
(3) $\neg\ ((mail.mapped \wedge others.mapped)$
$\mid (mail.mapped \wedge alert.mapped)$
$\mid (others.mapped \wedge alert.mapped))$
$\rightarrow ((mail.visible' = mail.visible$
$\wedge\ others.visible' = others.visible$
$\wedge\ alert.visible' = alert.visible)$
$\vee\ mail\_changevis \vee others\_changevis \vee alert\_changevis)$

Figure 4.8: Pop-up window interactor

interactor *popup*
importing
$window[dummy/update]$
actions
update
axioms
(1) $\neg\ >_{\texttt{action}} = dummy$
(2) $[update]\ mapped' \wedge newinfo' \wedge visible'$

The main specification now includes three windows (see Figure 4.8):

- *mail* — the e-mail main window;
- *alert* — the pop up window;
- *others* — all other possible windows in the system.

Whenever a message is received, *mail* and *alert* alike must be updated. This is expressed by Axiom 1. Additionally, if the user resets the e-mail client, the alert window must also be reset (Axiom 2). Axiom 3 states that unless more than one window is mapped, the visibility of a window can only change explicitly. The SMV code for this new model is presented in Appendix A.2.

The previous tests can now be rerun in the new model. First a check that the three windows can be mapped:

$$\mathtt{EF}(\mathtt{mail.mapped}\ \&\ \mathtt{others.mapped}\ \&\ \mathtt{alert.mapped})$$

Once again this serves only to enhance confidence in the specification itself. The first real test is whether the pop up window becomes visible when a new message arrives:

$$\mathtt{AG}(\mathtt{mail.action} = \mathtt{update} \rightarrow \mathtt{alert.visible})$$

In this case the answer is yes. However, it is one thing for the alert window to become visible, another for the user to see it. To test that the window will always be visible until the user sees it, a test that it can only disappear by direct action of the user is performed:

$$\mathtt{AG}(\mathtt{mail.action} = \mathtt{update} \rightarrow \mathtt{A}[(\mathtt{alert.visible}\ \&\ \mathtt{alert.newinfo})\ \mathtt{U}\ (\mathtt{alert.action\ in}\ \{\mathtt{seen}, \mathtt{unmap}\})])$$

Here it is being assumed that a user will always look at the window before acting on it. The validity of the conclusions drawn from a hypothetical proof of the theorem would be dependent on the validity of the assumption. In this case, however, the answer is no, and the counter example shows that some other window might hide the alert window before the user gets a chance to see it.

From this analysis it can be concluded that this approach, although an improvement on the previous one, still does not guarantee that the arrival of new messages will be noticed by the user. A similar analysis could now be carried out for the case where an icon showing the state of the mail box is always present in the desktop, or for the case where the pop up window cannot be hidden by another window.

Discussion

In terms of the example, from the previous discussion it can be concluded that, in order to maximize user awareness regarding the arrival of new mail, some kind of permanent window displaying the status of the mail box must be used. This is not a terribly ground breaking or surprising result. In fact, it could have been reached by simply *thinking* about the problem in some sort of informal way. The hope is that the same kind of analysis applied to more interesting (non trivial) examples, might allow conclusions to be reached which are not so obvious.

Another aspect to consider when choosing what type of mechanism to use in order to announce new main, is whether a permanent icon will produce screen clutter. This analysis is outside the scope of the present specification (probably of the technique). Nevertheless, if some (probably) psychological study regarding desktop clutter can be carried out, its results may then be combined with the results of the present analysis to make a choice.

4.4 Another example — Mode Complexity

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 in a MD-88 aircraft. A change in the flying mode, performed by the autopilot without intervention of the pilot, caused pilot action to cancel the capture on the desired altitude inadvertently. This situation has implications for the safety of the aircraft as it can result in an "altitude bust" and consequent air traffic problems of loss of separation with other aircraft. This particular problem was identified during simulation of realistic flight missions, although (Palmer 1995) notes that similar problems are frequently reported to the Aviation Safety Reporting System (ASRS[7]).

This section will show how checking specifications using a model incorporating the interface between the pilot and the automation, may detect problems such as these in early stages of design.

4.4.1 Approach to verification

When using automated systems, operators build mental models of the system which lead to expectations about system behaviour. When the system behaves differently from the expectations of the operator so called *automation surprises* (Woods et al. 1994) happen. The interesting point about the present example is that the system behaved as designed (i.e., it did not malfunction) but nevertheless an automation

[7] `http://olias.arc.nasa.gov/ASRS/ASRS.html` (last accessed on the 20th of July, 1999).

surprise happened. This suggests that the system is misleading operators into forming false beliefs about its behaviour (i.e., wrong mental models).

While the use of simulation allows for the detection of shortcomings in design, it has some intrinsic problems. In order to perform a simulation an actual system or prototype has to be built, this means that simulations are costly and can only be done late in the design/development life cycle, when design decisions have already been committed to, and change is difficult. The ability to analyse and predict potential problems from the initial stages of design would reduce the number of problems found later in the simulation stage. One of the implications of doing this early analysis is that it has to be done without the benefit of hindsight (apart from what has been learnt from previous analysis and systems). The analysis is not trying to explain why something went wrong, instead it is being used, during design, to identify potential sources of problems. In this context, hindsight is not available. So, when developing a methodology for the integration of verification into design, care must be taken to avoid relying on it.

In keeping with the above principle, the system will not be modelled around the scenario presented by Palmer (1995). Instead a generic model of the artifact under consideration will be built, and then those aspects of the behaviour that are highlighted by the case study will be analysed. Hence, if the analysis is able to detect the problem, it will have been shown that it would be possible to have prevented that same problem from creeping unnoticed into the design of the aircraft.

Note that since the analysis is concerned with the interaction between the user and the artifact, the model will focus mainly on what is relevant in that dialogue. In particular it will not model in great detail the inner workings of the artifact, only the manifestations that are present at the interface. This process of abstraction is common in model checking. Of course a question might be raised as to whether the interface presentation accurately reflects the internal state of the system. This type of analysis (relating the state of the system to the state of the presentation) is best dealt with in a theorem proving context (see Chapter 5). This highlights one of the advantages of the approach: the possibility of using different verification techniques, depending on the particular type of task at hand.

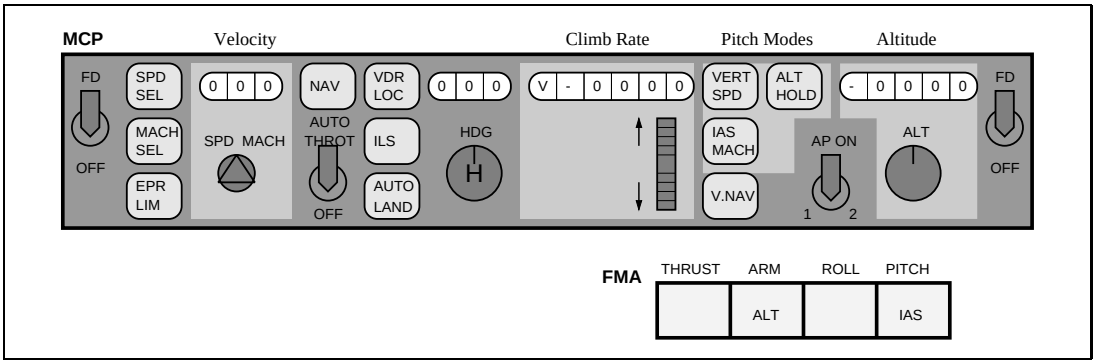

Figure 4.9: The Mode Control Panel (adapted from Honeywell Inc. 1988)

4.4.2 Selecting what to analyse

As seen in Figure 3.4 (page 94), the first step is deciding exactly what features of the systems to analyse. Identifying relevant requirements and properties to ensure correctness can be a nontrivial task. This is especially true of open systems, such as is the case with interactive systems, where the correctness of system behaviour can be verified only in the context of assumptions made about the environment (cf. the assumption-commitment paradigm de Roever 1998).

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

In the present case the analysis will be looking at how the automation and user interact during altitude acquisition. An expectation which is reasonable 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.

The property above relates to the vertical guidance subsystem of the aircraft mode logic. On the MD-88 the pilot interacts with it through a panel called the Mode Control Panel (MCP — see Figure 4.9).

The MCP enables the pilot to interact with the guidance subsystems of the aircraft. These include selection of altitude, heading, vertical velocity, and speed, and the associated navigation modes. The relevant functionality of the MCP will be described in section 4.4.3, together with the model that was built. Information regarding the current flying modes is displayed on the Flight Mode Annunciator (FMA). The model will include the relevant components of the FMA as attributes (*pitchMode* and *ALT*) of the MCP model (see Figure 4.12).

4.4.3 Modelling

Once some aspect of the system has been selected for analysis, the process continues with the development of an appropriate model.

Modelling the context as a finite system

As pointed out in Section 3.2.3, the most interesting issues arise when the system interacts, not only with the user, but also with its environment. So, in order to analyse a system it must be placed in its context of operation. The MCP will not be unsafe in itself, it only makes sense to talk of shortcomings in its design in relation to the actual system that the MCP is influencing.

In this case the aircraft state must modelled in order to be related to the automation state. Regarding the scenario that will be under consideration (altitude acquisition) the interesting information is its vertical speed, airspeed, and altitude.

Modelling this information poses an interesting problem. The aircraft is a continuous system, but interactor specifications are discrete. This means a way must be found to make the behaviour of the aircraft discrete in order for it to be expressed in the specification language. This can be done using abstractions. A state variable that ranges over a (potentially) non-finite state space, must be substituted by a corresponding (abstracted) state variable that ranges over a (small) finite domain (cf. Dwyer et al. 1997). By doing this the possible behaviours of the original system are restricted to a subset that can be modelled.

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

Figure 4.10: The aircraft

Care must be taken that the abstraction process captures the interesting features of the system. Imagine the state of the aircraft as a (continuous) flow over time, and place by its side a sequence of states representing interactor behaviour. The interactor states can then be matched to interesting moments in the behaviour of the aircraft. In the present context there is special interest in the altitude, so states will correspond to changes in the altitude by some amount (say 1 in some unit of measure). In order to make the state transitions possible the action *fly* is introduced. Besides asserting the change of altitude in each transition, the axioms relate climb rate to the altitude change. The model for the aircraft is shown in Figure 4.10.

Modelling the MCP

When modelling the MCP, the specific analysis that will be performed can be taken into account, in order to build a simpler model. In this case the analysis is related to 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. The components that were deemed relevant are shown in Figure 4.9 in a lighter background. The model will include setting velocity, climb rate, and altitude, and selecting the appropriate pitch mode (see below). The assumption is being made that the other components of the MCP (for example, lateral navigation, and thrust) will not affect the safety of operation of the pitch modes (cf.

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

Figure 4.11: Parameterised dial interactor

assumption-commitment paradigm). This assumption could then be discharged by a separate proof process. This enables, not only analysis about specific design decisions regarding particular aspects of the design, but also the verification of the safety of the overall design in a compositional manner.

From Figure 4.9 it can be seen that the model must consider three main dials:

- airspeed (velocity);
- vertical speed (climb rate);
- the altitude window (i.e., the altitude to which the aircraft should climb).

While airspeed, and altitude can only be positive values, the vertical speed can either be positive (going up) or negative (going down). Hence, a parameterised interactor is used to represent dials (see Figure 4.11).

It can be seen in Figure 4.9 that there is more than one type of dial on the MCP. At this level of abstraction, dials are represented by an attribute (*needle*) and an action (*set*). The action corresponds to setting a value (Axiom 1). The attribute represents the value that has been set.

How the MCP influences the automation will depend on its operating Pitch Mode. 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 automatically adjusted);
- *IAS* (indicated airspeed pitch mode): instructs the aircraft to maintain the airspeed indicated in the MCP (the climb rate will be automatically adjusted);

- *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).

Hence, the following type is defined:

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

Additionally, there is a capture switch (*ALT*) which, when armed, causes the aircraft to stop climbing when the altitude indicated in the MCP is reached.

The MCP is described by the interactor in Figure 4.12. Note that setting the airspeed or the climb rate causes the pitch mode to change accordingly (Axioms 1 and 2), and that setting the altitude dial arms the altitude capture (Axioms 3). These axioms specify mode changes that are implicitly carried out by the automation as a consequence of user activity. Axioms 4 to 8 are introduced to define the effect of the interactor's own actions, basically changing between different pitch modes and toggling the altitude capture. Note that action *enterAC* (setting the pitch mode to *ALT_CAP*) is an internal system event. Axioms 9 and 10 specify the mode logic that regulates when the event happens: the altitude capture must be armed, and the plane must be inside some neighbourhood of the target altitude. Regarding this abstract specification, the restriction on the value of this distance is that it should not be too small, in order to allow for the system to evolve while inside the neighbourhood of the target altitude. The value 2 was chosen. Similarly, Axiom 11 specifies that the system must set the pitch mode automatically to *ALT_HLD* once the desired altitude has been reached. And finally, Axioms 12 to 15 describe the effect of the pitch modes on the state of the aircraft. At this stage the types are left unspecified.

4.4.4 Checking the design

Having built a model the next step is to analyse it. Since the property under analysis relates to the behaviour of the model, model checking is the natural choice of the

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

Figure 4.12: The MCP model

technique to be used. Before this can be done, two further steps are necessary. First, a checkable version of the model must be obtained. Second, the properties of interest must be expressed in CTL.

Converting the Model

In order to check the specification some adjustments have to be made. The most relevant is the need to only have enumerated types in the specification. Regarding altitude and velocity this does not represent a problem. In fact, the aircraft will have its own physical limitations on maximum speed and altitude. Care must be taken

that the selected maximum value (hence, the maximum altitude) is higher than the tolerance distance in Axiom 9 of interactor *MCP*. The choice was to represent both as the range 0 to 5.

Regarding climb rate, the distinction between three situations must be made: climbing, holding altitude, or descending. Hence, three values will be considered: -1 (to represent all negative climb rates), 0, and 1 (to represent all positive climb rates). The following types are added to the specification:

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

As a consequence of this, the behaviour of interactor *plane* needs to be changed to take into account the maximum altitude. The spirit of this process is similar to that applied by Hayes (1990) in order to specify a continuous (ideal) process in a system with physical limitations. Finally, the name of interactor *MCP* is changed to *main*, and two extra clauses added to it: **test**, and **fairness**. Clause **test** is used to specify the CTL formula that should be checked by SMV. Clause **fairness** states that the system should not be continuously idle. The checkable version of the specification is presented in Appendix B. This version is automatically convertible to SMV using the compiler. The SMV code for the model is presented in Appendix C.

It is now possible to translate the model to SMV. The next step is to express the properties under analysis in CTL and check them using the model checker.

Formulating and checking properties

As seen in Section 4.4.2, the design of the interface has been based on the assumption that pilots expect that, if the altitude capture (*ALT*) is armed, the aircraft will stop at the desired altitude (selected in *ALTDial*). For the altitude acquisition case, this can be expressed as the CTL formula:

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

```
-- specification AG (plane.altitude < ALTDial.needle & AL... 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 : 198.7s, system time : 2.91s
BDD nodes allocated : 625225
Bytes allocated : 11075584
BDD nodes representing transition relation : 1787 + 301
```

Figure 4.13: SMV result for first attempt

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

When model checking a specification in SMV, the checker answers whether or not the test succeeds, and if the answer is false, and a counter example can be produced, it gives the first counter example it finds. Trying to check the specification against the formula above, produced the result presented in Figure 4.13[8]. What the model checker

[8]Note that from state to state only those values that have changed are shown. Also, for brevity only enough of the counter example to make the point is shown.

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

Since the model does not describe that aspect in detail, the analysis would have to refer back to the design in order to raise the point. If needs be, the model could then be refined to include this particular aspect of the automation behaviour in greater detail. This shows how the process is not self contained, but prompts questions that have to be dealt with at other stages of design.

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

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

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

Despite being a reasonable expectation, when the previous property is tried, the answer is still no. This time, the model checker points out that changing the pitch mode to *VERT_SPD* (for instance by setting the corresponding dial) when in ALT_CAP, effectively kills the altitude capture (i.e., the request to stop climbing at the target altitude). In effect, when the pitch mode changes to ALT_CAP, the altitude capture is automatically switched off. However, the aircraft is still climbing. This means that any subsequent action from the pilot that causes the pitch mode to change, will cause the aircraft to keep climbing past the target altitude.

Referring back to (Palmer 1995) it can be seen that this is a similar problem to that detected during simulation. Basically, once the aircraft changes into ALT_CAP

mode, there are user actions that might lead to a "kill the capture" mode error and a consequent altitude bust. Note that this result was achieved without using knowledge of the simulation results during the verification. In particular, SMV was given no specific chain of events to analyse. It was the tool that pointed out a particular sequence of events that could lead to a hazardous situation. This process could have been applied based only on a pen and paper scenario of an aircraft that was yet in its early design stages (in fact, that is the aim of the process) and effectively detected the problem.

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

4.4.5 Discussion

This section has looked at the automated verification of early specifications of interactive systems. Interactive systems are complex systems which pose difficult challenges to verification. Bringing the verification process closer to the design process affords better capturing the multiple concerns that come into play in the design of interactive systems, and better use of the available techniques.

The specific case study that was used is also analysed in (Leveson & Palmer 1997) and (Rushby 1999). Leveson & Palmer (1997) write a formal specification, based on a control loop model of process-control systems, using AND/OR tables. This specification is then analysed, manually, in order to look for potential errors caused by indirect mode changes (i.e., changes that occur without direct user intervention). An advantage of using a manual analysis process is greater freedom in the specification language, which can potentially lead to more readable specifications. However, the

possibility of performing the analysis in an automated manner will be an advantage when analysing complex systems. Here, the issue of readability is addressed by using a high level specification language (interactors) which is then translated into SMV.

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

Unlike the analysis presented here, however, Rushby (1999) builds his specification around the specific sequence of events that is identified in (Palmer 1995) as the cause for the altitude bust. His approach, then, is less flexible, and directly dependent on the hindsight from the simulation results. While we used the mode problem as a case study, the methodology can also be applied to the analysis of other issues, as the next chapters will show.

There are of course problems with this particular case study. Palmer's (1995) paper was already known and therefore the analysis was tainted by it. Nevertheless, building a generic model and using the scenario as a starting point for analysis is quite different from building the actual model around that scenario. In the latter case the results of the scenario directly influence the model so that the analysis is biased by hindsight. In the former case the scenario is used only to set up a context for verification, the verification process itself is independent from the results described in the scenario. In fact, in the case of a system still under development, the scenario might very well be only an idealised description of how the system should function in a particular situation.

4.5 Conclusions

This Chapter has shown how interactor specifications can be translated into SMV. The syntax and semantics of appropriate sets of both interactors and SMV modules

languages were modelled. This enabled the expression of interactors in the SMV framework, and the identification of the translation steps that should be performed. Two examples of application of the tool in conjunction with SMV were also described. The first was a very simple example of a mail client. Nevertheless it served as an introduction to the use of SMV, and showed how by choosing appropriate models, SMV can be used to reason about such user related issues as perception.

The second example was a more realistic case study. It concerned the analysis of the Mode Control Panel of an aircraft. CTL formulae were used to express user expectations about the operation of the artifact. During verification of such formulae, issues were raised about the behaviour of the system, and scenarios were found where the system did not behave as expected. The analysis of these results acted as a focus for further interdisciplinary discussion regarding the meaning of the results and how they should influence the design. This illustrates how the verification process can be useful by raising questions that have to be addressed in a broader context than the verification itself.

One problem with model checking is to find a model that is both sufficiently expressive and consisting of a small enough number of states. One aim of the chapter 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. This ability to reach interesting results from partial models of the system enables the analysis to overcome the problems that model checking poses when scaling up to complex systems. Verification can be done at the artifact level in order to verify that all artifacts function as required. Properties that were proved of individual artifacts can then be used to reason at a system wide level without having to include all the details of each artifact. In this way complexity is overcome by using a compositional style of verification (cf. Berezin et al. 1998).

The use of interactors and SMV gives a good degree of freedom and expressive power, but that does not come without a cost. In particular the use of CTL, while allowing for the expression of possibility, means that fairness concerns become an issue. As an example in the second case study a situation can happen where the pilot repeatedly sets the climb rate of the aircraft to zero, effectively preventing it from

reaching the altitude set in the capture. Situations of this kind can be solved either by imposing stronger fairness constraints on the system, by altering the property being checked, or by reworking the specification with the particular context of analysis in mind.

This chapter has shown how interactor models can be translated to SMV for model checking. A tool which performs this translation was described. The application of the tool was illustrated with an example dealing with issues related to mode complexity. The next chapter will deal with the use of theorem proving.

Chapter 5

Theorem Proving

The previous chapter has shown how model checking can be used in the verification of interactive systems designs. The properties that model checkers are able to verify have to do primarily with the behaviour of the system regarding the possible sequences of events/state changes. Thus, the analysis performed dealt essentially with the dialogue part of the design. This chapter looks at what can be learnt about the design of an interactive system using theorem proving.

5.1 Introduction

5.1.1 Theorem Proving vs. Model Checking

Attempting to repeat, in a theorem prover, the type of analysis previously performed with the model checker, a problem quickly becomes apparent: traditional theorem provers lack a temporal framework. The equational style of reasoning of traditional theorem provers deals with equality, while temporal properties deal with change (Meseguer 1992). In an equational reasoning context, given two expressions, if they rewrite to a common expression then they are equal. In order for this type of deduction to be possible, the rewriting system must be confluent and terminating. For an introduction to term rewriting see, for example, (Baader & Nipkow 1998). A rewriting system is confluent if for every two rewriting sequences $t \longrightarrow t_1$ and $t \longrightarrow t_2$ there always is a term t_3 along with rewriting sequences $t_1 \longrightarrow t_3$ and $t_2 \longrightarrow t_3$ (see Figure

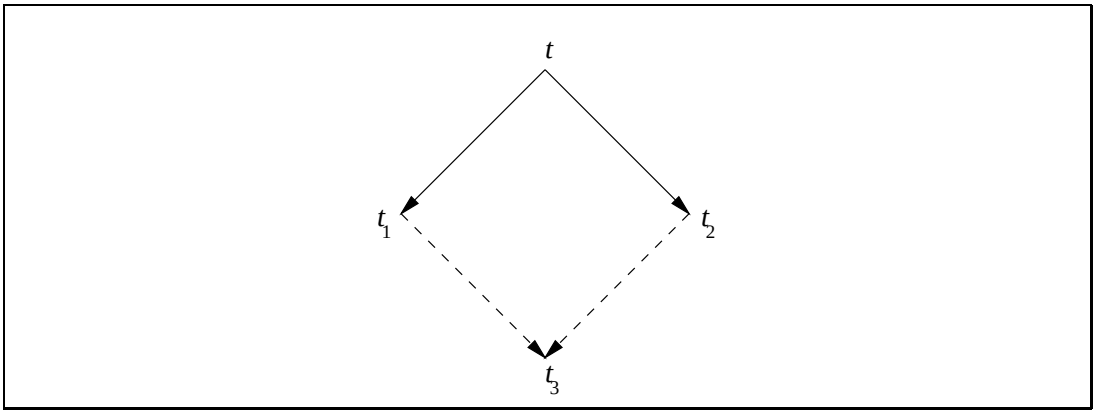

Figure 5.1: Confluence

5.1). That is, it is impossible to calculate different values from the same expression. In the context of temporal reasoning, however, this is not necessarily true. Think, for example, of how the finite state machines in SMV give rise to a tree of possible future states. The *problem* is that temporal systems will not be confluent. Rewriting is interpreted as change, not equality, so the order in which the rewriting rules are applied influences the result of calculating an expression. Unfortunately traditional equational reasoning has no way of allowing reasoning about these alternative behaviours.

A major advantage of theorem provers over model checkers, however, is their greater expressive power. In fact, it is even possible to develop a framework for reasoning about temporal issues in the context of a higher-order theorem prover. One possibility is to specify the state attributes as functions of time. In this way, it becomes possible to express temporal concepts such as possibility:

$$i(t_0) \rightarrow \exists_{t \in Time} \cdot t > t_0 \wedge p(t) \tag{5.1}$$

Another possible approach is to define the behaviour of the system over traces of events. This also enables the expression of temporal concepts. Possibility can be expressed as:

$$\exists_{t \in Trace \wedge i_{aux} \in State} \cdot \beta(i, t, i_{aux}) \wedge p(i_{aux}) \tag{5.2}$$

where β *calculates* the effect of a trace of events (see Section 5.4).

The expressive power of theorem provers, however, comes at the cost of decidability.

Going back to the examples above, although it was simple to express the concept of possibility, reasoning about it will be difficult. Since both time and sequences are potentially non finite types, proving an existential quantifier might become an undecidable problem. Even if the problem is decidable, while a model checker will automatically check all possible behaviours of the system, with a theorem prover it is usually up to the verifier to indicate which instantiations to use where. This tends to make the proofs very labour intensive.

Another problem which arises when expressing interactors in a theorem proving context, is the lack of a framework to reason about objects. Again, a framework for modelling objects can be developed using the theorem prover logic. The problem with this approach is that when all the machinery is finally in place in order to model systems of multiple interactors a considerable amount of *mathematics* has been created. This will make reasoning about the specification harder (cf. Bumbulis et al. 1996). In particular, it starts being unclear whether the reasoning has to do with the specified system or with the particular model of object oriented approach which is being used.

In summary, theorem proving is usually not as well suited for temporal reasoning as model checking is. Where the theorem prover gains advantage is in the expressiveness of the logics that are used, which allow for more flexibility and detail in the descriptions of the systems. With this added expressive power, comes the need for human intervention in the proof process.

5.1.2 Theorem Proving and Model Checking

Since model checkers and theorem provers have complementary capabilities, attempts have been made at integrating the two techniques. This has lead to the development of a number of tools where theorem proving and model checking are combined.

TLP (Engberg 1994, Engberg 1995) is a system that allows reasoning in TLA (the Temporal Logic of Actions) (Lamport 1994). The system does not perform reasoning itself. Instead, it provides a specification language (the TLP language) and translation from that language to the various back-ends that perform the actual verification. An interactive front-end (a GNU Emacs mode) provides an interface to the translator and

to the verification back-ends. In the latest release (2.5a — Engberg 1994), TLP uses two back-ends. The Larch prover (LP) (Garland & Guttag 1991, Guttag et al. 1993) and a model checker (LTL). A set of temporal axioms and rules (including the TLA proof rules) enable LP to perform temporal reasoning in TLA.

A different approach was taken to allow temporal reasoning in the theorem prover PVS (Rajan et al. 1995, Owre et al. 1997). Here the model checker (also LTL) has been integrated directly into the theorem prover as one more proof method. This integration was done using μ-calculus to define the temporal operators in PVS. In order for the model checker to be used, the specification must be in the form of a transition relation. One way to achieve this is to use PVS to construct a finite state abstraction of the specification and then use the model checker to check that abstraction. One problem with this approach is the feedback from the model checker to PVS. At the moment there is no way to provide information on why the checking of a property has failed.

STeP (Bjørner et al. 1996, Manna & Pnueli 1995) is another system that combines model checking with theorem proving. The system is aimed at the verification of concurrent and reactive systems. So, unlike PVS, this is not a general purpose tool. In this case, the main verification technique is model checking. For large or infinite state systems a theorem prover is used: verification rules reduce temporal properties to first-order verification conditions, and verification diagrams (Lamport 1995) can be used to perform the proofs.

5.1.3 The PVS theorem prover

The theorem prover which will be used in this thesis is PVS (Owre et al. 1992). The use of the tools above (including LP) was also considered. The objective was to choose a tool which was generic enough to allow the exploration of different types/styles of analysis. STeP, for instance, was found too biased towards concurrent program specification. TLP is more of a proof checker than a theorem prover. The proof must be fully written in advance, and is then checked by TLP. PVS was chosen due to its powerful higher order logic and verification environment. PVS is a typed higher-order logic theorem prover, which provides an integrated environment for development and

analysis of specifications. For a tutorial introduction to PVS see (Crow et al. 1995) or (Rushby & Stringer-Calvert 1996).

PVS specifications are organised in theories. Typically a theory will introduce a number of types and constants (which can be functions), and formulas associated with them (axioms and theorems, for instance). Theories can be parameterised on types and constants. Entities declared in a theory can be made available to others by exporting them (using the EXPORTING clause). By default all declarations are exported. Entities that are exported in a theory can be imported by another using the IMPORTING clause.

PVS features a powerful type system. This is very useful when writing specifications, but means that type checking becomes undecidable. To cope with this, the type checker generates proof obligations (TCCs - Type Correctness Conditions) that must be discharged by the theorem prover. If the system is unable to prove a TCC automatically, then the user is asked to do it. The usual types are available in PVS: natural numbers (nat), real numbers (real), sequences (sequence[$\mathcal{X}$]), sets (set[$\mathcal{X}$]), tuples ([#...#]), etc. These types are either built-in in the system or defined in the prelude library. A library is a collection of theorems. The prelude is a special library whose theories are always available, without the need for explicit importing. PVS also allows for the definition of predicate subtypes, dependent types and abstract data types.

Figure 5.2 gives an example of a PVS theory. The theory, adapted from Section 5.4, is named "attributes", and starts by declaring a uninterpreted (nonempty) type: "Message". It then imports theory "optional_types" using "Message" as a parameter. Two additional types are declared: "attributes" and "presentation"; both of which are tuples. One of the types has the same name as the theory, context of usage will determine whether one or the other is meant. Note the use of type "list", this is a data type defined in the prelude. After these types a function is declared (ρ). Finally, a variable and a theorem are also declared. The theorem is trivial, and it is meant to serve just as an example.

The PVS system is used interactively. Its interface is mainly implemented as an Emacs major mode, which integrates functionality for editing specifications and prov-

```
attributes : THEORY
  BEGIN

  Message : TYPE+

  IMPORTING optional_types[Message]

  attributes : TYPE  = [# queue : list[Message],
                          current : optional[Message]
                       #]

  presentation : TYPE  = [# current : optional[Message] #]

  ρ : [attributes → presentation] = λ (s : attributes) : (#current := current(s)#)

  q : VAR attributes
  extheorem : THEOREM current(q) = current(ρ(q))

  END attributes
```

Figure 5.2: A PVS theory

ing theorems. When performing a proof, the system presents a goal in the form of a sequent, and prompts the user for an appropriate command. If the command does not solve the sequent, it will generate a new sequent or a number of new sequents (i.e. subgoals), and the user will be asked to prove them in turn. When all subgoals are proved the original goal has been proved.

The specific version of PVS used to perform the proofs in this thesis was version 2.1 (patch level 2.414).

5.2 Modelling Interactors in PVS

The flexibility that theorem provers provide, means that they can be used in a number of different ways. This is unlike model checkers, which specialise in a specific type of model and properties. This freedom means, in turn, that models for interactive systems can be built in many different ways, depending on the purpose in mind. This section introduces a generic framework for the expression of interactors in PVS. This framework is meant mainly as an example of how theorem provers can be used to model interactors. As will be shown subsequently, the framework can be used freely depending on the task at hand. Sometimes the framework is followed closely, and at

others used only as a guide. This means that there is no point in automating the translation from interactors to PVS, since that would restrict the freedom to adopt the best modelling strategy in each concrete situation.

As said in Section 5.1.1, two possibilities to express change are indexing the state to some time index, and including the notion of traces of behaviour in the model. The latter approach will be taken. Following (Duke & Harrison 1993), and assuming a set of events (E), states (S), and presentations (P), an interactor is defined by a tuple $(\alpha, \tau, \iota, \beta, \rho)$ where:

- $\alpha : \mathbb{P}\, E$ — is the alphabet of the interactor (a set of events);
- $\tau : \mathbb{P}\, TRACE$ — is the sequences of events the interactor can engage in (a set of traces);
- $\iota : \mathbb{P}\, S$ — is the set of initial states;
- $\beta : E \nrightarrow (S \leftrightarrow S)$ — maps each event to a set of state transitions;
- $\rho : S \leftrightarrow P$ — gives the relation between states and presentations.

The current model defines the events accepted by the interactor, the set of initial states of the interactor, and a (labelled) transition system defining the effect of each event on the state. Note that permission and obligation are not considered directly. Instead, predicates defining valid traces are included.

This model can easily be represented as a tuple type in PVS. Assuming that $\alpha = \mathrm{E}$, i.e. that interactors can accept all events, allows α to be removed from the tuple and simplifies the model. For example, it is no longer necessary to make sure that all events from traces in τ are in α. Figure 5.3 shows the resulting PVS theory. The auxiliary function "apply" calculates the set of all states that can be reached by executing a trace t from states in s. R and V are the result and view relations: for each trace they give the set of states/presentations that will result from executing the trace from the initial state. The theory is parameterised over states, presentations and events (S, P and E). These types will only be known when the theory is instantiated to obtain a concrete interactor. Note that ρ has been represented as a function from states to

```
InterActor[S : TYPE, P : TYPE, E : TYPE] : THEORY
  BEGIN
  Trace : TYPE  = list[E]
  Interactor : TYPE  = [# τ : pred[Trace],
                          ι : setof[S],
                          β : [E → pred[[S, S]]],
                          ρ : [S → P]
                       #]

  apply(s : setof[S], t : Trace, i : Interactor) : RECURSIVE setof[S] =
    CASES t OF
      null : s,
      cons(hd, tl) : apply({s1 : S |
      ∃ (s2 : (s)) : β(i)(hd)(s2, s1)}, tl, i)
      ENDCASES
    MEASURE length(t)

  R(i : Interactor) : pred[[Trace, S]] = λ (t : Trace), (s : S) : (s ∈ apply(ι(i), t, i))

  V(i : Interactor) : pred[[Trace, P]] =
    λ (t : Trace), (p : P) : ∃ (s : S) : R(i)(t, s) ∧ ρ(i)(s) = p

  END InterActor
```

Figure 5.3: Interactors in PVS

presentations. What this implies is that each state has only one possible presentation. For analysis relating to the relationship between states and presentations, this might not be acceptable. In that case a relation should be used in the definition of ρ. In the present case such will not be needed and using a function will help simplify some of the proofs in Section 5.4. This is an example of the freedom that theorem provers allow when modelling, and how it can be used to model according to what is intended. Note also that this model allows reasoning about single interactor models only, since it makes no provision for inter-interactor communication.

Until now only input events have been considered. The meaning of these is defined by relations which express what states can result from executing each event. In a system with several interactors, the need arises to have interactors sending events to each other. Then, the mechanism through which this communication happens must be modelled. If inheritance and interactor composition are also considered, the model can become very complex very quickly. Again, the alternative is not to build generic models of the system but focus on, and model, specific aspects. Hence, this model will

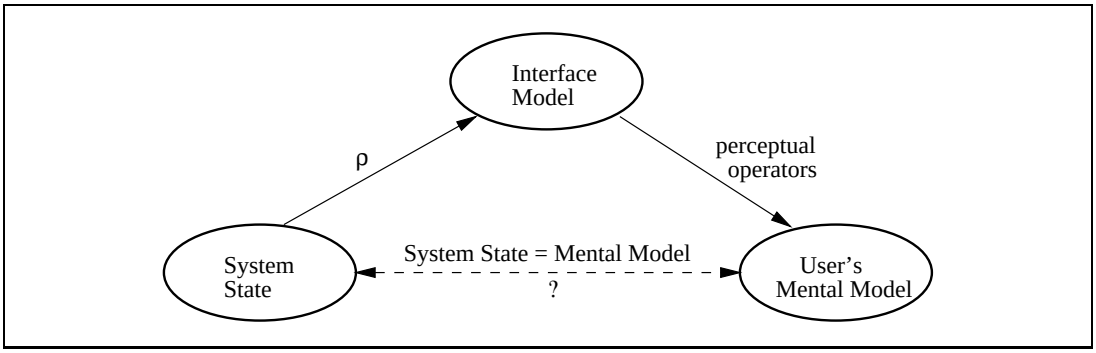

Figure 5.4: Gulf of Evaluation

not be developed further. Instead, the next two sections will show how PVS can be used to perform different styles of analysis. In Section 5.4 the present model will be used.

5.3 Evaluating the Gulf of Evaluation

Section 3.2 identified an equation for theorems related to Norman's (1988) Gulf of Evaluation (Equation 3.3). The equation related the user's perception of the state to the actual system state. Figure 5.4 illustrates this. This section deals with the problem of putting this type of reasoning into practice.

Initially, a VDM (Jones 1986) specification was put forward by a third party, together with specific requests for analysis. The idea for this analysis was triggered by (Hutchins 1995). However, the analysis was based on the VDM model, and on the questions, which were provided. The results were related back to Hutchins's (1995) psychology based analysis. A complete account of this case study can be found in (Doherty et al. 2000). Here, the PVS version of the specification will be first introduced, and the verification that was performed will then be described.

5.3.1 The airspeed indicator

In this case, the artifact to be analysed, and the analysis that should be performed, were given on the outset. So, the first stage of the analysis process proposed in Chapter 3 (see Figure 3.4 on page 94) was preempted.

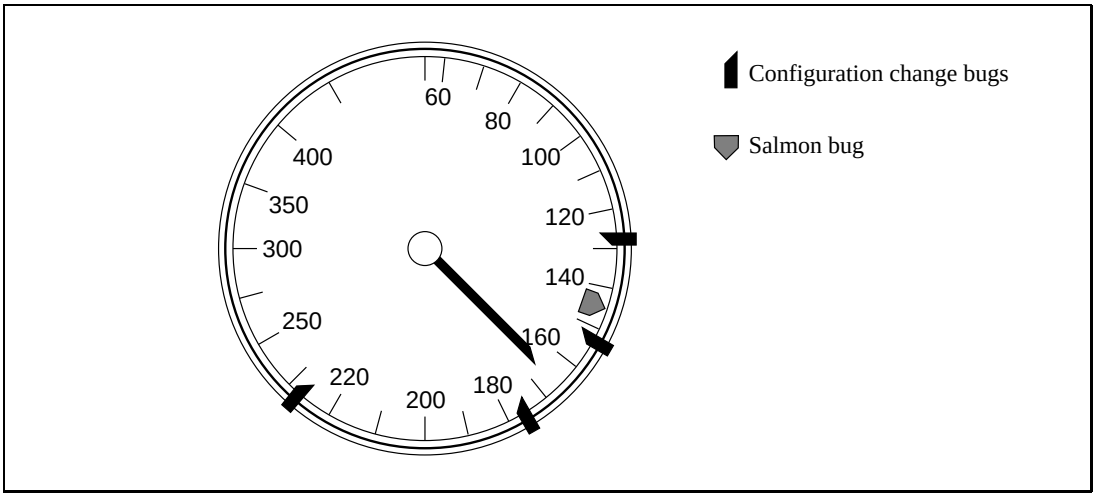

Figure 5.5: Airspeed indicator (adapted from Hutchins 1995)

The artifact under analysis is an airspeed indicator (see Figure 5.5). During final approach, the aircraft's wings go through a number of different configurations, in order to maintain appropriate lift, as the aircraft slows down to the approach speed (V_{ref}). Each wing configuration has a minimum manoeuvering speed, below which the aircraft might become unstable. Hence, the aircraft's speed must not go below the minimum manoeuvering speed for the current configuration. In the current case small indicators (the configuration change bugs) are placed around the dial's perimeter to indicate the minimum manoeuvering speed for each of the configurations that will be used during the approach. The pilots must keep track of the current speed (indicated by the needle in the dial) in relation to the minimum manoeuvering speed (indicated by the bugs) of the current configuration. As the speed decreases, the appropriate configurations must be selected. Once the aircraft reaches the approach speed (indicated by the salmon bug), its speed must be maintained within a predefined safe margin of that speed until landing.

The analysis requested was concerned with how the bugs would be used by the user to maintain an appropriate configuration. Two specific points were a focus of concern:

- checking the speed against the reference speed — will the user be able to determine whether the current speed is inside the safe margin of the reference speed?
- checking the speed against the minimum manoeuvering speed of the current con-

figuration — will the user be able to determine when it is appropriate/necessary to change configuration?

The next two sections describe the modelling and the proofs that were carried out to answer these questions.

5.3.2 Modelling the airspeed indicator in PVS

The questions above are related to how the pilots will perceive the state of the interface, and how this perception relates to the actual state of the system. These are the type of questions that can be expressed using Equation 3.3 (page 81). This type of analysis, dealing with questions of equality between the states of different models, is best performed in a theorem proving context. PVS will be used.

The framework presented in Section 5.2 will be followed only loosely. Since no actions will be modelled, the interactor theory will not be used. Only the state attributes will be modelled. The model is organised into three theories: one for the functional model (theory ASI), another for the interface model and perceptual operators (theory perceptualASI), and a final theory that combines the previous ones and introduces the equivalences to be proved (theory ASIverification).

Figure 5.6 presents the PVS theory for the functional model. The theory starts by introducing the types:

- "Speed" — to represent air speed;
- "Speeds" — sequences of "Speed";
- "Configuration" — a natural number representing the different configurations;
- "abstractASI" —a tuple with four elements, representing the state of the system:
  - "Vc" — the current speed;
  - "Cc" — the current configuration;
  - "Smm" — the minimum manoeuvering speeds;
  - "Vref" — the approach reference speed.

```
ASI : THEORY
  BEGIN
  Speed : TYPE  = nat
  Speeds : TYPE  = sequence[Speed]
  Configuration : TYPE  = nat
  AbstractASI : TYPE  = [# Vc : Speed,
                           Cc : Configuration,
                           Smm : Speeds,
                           Vref : Speed
                        #]
  Smargin : Speed
  Ssafe : Speed

  abs_asi : VAR AbstractASI
  inv_abs_asi2 : AXIOM
    ∀ (i, j : nat) : i < j ⇔ Smm(abs_asi)(i) > Smm(abs_asi)(j)

  configChangeCheck(asi : AbstractASI) : bool =
                    Vc(asi) ≤ Smm(asi)(Cc(asi)) + Smargin
  approachSpeedCheck(Vc, Vref : Speed) : bool =
                      Vref − Ssafe ≤ Vc ∧ Vc ≤ Vref + Ssafe
  END ASI
```

Figure 5.6: Initial version of the functional model

After the types, two constants are introduced:

- "Smargin" — the margin above the minimum manoeuvering speed inside which the configuration can be changed;
- "Ssafe" — the safe margin around "Vref".

Note that they are left uninterpreted (i.e. no actual values are given). The variable "abs_asi" is declared and used in the axiom that follows (axiom "inv_abs_asi2"). This axiom asserts the invariant of "abstractASI" (the state of the model): that the bugs are arranged in order of decreasing speed. Finally, the theory declares the two logical operators:

- "configChangeCheck" — this operator determines whether the configuration should be changed;
- "approachSpeedCheck" — this operator determines whether the current speed is inside the safe margin of "Vref".

The theory in Figure 5.6 is the initial version of the model obtained by translating the VDM specification in (Doherty et al. 2000). As it will be shown, the verification process will prompt some changes in the specifications.

The "perceptualASI" theory is introduced in Figure 5.7. An immediate impression is the difference in size between the two models (even if Figure 5.7 does not show all of the theory). This happens because theory "ASI" is an abstract model of the system functionality, while theory "perceptualASI" is a model of the actual physical dial. All the relevant components of the artifact are modelled. This is done with type "ASIInstrument" a tuple with four elements:

- "needle" — the needle's position, represented as an angle;
- "bugs" — the sequence of speed bug around the dial, represented by their position and extent (size);
- "salmonbug" — the position and size of the salmon bug;
- "scale" — the scale around the dial.

The size of the bugs is represented by the constants "BugExtent" and "SalmonExtent". Axiom "inv_ASIInstrument" is the invariant for "ASIInstrument". It states that the sequence of bugs is sorted by position, and that bugs cannot be closer to each other than what their size physically allows.

Function ρ creates a dial from a system state. Several auxiliary functions are used, each translating a component of the dial. Finally, functions "salmonBugCheck" and "asiConfigCheck" are the perceptual counterparts of "approachSpeedCheck" and "configChangeCheck". The auxiliary functions used in these two last functions represent perceptual operations that the user needs to carry out in order to interpret the interface. Besides being used in the formal analysis that follows, they can act as starting points for discussion, regarding whether those are sensible expectations to have of the user's perception capabilities. The final version of theory "perceptualASI" (after verification) is given in Appendix D.

The last theory, ASIverification (see figure 5.8), imports the two previous theories and introduces the equivalences to be proved.

```
perceptualASI : THEORY
  BEGIN
  ASSUMING
    IMPORTING ASI
    ENDASSUMING

  Angle : TYPE = nonneg_real
  ASINeedle : TYPE = [# posn : Angle#]
  ASISpeedBug : TYPE = [# posn : Angle, extent : Angle#]
  ASISpeedBugs : TYPE = sequence[ASISpeedBug]
  ASIScale : TYPE = [# interpret : [Angle → nonneg_real] #]
  ASIInstrument : TYPE = [# needle : ASINeedle,
                            bugs : ASISpeedBugs,
                            salmonbug : ASISpeedBug,
                            scale : ASIScale
                          #]

  ScaleFactor : posreal
  BugExtent, SalmonExtent : Angle

  bug_posn_extent : AXIOM ∀ (bug : ASISpeedBug) : posn(bug) − extent(bug) ≥ 0
  asi : VAR ASIInstrument
  abs_asi : VAR AbstractASI
  inv_ASIInstrument : AXIOM ∀ (i, j : nat) : i < j ⇒
     (posn(bugs(asi)(i)) > posn(bugs(asi)(j))∧
          posn(bugs(asi)(i)) − extent(bugs(asi)(i)) >
                    posn(bugs(asi)(j)) + extent(bugs(asi)(j)))
  ...

  ρ(a : AbstractASI) : ASIInstrument = (#needle := ρNeedle(Vc(a)),
                                         bugs := ρBugSeq(Smm(a)),
                                         salmonbug := ρSalmon(Vref(a)),
                                         scale := ρScale
                                       #)
  ...

  getCurrentBug(needle : ASINeedle, bugs : ASISpeedBugs) :
    ASISpeedBug = next_counterclockwise(needle, bugs)
  salmonBugCheck(needle : ASINeedle, bug : ASISpeedBug) : bool =
    in_arc(needle, posn(bug) − extent(bug), posn(bug) + extent(bug))
  configBugCheck(needle : ASINeedle, bug : ASISpeedBug) : bool =
    in_arc(needle, posn(bug), posn(bug) + Smargin / ScaleFactor)
  asiConfigCheck(asi : ASIInstrument) : bool =
    configBugCheck(needle(asi), getCurrentBug(needle(asi), bugs(asi)))
  END perceptualASI
```

Figure 5.7: Initial version of the interface model

```
ASIverification : THEORY
  BEGIN

  IMPORTING ASI, perceptualASI

  vc, vref : VAR Speed

  approach_speed_task : CONJECTURE
    approachSpeedCheck(vc, vref) = salmonBugCheck(rho_Needle(vc),
                                                  rho_Salmon(vref))

  abs_asi : VAR AbstractASI

  configuration_change_task : CONJECTURE
    configChangeCheck(abs_asi) = asiConfigCheck(ρ(abs_asi))

  END ASIverification
```

Figure 5.8: ASIverification theory

5.3.3 Verification and analysis

Approach speed task

The first analysis which was requested relates to how the user perceives whether or not the plane is inside the safe margin of the reference speed for landing. This is expressed by conjecture "approach_speed_task" in Figure 5.8. The conjecture declares the equivalence between the operators "approachSpeedCheck" (the test at the functional level) and "salmonBugCheck" (the test at the user level).

Starting the proof, in PVS, by applying a `grind` (a powerful, somewhat brute force, rule which, among other commands, applies rewriting and simplification) leaves four subgoals to be proved (see Figure 5.9). The sequent for the first subgoal is (where the prime symbol is used to indicate skolem constants[1]):

Sequent 5.3.1 (approach_speed_task.1)

$$
\begin{array}{|ll}
\{-1\} & \mathrm{vc}' \geq 0 \\
\{-2\} & \mathrm{vref}' \geq 0 \\
\{-3\} & \mathrm{vref}' - \mathrm{Ssafe} \leq \mathrm{vc}' \\
\{-4\} & \mathrm{vc}' \leq \mathrm{Ssafe} + \mathrm{vref}' \\
\hline
\{1\} & (\mathrm{vc}' \ / \ \mathrm{ScaleFactor} \leq \mathrm{SalmonExtent} + \mathrm{vref}' \ / \ \mathrm{ScaleFactor})
\end{array}
$$

[1]Skolem constants are arbitrary representatives for quantified variables.

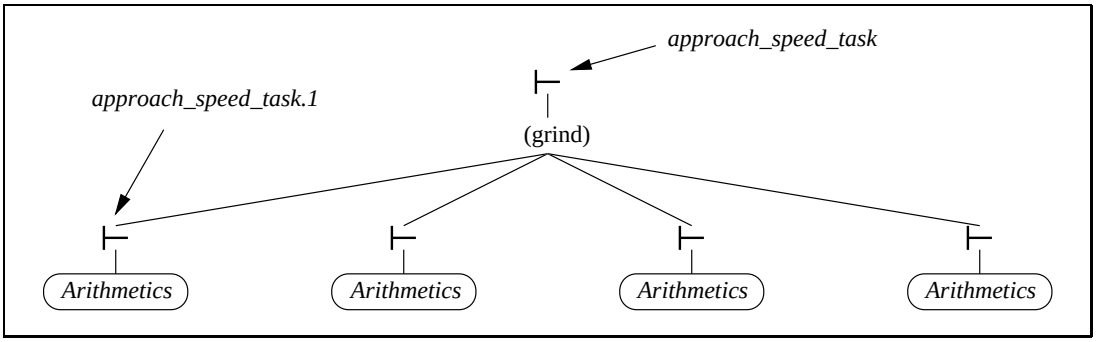

Figure 5.9: Schematic proof tree for approach_speed_task

What PVS is asking to be proved is that, if the current velocity (vc′) is inside the safe margin of the reference speed (vref′), then the needle must be below the speed indicated by the edge of the salmon bug when positioned at "vref′" in the dial. What is happening here is that an assumption about the Salmon bug (that its extent indicates the safe margin) is being made at the *operational level* of the presentation (the definition of how the presentation will be used) but not at the level of its state definition (the definition of what the presentation is). It is easy to imagine such a situation arising when, for example, more than one team is working on the design. Note how the proof process can be useful at detecting inconsistencies between the different levels at which the interactive system can be looked at.

To incorporate this assumption into the state definition, "SalmonExtent" must be defined as an interpreted constant which represents, at the perceptual level, the value of "Ssafe":

SalmonExtent : Angle = Ssafe / ScaleFactor

Since the size of the salmon bug becomes a critical issue, its definition must be analysed with care. In particular the scale factor used at the dial gains relevance. The model assumes a linear scale factor. What happens if, as Figure 5.8 seems to indicate, the scale is not linear? This will invalidate the use of the salmon bug size as an indication of the safe margin, since, at different positions, its size will represent different speed margins. This might indicate a flaw in the design (more accurately, in how the design is expected to be used). On the other hand, since the approach speed will be restricted

to a specific range of values, it might be the case that the scale is close enough to linear in that subset. In that case, then, the design will work appropriately. The verification process is raising questions about the design. Questions that need to be addressed in a broader context than that of the proof process alone.

With the above definition of "SalmonExtent" it becomes easy to prove Sequent 5.3.1. The other three sequents are of similar nature and easily proven once the relation above is established[2].

Configuration change task

The previous analysis dealt with keeping the aircraft inside the reference speed for landing. Another important task the pilots need to carry out, during the landing procedure, is maintaining the aircraft in the correct configuration for the speed. For that they need to be aware of when configuration should be changed. At the functional level this is determined by function "configChangeCheck", at the interface level, the perceptual process the pilot is expected to use is modelled by function "asiConfigCheck". The conjecture that represents the equivalence between the two functions is introduced in theory ASIverification (see Figure 5.8).

The proof starts by skolemising and expanding definitions. Eventually it reaches a point where, after introducing the definition of "next_counterclockwise", it splits into two subgoals: *configuration_change_task.1* and *configuration_change_task.2* (see Figure 5.10). Proceeding with the first subgoal, after expanding definitions and some arithmetic simplifications, the following sequent is reached, where "ccbugindex" is a skolem constant representing the index of the first bug just below the needle (which is interpreted in "configChangeCheck" as representing the current configuration):

Sequent 5.3.2 (configuration_change_task.1)

$$
\begin{array}{ll}
\{\text{-}1\} & \text{posn}(\rho\text{BugSeq}(\text{Smm}(\text{abs\_asi}'))(\text{ccbugindex})) \leq \text{posn}(\rho\text{Needle}(\text{Vc}(\text{abs\_asi}'))) \\
\{\text{-}2\} & (\forall\ (j:\ \text{nat}) : j\ <\ \text{ccbugindex} \Rightarrow \\
 & \quad \text{posn}(\rho\text{BugSeq}(\text{Smm}(\text{abs\_asi}'))(j))\ >\ \text{posn}(\rho\text{Needle}(\text{Vc}(\text{abs\_asi}')))) \\
\hline
\{1\} & (\text{Vc}(\text{abs\_asi}') \leq \text{Smm}(\text{abs\_asi}')(\text{Cc}(\text{abs\_asi}'))\ +\ \text{Smargin} = \\
 & \quad \text{configBugCheck}(\rho\text{Needle}(\text{Vc}(\text{abs\_asi}')), \rho\text{BugSeq}(\text{Smm}(\text{abs\_asi}'))(\text{ccbugindex})))
\end{array}
$$

[2]The proofs in this chapter are available at: `http://www.cs.york.ac.uk/~jfc/thesis/proofs.ps`.

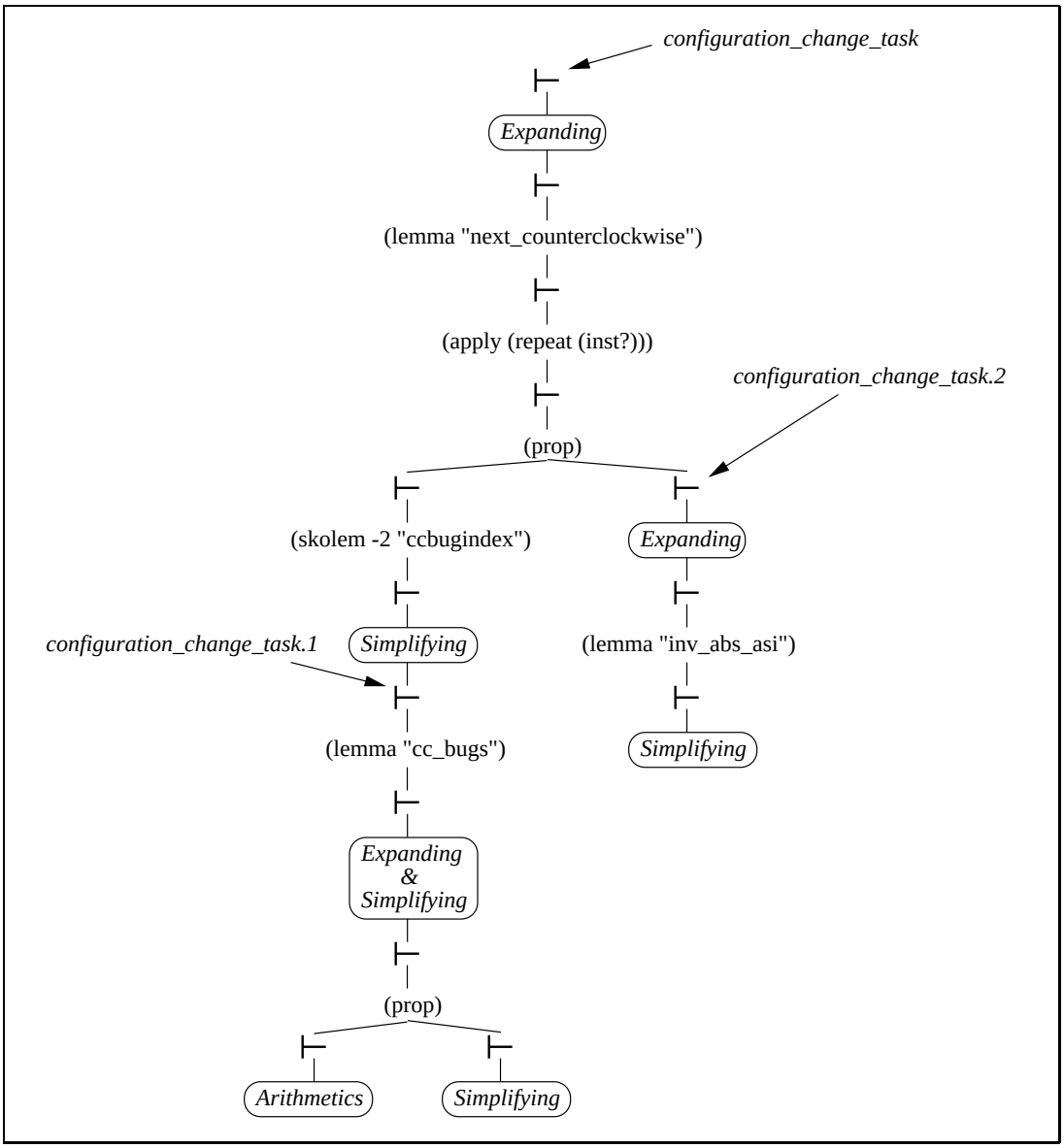

Figure 5.10: Schematic proof tree for configuration_change_task

The sequent can be read as: if "needle" points to a velocity above or equal to the bug "ccbugindex" (antecedent -1), and all bugs above "ccbugindex" are also above the needle (antecedent -2), then, testing that the current (functional level) velocity (Vc(abs_asi′)) is below the minimum manoeuvering speed of the current configuration (Smm(abs_asi′)(Cc(abs_asi′))) plus the safe margin, yields the same result as performing a "configBugCheck" on the needle and the bug just below the needle. This is not unexpected and should be true, as the bug just below the needle is interpreted as indicating the current configuration. To prove this, it must be possible to establish that "Cc(abs_asi′)" (the configuration index at the functional level) and "ccbugindex" (the configuration index at the interface level) point to the same minimum manoeuvering speed. This corresponds to assuming that, at the user interface, the bug just below the needle always indicates the current configuration. This assumption can be formalised as the axiom:

Axiom 5.3.1 (cc_bugs)

$$\forall\ (i:\ \text{nat}, \text{abs\_asi} : \text{AbstractASI}):\ ((\text{posn}(\text{bugs}(\text{asi})(i)) \times \text{ScaleFactor} \leq \text{Vc}(\text{abs\_asi})) \wedge (\neg \exists\ (j:\ \text{nat}): j < i \wedge (\text{posn}(\text{bugs}(\text{asi})(j)) \times \text{ScaleFactor} \leq \text{Vc}(\text{abs\_asi})))) \Rightarrow i = \text{Cc}(\text{abs\_asi})$$

By forcing this relation between the needle and the bugs onto the perceptual model, the proof process is unveiling relationships between the different components of the presentation which were not initially considered. In this case it is pointing out that pilots will use bugs to detect the current configuration of the aircraft. The possibility of being able to predict how pilots might interpret the presentation is valuable to the design process. In particular it might be asked whether the bugs are a suitable representation for the current configuration. Next, it will be shown how considerations about this also arise from the proof.

After applying the axiom, the proof proceeds once again by expanding definitions and simplifying. Eventually, the subgoal is further subdivided into two subgoals, both of which are easy to prove. The subgoal *configuration_change_task.2* is left. This subgoal is represented by the sequent:

Sequent 5.3.3 (configuration_change_task.2)

$$\{1\} \quad (\exists\ (i:\ \text{nat}):\ (\text{Smm}(\text{abs\_asi}')(i)/\text{ScaleFactor} \leq \text{Vc}(\text{abs\_asi}')/\text{ScaleFactor}))$$

It has to be shown that a bug exists below the needle. This stems from the precondition to "next_counterclockwise". What this points out is that the pilot must be able to determine the current configuration. In conjunction with Axiom "cc_bugs" above it can be concluded that it is crucial that the pilot never lets the speed go below the current configuration bug. A change to the next configuration should be performed before that happens. Failing to do so would mean that the information about the configuration of the aircraft would be incorrectly perceived. The assumption can be formalised with the following invariant for "ASI":

Axiom 5.3.2 (inv_abs_asi)

$$\forall(\text{abs\_asi} : \text{AbstractASI}) : \text{Vc}(\text{abs\_asi}) \geq \text{Smm}(\text{abs\_asi})(\text{Cc}(\text{abs\_asi}))$$

Being operational in nature, this assumption prompts the analysis of how realistic it is to assume that it will hold, and whether the proposed presentation should be changed or used in conjunction with some other perceptual artifact explicitly representing the current configuration. It might even be decided to go back to the functional state and make a change there (for instance, introducing interlocks). This shows how the proof process prompts reasoning about the artifact in the context of the overall design of the cockpit.

Using the above invariant the proof for this subgoal can be finished. Proving the subgoal, concludes the proof of "config_change_task".

5.3.4 Discussion

More than showing that a property does or does not hold, this example highlights how the verification process can be useful at providing greater insight into the design. In the end, the question was not so much "are the properties true of this design?", but rather "in what conditions are the properties true of this design?".

In the process of proving the properties a number of mismatches between the two models was found. Some arose because of assumptions about the meaning of specific

interface components (and how they would be used) that were made by the designers of the presentation (the salmon bug extent as a representation of the safe margin for the reference speed). Others, because of the meanings that might be *imposed* on the interface by its users (the notion of the speed bugs as identifying specific configurations), together with operational constraints that must be obeyed in order for the user's mental model to remain consistent with the system state (Axiom 5.3.2). These last issues are primarily of a cognitive nature. It is interesting to note that they were unveiled using a primarily software engineering type of approach.

5.4 Analysing the CERD Message Display Window

The next example, taken from (Harrison et al. 1996), is a specification of the CERD Message Display Window. The previous example dealt only with the presentation of the system. In this example, the analysis is concerned with specific actions, and whether they achieve the desired goals. In terms of the framework presented in Figure 3.2 (page 80), the analysis is primarily concerned with the horizontal dimension. As it will be shown, the vertical dimension needs also to be considered, in order to account for usability issues.

5.4.1 The CERD

The CERD (Computer Entry and Readout Device) is a component of a data-link Air Traffic Control (ATC) system, developed by Praxis Systems Ltd. (Hall 1996). The CERD is a touch sensitive screen that replaces the flight strips commonly used by ATC Officers (ATCO). The user interface of the system has already been formally modelled in (Duke & Harrison 1994*a*, Paternò & Mezzanotte 1995). Here, the Message Display Window will be used to exemplify some points on the use of theorem provers to analyse interactive systems. A model of the CERD presented in (Harrison et al. 1996) will be used.

The Message Display Window is responsible for showing incoming messages to the ATCO. Incoming messages are kept internally in a queue, and the head of the queue

interactor *MessageDisplay*
attributes
$queue : Message^*$
vis $current : [Message]$
actions
$receive(m : Message)$
vis $clear$
axioms
(1) $queue = <> \rightarrow current = \text{nil}$
(2) $queue \neq <> \rightarrow current = \text{hd}\ queue$
(3) $per(clear) \rightarrow queue \neq <>$
(4) $[clear]\ queue' = \text{tl}\ queue$
(5) $[receive(m)]\ \exists_{l,r:Message^*} \bullet (queue = l \frown r) \wedge (queue' = l \frown <m> \frown r)$
$\wedge\ (l \neq <> \rightarrow pr(last(l)) \leq pr(m))$
$\wedge\ (r \neq <> \rightarrow pr(\text{hd}\ r) > pr(m))$

Figure 5.11: The CERD Message Display Window (adapted from Harrison et al. 1996)

is displayed on the window. Each message has an associated priority, and the message queue keeps the messages by descending order of priority.

As in the previous example, the model and properties are given at the outset. The interactor model of the system is presented in Figure 5.11. The interactor can receive messages through event *receive*. Each message has an attached priority (given by *per*) and messages are kept in a queue sorted by priority. Note that higher numbers mean lower priority. The head of the queue is visible and the user can delete it with event *clear*. In (Harrison et al. 1996) some desirable properties of the system are put forward. One deals with the arrival of new messages, and will be analysed here. The other, deals with behavioural aspects of the system and would be best analysed using a model checker. Hence, in a real verification context, the model above would be analysed with different tools, according to the properties being explored. Here, the following two questions will be considered:

- Is it true that, if a message with higher priority than the one at the head of the queue arrives, then that message becomes the message on the screen? This is a generalisation of the first property in (Harrison et al. 1996).
- Is it true that, after clear is pressed, then the message at the top of queue is removed?

Both properties relate to the effect of specific actions in the state, hence the use of

```
attributes : THEORY
  BEGIN
  Message : TYPE+

  IMPORTING optional_types[Message]

  attributes : TYPE = [# queue : list[Message], current : optional[Message] #]
  presentation : TYPE = [# current : optional[Message] #]

  ρ : [attributes → presentation] = λ (s : attributes) : (#current := current(s)#)

  END attributes
```

Figure 5.12: State and Presentation of MessageDisplay

```
actions : THEORY
  BEGIN
  ASSUMING
    IMPORTING attributes
    ENDASSUMING

  actions : DATATYPE
     BEGIN
     clear : clear?
     receive(msg : Message) : receive?
     END actions

  END actions
```

Figure 5.13: Actions for MessageDisplay

theorem proving.

5.4.2 Modelling in PVS

Unlike the previous example, actions need to be considered, so the theory developed in Section 5.2 will be used. To specify interactor *MessageDisplay*, three theories were written. The first (theory "attributes") defines the interactor state (its attributes), as well as its presentation (see Figure 5.12). The second (theory "actions") defines the possible actions: "clear" and "receive" (see Figure 5.13). Given an action, predicates "clear?" and "receive?" can be used to determine its type. Finally, theory "messageDisplay" (see Figure 5.14) uses the previous two theories to create interactor "messagedisplay".

A fourth simple theory was also written: "optional_type". It defines the "optional"

```
MessageDisplay : THEORY
  BEGIN
  IMPORTING attributes
  IMPORTING actions
  IMPORTING InterActor[attributes, presentation, actions]

  pr : [Message → int]

  β(ev : actions) : pred[[attributes, attributes]] =
    CASES ev OF
      clear :
          λ (s1, s2 : attributes) :
           queue(s1) ≠ null∧
             queue(s2) = cdr(queue(s1))∧
               IFqueue(s2) = null THEN current(s2) = nil
                ELSE current(s2) = value(car(queue(s2)))
                ENDIF,
      receive(msg) :
          λ (s1, s2 : attributes) :
           ∃ (l, r : list[Message]) :
             queue(s1) = append(l, r)∧
               queue(s2) = append(l, cons(msg, r))∧
                 (l = null∨
                     pr(nth(l, length(l) − 1)) ≤
                       pr(msg))∧
                  (r = null ∨ pr(car(r)) > pr(msg))∧
                    current(s2) = value(car(queue(s2)))
      ENDCASES

  sorted : pred[attributes] =
    λ (s : attributes) :
      ∀ (i, j : {k : int | k ≥ 0 ∧ k < length(queue(s))}) :
        i ≤ j ⇒ pr(nth(queue(s), i)) ≤ pr(nth(queue(s), j))

  valid : pred[attributes] =
    {s : attributes |
    (queue(s) = null ∧ current(s) = nil)∨
      (queue(s) ≠ null ∧ sorted(s) ∧ current(s) = value(car(queue(s))))}

  messagedisplay : Interactor =
    (#τ := λ (t : Trace) : TRUE, ι := valid, β := β, ρ := ρ#)

  END MessageDisplay
```

Figure 5.14: Initial version of the abstract model

data type, which is used to model attributes that can be undefined (cf. attribute "current")

5.4.3 Verification and analysis

With these PVS theories, the properties can now be expressed and the verification and analysis steps performed.

Verification of the first property

The first property to be considered is:

> If a message with higher priority than the one at the head of the queue arrives, then that message becomes the message on the screen.

An initial formulation of the first property above could be:

Theorem 5.4.1 (verify_receive_higher)

$$\begin{array}{l}\forall\ (\text{initial},\ \text{final}:\ \text{attributes},\ \text{m}:\ \text{Message}):\\ \quad(\text{valid}(\text{initial}) \wedge\\ \qquad \text{queue}(\text{initial}) \neq \text{null} \wedge\\ \qquad\ \ \text{pr}(\text{get}(\text{current}(\text{initial}))) > \text{pr}(\text{m}) \wedge\\ \qquad\qquad \beta(\text{messagedisplay})(\text{receive}(\text{m}))(\text{initial}, \text{final})) \supset \text{get}(\text{current}(\rho(\text{final}))) = \text{m}\end{array}$$

where "messagedisplay" is the interactor, "initial" and "final" are two states (i.e. the interactor attributes), and "m" is the message to be received. Function "get" is used to retrieve a "Message" value from a "optional[Message]" value. Finally "queue(initial)≠null" is included since it only makes sense to calculate the priority of the current message if the message queue is not empty (if this condition is left out, the prover will generate a TCC asking for it to be shown that the priority of the current message can always be calculated).

A schema for the proof of this theorem is presented in Figure 5.15. The proof is conceptually simple. During the proof, PVS presents several sub-cases to be solved. In the majority they can be discarded due to contradictions between the antecedents (relating to the ordering of priorities between different messages). Consider *Sequent .3* from Figure 5.15:

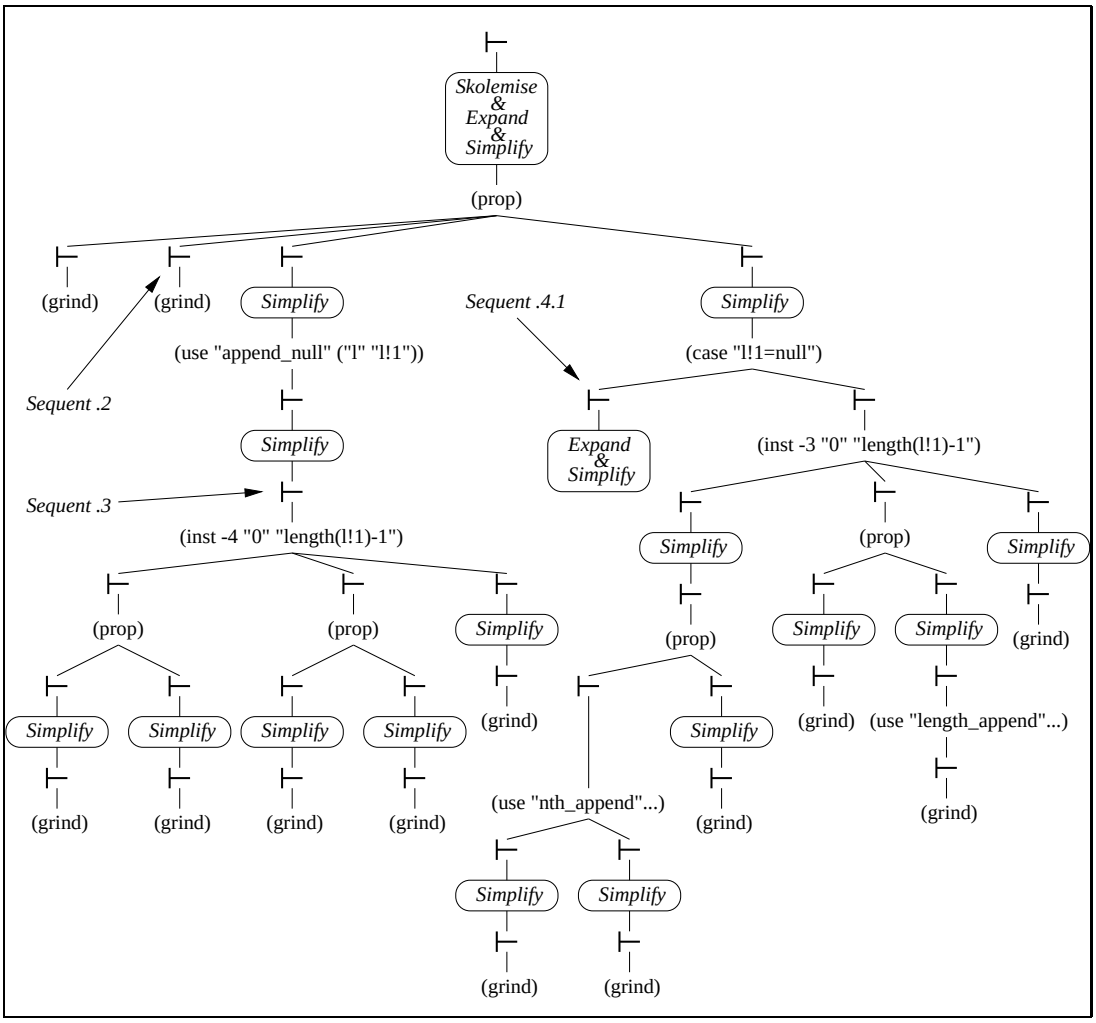

Figure 5.15: Proof for verify_receive_higher

Sequent 5.4.1 (Sequent .3)

[−1]	$\text{append}(l', \text{null}) = l'$
[−2]	$r' = \text{null}$
[−3]	$\text{pr}(\text{nth}(l', \text{length}(l') - 1)) \leq \text{pr}(m')$
{−4}	$\forall(i, j : \{k : \text{int} \mid k \geq 0 \wedge k < \text{length}(\text{queue}(\text{initial}'))\}) :$
	$i \leq j \Rightarrow \text{pr}(\text{nth}(l', i)) \leq \text{pr}(\text{nth}(l', j))$
{−5}	$\text{current}(\text{initial}') = \text{value}(\text{car}(l'))$
{−6}	$\text{pr}(\text{car}(l')) > \text{pr}(m')$
{−7}	$\text{queue}(\text{initial}') = l'$
[−8]	$\text{queue}(\text{final}') = \text{append}(l', \text{cons}(m', \text{null}))$
[−9]	$\text{current}(\text{final}') = \text{value}(\text{car}(\text{append}(l', \text{cons}(m', \text{null}))))$
{1}	$l' = \text{null}$
{2}	$l' = \text{null}$
[3]	$\text{get}(\text{current}(\rho(\text{final}'))) = m'$

From antecedent -6 it is known that message "m′" has higher priority than the head of "l′" (it should be remembered that higher numbers correspond to smaller priorities). From antecedent -3 it is known that the last element of "l′" has higher priority than message "m′". Hence, the last element of "l′" has higher priority than the head of "l′". However, antecedent -4 says that "l′" is sorted by decreasing order of priority. This contradicts the previous conclusion. Instantiating "i" and "j" in -4 with "0" (the index of the head), and "length(l′) − 1" (the index of the last element), it is possible to use this contradiction to solve the sequent.

Following the traditional approach of subdividing a proof in a number of smaller proofs, two auxiliary theorems were proved so that they could be used in the main proof. Both were about lists. The first theorem states that appending to a list will not decrease its size:

Theorem 5.4.2 (length_append)

$\forall\ (l1, l2 :\ \text{list[Message]}) :\ \text{length}(l1) \leq \text{length}(\text{append}(l1, l2))$

The second, that appending to a list will not alter the elements already in the list:

Theorem 5.4.3 (nth_append)

$\forall\ (l1, l2 :\ \text{list[Message]}, i :\ \{j :\ \text{int}\ \mid\ j \geq 0 \wedge j\ <\ \text{length}(l1)\}) :$
$\text{nth}(\text{append}(l1, l2), i) = \text{nth}(l1, i)$

Both theorems can be easily proved by induction. Axiom "append_null", from the prelude theory, is also used.

Note that the proof above is the result of the first successful attempt at proving the theorem. The concern was simply to do the proof using simple commands. By appropriately tailoring the use of the PVS commands, the proof could, almost certainly, be made smaller.

Although the property just proved is a generalisation of that proved manually in (Harrison et al. 1996), some degree of comparison can still be made. Most noticeably, it is clear that PVS proofs are more detailed. No leaps can be done in the reasoning, and every step has to be formally justified. This can become a problem, but since PVS has powerful proof commands, most of the work is done by the prover. On the positive side, once a proof is performed it is easy to use it repeatedly in a completely automated way. The proof presented in (Harrison et al. 1996), on the other hand, is quite concise since some steps are assumed trivial. This however can create problems. As an example, the proof at some point (step 2.2 — see Harrison et al. 1996, page 5) infers a contradiction in a similar way to what was done for Sequent 5.4.1. As was shown above, this contradiction uses the fact that the list is sorted. In the manual proof, however, this is not explicitly stated (no mention of the corresponding invariant is made). This could create problems. Consider a situation where the specification was changed so that the queue is no longer sorted. It will be necessary to check that all the relevant properties of the system are still maintained. Since this particular proof makes no explicit mention of the sorted invariant, this could go overlooked, and the property thought still valid. The theorem prover, on the other hand, would detect the problem automatically.

So, it has been proved that whenever a message with higher priority arrives, it will be put on the display. Looking at the framework in Figure 3.2, the theorem that was proved can be classified as expressing a property over:

$$\rho \circ \text{operation}_{\text{system}}(\text{state}_{\text{system}}) \tag{5.3}$$

That is, the theorem is dealing with the system side of the design only. It was proved that the interface is updated, but nothing has been said regarding the user perception of the interface change. This is an example of where care must be taken in interpreting

what exactly is being proved and what are the implications. In order to prove that the interface is adequate, two aspects have to be considered:

- the *intention* (or goal) behind the interface update;
- how the user perceives it.

Hence, Equation 3.6 (page 82) should be used. Regarding intention, a plausible expectation, in the case under analysis, is that the user might be able to recognise that a new message has arrived. Note that in this context *intention* has a different meaning from that used in Norman's (1988) Gulf of Execution description. There, the user forms an intention and acts upon it, to create an effect on the system state. Here, the starting point is "receive", which is an internal system action. What is being analysed is the effect that the system action will have on the user's mental model of the state.

Regarding perception, the only information at the interface is the message at the top of the queue. In order for users to recognise that a new message has arrived, they must be able to remember the last value of the interface and compare it with the new one. The theorem to be proved should then be:

Theorem 5.4.4 (verify_receive_higher_percept)

$$\begin{array}{l}\forall\ (\text{initial},\ \text{final}:\ \text{attributes},\ \text{m}:\ \text{Message}):\\ \quad(\text{valid}(\text{initial})\wedge\\ \qquad \text{queue}(\text{initial}) \neq \text{null}\wedge\\ \qquad\quad \text{pr}(\text{get}(\text{current}(\text{initial}))) \ >\ \text{pr}(m)\wedge\\ \qquad\qquad \beta(\text{messagedisplay})(\text{receive}(m))(\text{initial}, \text{final})) \supset \rho(\text{final}) \neq \rho(\text{initial})\end{array}$$

i.e., if a message with a higher priority than the current displayed message is received, the presentation will change. It could be argued that the theorem should end with "current(ρ(final)) $\neq$ current(ρ(initial))". However, since the presentation only has the "current" component, in the present case the expressions are equivalent.

Since this new theorem is similar to the previous one, an attempt was made to apply the same proof. This can be done automatically by PVS. The proof attempt was partially successful. Two sequents were left unproved: sequent .2 and sequent .4.1 (see Figure 5.15). Sequent .2 is divided into four sub cases. The first one is:

Sequent 5.4.2 (Sequent .2 – first subgoal)

[−1]	$pr(car(r')) > pr(m)$
{−2}	$null?(l')$
{−3}	$current(initial') = value(car(r'))$
{−4}	$queue(initial') = r'$
{−5}	$queue(final') = cons(m, r')$
{−6}	$current(final') = value(car(r'))$
{−7}	$value(m) = value(car(r'))$
{1}	$null?(r')$

Sequents can be seen as specific interaction situations about which something has to be proved. In this case a situation is being considered where the received message is equal to the head of the queue in the initial state (antecedents -7 and -4), and the priority of the head of the queue is less than that of the received message (antecedent -1). This is a contradiction, and proving the sequent should be trivial. First, however, it is necessary to transform antecedent -7 into "$m = car(r')$". To see if this is valid, it must first be noted that "value" is the function that creates a value of type "optional[Message]" from a value of type "Message". It is clear that "value" should be injective. Since the optional type is used only to add the "nil" value, its use should preserve the meaning of messages. Using this, the sequent can be solved easily. All the remaining sequents (including Sequent .4.1) are solved similarly.

It has now been proved that the interface allows users to perceive when a higher priority message has arrived. Note that a number of assumptions have been made:

- that users can see the messages on the presentation;
- that users can remember messages;
- that users can perceive the differences between different messages.

Deciding whether or not these assumptions are valid will be a starting point for discussion with the designers and human-factors experts.

At the technological level, this second proof illustrates some points (both positive and negative). On the positive side, it shows how proofs can be reused, not only in the more natural situation where a change is made to the specification and all the proofs are rerun to ascertain the soundness of the change, but also when proving a number of similar theorems. On the negative side, it shows how it can happen that

the mathematical *machinery* used in the specification can some times *get in the way* of the proof. In the present case, the way in which the optional type was specified meant that some more work had to be done in the second proof. Although in this case the solution was simple, it does illustrate that the style of specification can impact on how easy/hard the proofs become. Obviously, the fact that models are being used that focus on specific features of the system, helps control this problem. This is because only what is needed and relevant will go into the model.

Verification of the second property

As said above, the property just proved was concerned with how the user perceived the result of an action performed by the system. The next property works in the other direction: the user wants to remove a message, can this goal be achieved, and can the user perceive that the goal has been achieved?

This property relates to the "clear" action. Proving that the action does indeed achieve the removal of the message at the head of the queue would be easy and follows from its definition. A more interesting question is whether the user can perceive that the intended effect has been achieved. Again, this is a property of the type represented by Equation 3.6. As in the previous case this can be accomplished if the user sees a change at the interface. This time, after the "clear" action is executed. This can be expressed by the theorem:

Theorem 5.4.5 (verify_clear)

$$\begin{array}{l}\forall\ (\text{initial},\ \text{final}:\ \text{attributes}):\\ \quad(\text{valid}(\text{initial})\wedge\\ \qquad\text{queue}(\text{initial})\neq\text{null}\wedge\\ \qquad\quad\beta(\text{messagedisplay})(\text{clear})(\text{initial},\text{final}))\supset\rho(\text{final})\neq\rho(\text{initial})\end{array}$$

Attempting this proof, the following sequent is quickly reached:

Sequent 5.4.3

$$\begin{array}{|rl}
\{-1\} & \text{queue}(\text{final}') = \text{cdr}(\text{queue}(\text{initial}'))\\
\{-2\} & \text{current}(\text{final}') = \text{value}(\text{car}(\text{queue}(\text{initial}')))\\
\{-3\} & \forall(i,j:\{k:\text{int}\mid k\geq 0\wedge k<\text{length}(\text{queue}(\text{initial}'))\}):\\
 & \quad i\leq j\Rightarrow\text{pr}(\text{nth}(\text{l}',\ i))\leq\text{pr}(\text{nth}(\text{l}',j))\\
[-4] & \text{current}(\text{initial}') = \text{value}(\text{car}(\text{queue}(\text{initial}')))\\
[-5] & \text{value}(\text{car}(\text{cdr}(\text{queue}(\text{initial}\prime)))) = \text{value}(\text{car}(\text{queue}(\text{initial}\prime)))\\
\hline
\{1\} & \text{null?}(\text{cdr}(\text{queue}(\text{initial}')))\\
[2] & \text{null?}(\text{queue}(\text{initial}'))
\end{array}$$

From the antecedents, can it be concluded that either the initial queue is empty or has only one element?

Antecedents "-1" to "-4" follow directly from the definitions and do not imply the consequents. Antecedent "-5" is more interesting. It states that the first two elements of the initial queue are equal. Although this does not help in solving the sequent, it does indicate what the problem is: if the two messages are equal, then removing one will not cause a change at the interface, since the new current element will be equal to the one just removed. Unless it can be shown that this type of situation will not arise, this highlights a problem with the interface. Unless further feedback is given, under some circumstances the user will not be able to tell whether an attempt to clear the top message has succeeded. It can then be said that the clear action is not fully reactive.

Several solutions could be adopted. A possibility is to enrich the interface by showing also the number of pending messages. The presentation now becomes:

```
presentation : TYPE = [#current : optional[Message],
                       nmesgs : int#]
```

With this change, the proof becomes trivial.

It is interesting to note that the design described in (Duke & Harrison 1994*a*) does actually include the number of messages in the queue as part of the display. This illustrates some points:

- Care must be taken when deciding what to include in the model — this is particularly relevant since the exact implications of the properties which will be verified might not all be obvious before the analysis is actually carried out. What the example shows is that the proof itself might point out incompletenesses in the model. Hence, the model is never a finished product, rather it might evolve with the analysis. In this case what was left out was contributing to the overall quality of design (hence, was identified as missing from the model). A more difficult problem arises when something is left out which might invalidate the design. As pointed out before, the only solution here, is to clearly state all the assumptions that are being made regarding what is (considered) relevant and what is not. These assumptions can then later be discharged.

- The proof helps uncover hidden meanings of interface components — in this case, the number of messages in the queue is useful not only as an indication to the ATCO of the number of messages waiting attention, but might also start being used to help determine state changes in the queue. An obvious advantage of identifying these, thus far, unforeseen meanings, is that they can then be analysed regarding their effectiveness: how good will the number of messages be regarding the determination of state changes? Should alternative feed-back be provided?

Finally, a note on reuse. While the CERD design did include the number of messages still waiting, an ATC design described in (Sage & Johnson 1999) displays the head of the queue only. From the discussion above, it can be concluded that unless some alternative form of feed-back is given, the ATCO will have problems interacting with that component of the ATC system. This shows how analysis can be reused, not only when similar theorems are being proved, but also when similar components are used in different systems/designs. Obviously this does not apply exclusively to automated analysis, but the fact that proofs have already been carried out, and can quickly be repeated provides a great advantage over traditional forms of analysis. Specially when small details which might invalidate the proof can go unnoticed in an informal setting.

5.5 Conclusions

As mentioned in the beginning of the chapter, theorem proving is not as effective as model checking when the problem deals with temporal concepts. However the greater expressive power of theorem provers provides an advantage over model checkers when the problem involves reasoning about the state of the system, or relating different models of the system. This greater expressive power means greater flexibility, which, in turn, means that theorem provers will not be used in rigid ways as model checkers are.

Two examples of application of theorem proving were described. The first example dealt with the analysis of the interface design. Through the verification process, three

significant changes to the specification were identified, in the form of assumptions about the system. Each of these assumptions was derived directly from the information provided by the proof process. Two of these highlighted aspects of the functional level that needed to be better represented at the perceptual level:

- the extent of the salmon bug had to be explicitly related to the safe speed margin, for the perceptual operator *configBugCheck* to work properly,
- the relationship between bugs and current configuration had to be made explicit for the configuration change task to be supported.

The third brought out the implications of some of the assumptions that were being made about the interface and showed that those issues had not been included in the functional model. In summary, checking for consistency between an abstract specification of the functional level of the system and the presentation for that same system, allowed for a better understanding of the issues involved in the design of the interface. This understanding was based on two main classes of issues:

- what assumptions were being made (implicitly) about how the interface represents the underlying system;
- what assumptions were being made (implicitly) about how the user will perceive the interface.

The proof process allowed for exploration of how assumptions made at one level will impact the other level, and about how users will be likely to start finding *hidden* meaning in the interface.

The second example dealt with investigating the effect of specific actions both on the system state, and on the system interface. This showed how theorem proving can be used with advantage to reason about behaviour when such behaviour does not involve quantification over traces of events, and a greater insight onto the effect of actions in the state is sought. Additionally the importance of taking the user into consideration when defining the properties that should be checked was also addressed. Overall, the chapter shows how, more than giving an absolute measure of quality, verification can be used to raise issues and generally help reasoning about the system design.

Although the aim of the chapter was to address the use of theorem proving in general, in the end a particular tool had to be used. Regarding PVS, although these proofs are not so complex that it would be infeasible to do them by hand, the prover was found to be helpful in two ways:

- PVS has a number of powerful proving commands that, most of the time, can save a lot of time and patience — as more and more concrete specifications of the perceptual level are used, so the theorem prover will become more and more useful in this regard
- at a different level, by being *totally impartial*, the theorem prover better exposes assumptions made about the system. Had the verification been performed by hand, some of those assumptions might have crept into the proof unnoticed.

The thesis has now covered the role that formal verification might have on interactive systems development, and the use of model checking and theorem proving have been addressed. This, however, has been made in isolation. The next chapter uses a case study to show how both techniques can be integrated into a coherent verification process.

Chapter 6

ECOM: A Case Study

The previous chapters have proposed a closer integration of verification in the development process of interactive systems. The use of model checking and theorem proving have been discussed. This chapter presents a case study where both techniques are brought together. An initial report on this case study was presented in (Campos & Harrison 1999*b*).

6.1 Introduction

This Chapter describes a case study in the use of automated reasoning to aid the formal analysis of a proposed design of an interactive system. The system under analysis is ECOM, a digital audio-visual communication system developed as part of the EuroCODE project. EuroCODE was Esprit Project 6155, aimed at the development of an open development environment for Computer Supported Cooperative Work (CSCW)[1]. The case study was performed with a number of objectives in mind:

- analyse how the verification techniques discussed in previous chapters behave in a *real life* interactive system design context;

[1] `http://www.research.ec.org/esp-syn/text/6155.html` (last accessed on the 28th of August, 1999).

- analyse how the best verification technique to apply in each concrete verification situation might be identified, and how to identify what is relevant for the models that have to be built;

- more generally, to analyse how automated reasoning results can be used to bridge the gap between the fields of software engineering and cognitive analysis, in the context of the interactive part of the system.

The chapter will illustrate how human-factors issues can be used to generate properties of interest. Models are developed that reflect a view of the system with emphasis on the property under consideration. These models are analysed using either SMV, the model checker, or PVS, the theorem prover. The chapter then points out how the results of such analysis can be fed back into a human-factors context for cognitive analysis.

6.2 The case study

As said above, the case study consists in analysing some aspects of ECOM, an audio-visual (AV) communication system. ECOM has already been used as a case study in interactive systems analysis in (Bellotti et al. 1995, Bellotti et al. 1996) and, more recently, (Duke et al. 1999). There, three main categories of analysis techniques were used:

- formal system modelling — interactors were used to build a formal model of the system, which then was informally analysed;

- architectural analysis — PAC-AMODEUS (Nigay & Coutaz 1993) was used to model and analyse the software architecture of the system;

- user modelling — Programmable User Models (Young et al. 1989) (see also Butterworth & Blandford 1997, for a review on PUM), and Cognitive Task Analysis (Barnard 1987) were used to perform usability analysis from a cognitive perspective.

This chapter builds on the work of Bellotti et al. (1996), showing how automated reasoning could have been used in the context of the formal system modelling approach. The result will be compared also with the results from the user modelling approach. The PAC-AMODEUS analysis, being architecture oriented, is not so much interested in usability issues, but more in implementation issues, hence will not be addressed. However, it will be shown how architecture related concerns also emerge.

The main aim of AV communication systems is to enhance collaboration between a community of users distributed over a number of distinct physical locations. This is done by offering a number of means of contact between users. These will include audio, video, and any other form of exchange of information that might be judged appropriate for a specific system. A key factor in the establishment of a good collaboration environment is the promotion of mutual awareness between the users (Gaver et al. 1992). To this end, AV systems provide mechanisms to allow users to be aware of where other users are and of what they are doing. These mechanisms are provided to help users to determine how receptive other users might be to contacts: their availability. Ideally, then, each user should be able to see every other user in the system as if they where in the same physical space. This, however, is unacceptable in the face of another design criterion: privacy. As Mantei et al. (1991) point out, it is essential that individual users are provided with appropriate mechanisms to allow control over who can contact them, and how: in other words, their accessibility. Hence, the designer of this type of system is presented with a tension between the need to promote a sense of awareness between users, and the need to preserve individual privacy.

One of the features that makes ECOM interesting from a design analysis point of view is the attempt at integrating features from two previous AV systems: CAVECAT (see Mantei et al. 1991, for an initial report) and RAVE (Gaver et al. 1992). In CAVECAT (as reported by Bellotti et al. 1996) accessibilities are represented by a door state. There are four such door states (open, ajar, closed, and locked), each representing a different accessibility level, with its associated set of allowed connections. Users can set an appropriate door state and can see the door states of other users in the system. In this way users can set an appropriate level of accessibility. If, for example, some user is in an important meeting, and does not want to be interrupted, the user

can *lock the door* thus barring all incoming connections. At the same time users can have a notion of how accessible other users are, by looking at each other's door state.

In RAVE, accessibilities are set on a per caller basis. Users can select a specific type of connection and specify which users are allowed to establish it. Awareness of each user's availability is promoted by a panel showing snapshots of all users which are periodically updated. The idea being that these snapshot will give an idea of how busy (or not) a user is, and hence how receptive to connections.

CAVECAT allows for an easy change in the accessibility level. However it does not include the possibility of exceptions to the general setting: you might want the meeting not to be interrupted *unless* it is the boss! On the other hand, RAVE allows a better tailoring of the accessibility setting, but makes it harder to make a global change since the accessibility setting of every user will have to be updated manually.

The designers of ECOM decided to attempt the integration of both mechanisms (see Figure 6.1). Users can set a general accessibility level using the door state metaphor, but a mechanism for exception setting is introduced that enables them to give different accessibility rights to specific users. Setting exceptions is done by selecting a user, a specific type of connection and the door state for which the user will have permission to establish the connection (i.e. the exception level — see the exceptions panel in Figure 6.1). One early requirement was that allowing a connection for a specific door state, should automatically allow it in all door states that are more open. Hence, if the exception level is set to "when door closed", the connection will also be allowed when the door is ajar or open. To allow for unrestricted access, and for complete blockage of connections, the exception level can also be set to "always" or "never". As in CAVECAT, awareness of user accessibility is promoted by presenting the door state of each user to all other users.

By merging both designs, the designers hope was to improve on the previous systems by incorporating the best features of each. This merging raises the question of how well the two mechanisms will work together.

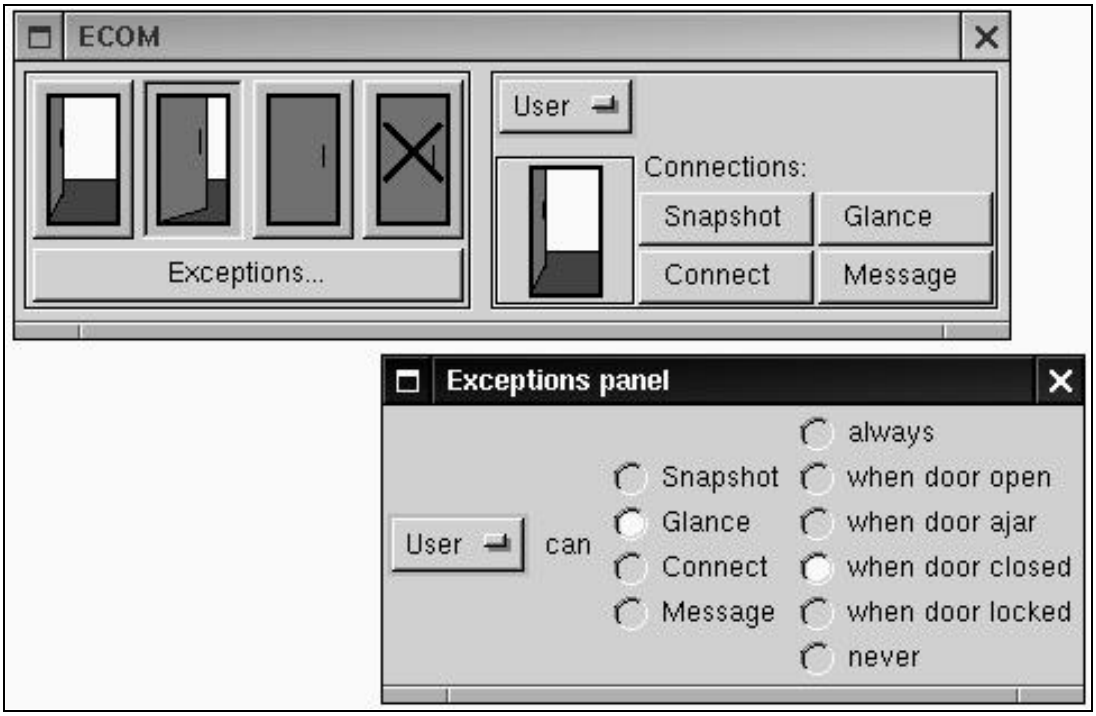

Figure 6.1: The ECOM system (adapted from Bellotti et al. 1996)

6.3 Checking the proposed design

As discussed in Chapter 3, the verification process is triggered by the identification of potentially interesting features of the system that need looking into. What features are selected has to do mainly with the characteristics of the system, and with what are the main concerns that its use raises. Automated reasoning can then be seen as one more tool that is available to help analyse suitable aspects of the system. In the present case, some of the decisions that were made regarding what properties to check were informed by the concerns regarding the system design reported in (Bellotti et al. 1996). Since the purpose of the case study is to analyse the use of automated reasoning, this has been taken into consideration when selecting the properties of interest.

Initially, three particular aspects of the system were selected. First an issue of predictability. Being a general principle, predictability cannot be used as an absolute design criterion. There are even many situations were it will be undesirable or simply impossible to attain. There are however situations where predictability is desirable and in these cases the design should be such as to maximize it. An angle which is commonly

taken regarding predictability is: "the user should understand from the interface if an operation they might like to invoke cannot be performed" (Dix et al. 1993, page 133). In the present case, this angle is relevant due to at least two factors. One of the goals of the designers is to promote a sense of awareness of how available other users are in the system: if the mechanism to establish connections is not predictable, then users will lose faith in the state they perceive the system to be in, thus defeating awareness (cf. Woods et al. 1994). Also, as pointed out by Dix et al. (1993), predictability has an impact on learnability: if users cannot predict how the system will react to specific command, they will obviously find it harder to learn how to operate the system. Hence, the design will be evaluated against the following criterion:

> Can a user predict whether attempting a connection will succeed?

In (Bellotti et al. 1996) the importance of cumulative exception setting is stressed by the user modelling analysis. An interesting question, then, is whether this cumulative nature of exceptions is conveyed by the proposed design. This concern leads to the definition of the second criterion for analysis:

> Does the user interface appropriately convey the cumulative nature of the exception levels?

Finally, the issue of redundancy at the interface was also addressed. As a general principle, designers should strive to reduce the user's cognitive load and maintain clarity at the interface (cf. Preece et al. 1994). The presence of redundancy can clearly defeat both these aims. In addition, as Woods et al. (1994) point out, designers should aim at reducing the number of modes in the system. At the user interface, modes have to do with the mapping between input/output and its interpretation by the system/user (cf. Norman 1988, page 110). The same command will produce different outcomes depending on system mode, and conversely, the some output will have different meaning depending also on system mode. The different levels of exception setting can be seen as different modes regarding the decision on whether a connection should be allowed, since different exception levels will induce different results from the action of attempting a connection. According to Woods et al. (1994) then, the design

should avoid unnecessary or redundant exception levels (i.e. modes). In the present context, it will be investigated whether there is any real need to have the "always" and "never" exception levels. Two additional criteria for analysis are defined:

> The effect of setting the exception level to "always" cannot be achieved by setting it to some other level.

and also:

> The effect of setting the exception level to "never" cannot be achieved by setting it to some other level.

In the remainder of this chapter the proposed design will be assessed against the criteria enumerated above. To do that, models that focus on those aspects of the system that are most relevant to the principles being analysed will be developed (cf. Fields et al. 1997, Campos & Harrison 1998). Note that this is yet another instance of the general process of model building: abstracting away what is accessory, and focusing on what is relevant. As with any other model, care must be taken that all which is relevant is conveniently included.

At each stage, the models are developed bearing in mind the particular verification technique that seems more appropriate to the criterion under consideration.

6.3.1 Analysing predictability

The first criterion will now be looked at:

> Can a user predict whether attempting a connection will succeed?

First an appropriate model of the proposed system must be built. To do that, those aspects of the system which relate to the property under consideration must be identified. In the present case the model must include:

- The mechanism for accessibility setting (including exceptions) — this has an impact on which connections can be established.

- The mechanism for establishing connections — this is what it is being analysed.
- The mechanism for promoting awareness — this gives users information about the availability of other users; hence, what connections they are willing to accept.

The model

The model is loosely based on one presented in (Bellotti et al. 1996). It will consider a user panel representing the interface to a single user, and the system core representing the remainder of the system. The user accesses the system core through the user panel.

The interactor language described in Chapter 3 will be used to write the model. First, some types are introduced:

types

User	# all users
Service	# available services
Door	# door states
Exception	# levels of exception
$Conn = User \times Service \times User$	# connections

The type names should be self explanatory. Services represent possible connection types, and connections are defined as tuples built with two users (the caller and the callee), and a service (the type of connection).

The system core model is presented in Figure 6.2. Its main task is to manage connections. At each instant there are a number of connections in progress (attribute *current*). Two actions manipulate *current*: action *establish* initiates a given connection, and action *close* ends it. Attribute *accessibility* associates, with each door state, the valid services for that state. Each exception level is associated with a set of door states (attribute *exceplevel*). Connections can be associated with an exception level (attribute *accessexcep*) hence becoming valid in all the corresponding door states. The door state of each user is defined by attribute *availability*.

To simplify reading the model, a further attribute is used which represents the allowed connections at each moment (attribute *allowed*). The process of determining if a connection is allowed is specified by Axiom 7. It mimics the reasoning just described. Since attribute *allowed* is fully defined by the invariant in Axiom 7, it need not be

interactor *core*
attributes
$current : \mathbb{P} Conn$
$allowed : \mathbb{P} Conn$
$availability : User \rightarrow Door$
$accessexcep : Conn \nrightarrow Exception$
$accessibility : Door \rightarrow \mathbb{P} Service$
$exceplevel : Exception \rightarrow \mathbb{P} Door$
actions
$establish(Conn)$ $close(Conn)$
$setavail(User, Door)$ $setexcep(User, Service, User, Exception)$
axioms
(1) $[establish(c)]\ current' = current \cup \{c\} \wedge \text{nochg}(availability, accessexcep)$
(2) $\text{per}(establish(c)) \rightarrow c \in allowed$
(3) $[close(c)]\ current' = current - \{c\} \wedge \text{nochg}(availability, accessexcep)$
(4) $\text{per}(close(c)) \rightarrow c \in current$
(5) $[setexcep(o, s, t, e)]\ accessexcep' = accessexcep + [(t, s, o) \rightarrow d] \wedge$
$\text{nochg}(current, availability)$
(6) $[setavail(u, d)]\ availability' = availability + [u \rightarrow d] \wedge \text{nochg}(current, accessexcep)$
(7) $(caller, s, callee) \in allowed \leftrightarrow$
$(((caller, s, callee) \in dom(accessexcep) \rightarrow$
$availability(callee) \in exceplevel(accessexcep((caller, type, callee))))$
$\wedge$
$((caller, s, callee) \notin dom(accessexcep) \rightarrow s \in accessibility(availability(callee)))$
)
(8) $s \in accessibility(d) \leftrightarrow ...$
(9) $d \in exceplevel(e) \leftrightarrow ...$

Figure 6.2: Core

included in the modal axioms. Similarly, attributes *accessibility* and *exceplevel* will be defined by invariants in Axioms 8 and 9. At this stage these axioms have not been written since concrete values for doors and exception levels have not yet been defined. Two more actions are considered: action *setexcep* introduces an exception into the system, and action *setavail* sets the availability of a given user.

The user panel is built on top of the core, its model is presented in figure 6.3. Since the analysis is looking at a situation where one user is trying to establish a connection, the model of the user panel is developed only so far as to make that analysis possible. Hence, the model includes the buttons that are used to request connections (attribute *buttons*), the user that has been selected as callee (attribute *callee*), and its door state (attribute *door_icon*). The callee is set by action *select*. Finally, the user panel has an owner, the caller (attribute *owner*). The axioms should be self explanatory. Axiom 2 defines that when a button is pressed ($buttons(s).\ >_{\texttt{action}} = pressed$)

interactor *userpanel*
importing *core*
attributes
owner : *User*
vis *callee* : *User*
vis *buttons* : *Service* → *button*
vis *door_icon* : *Door* # door state of selected user
actions
vis *select*(*User*)
axioms
(1) $door\_icon = availability(callee)$
(2) $(buttons(s).>_{action}= pressed \wedge (owner, s, callee) \in allowed) \rightarrow$
$>_{action}= establish((owner, s, callee))$
(3) $[select(u)]\ callee = u \wedge \text{nochg}(current[], availability[], accessexcep[])$
(4) $[establish(caller, s, cal)]\ \text{nochg}(callee)$
(5) $[close(caller, s, cal)]\ \text{nochg}(callee)$
(6) $[setavail(u, d)]\ \text{nochg}(callee)$
(7) $[setexcep(o, s, t, e)]\ \text{nochg}(callee)$

Figure 6.3: User Panel

a request to establish the corresponding connection should be originated ($>_{action}=$ $establish((owner, s, callee))$). Note, however, that this is only true if the precondition to *establish* (dictated by Axiom 2 in *core*) is verified. This is important since only legal calls to the underlying functionality should be allowed (Campos & Martins 1996).

The analysis

In order to verify the proposed design against the criterion set forth above, that criterion must first be formalised. Looking at the user panel, it can be seen that the user gets information on the callee's availability through its door state. It will be assumed that the user can *remember* which connections are valid for each door state. Determining whether or not this is an ideal/feasible solution is much more the field of cognitive science than that of software engineering. An approach such as Cognitive Task Analysis could be used in this case. It could be argued that this information should be encoded in the interface by disabling *illegal* buttons for the given door state, thus reducing the cognitive load on the user side. As it will be shown this is not an ideal solution: determining the meaning of the buttons runs deeper than a direct association between door states and button states.

If it is assumed that users will remember, and only try, valid connections, then

for the system to be predictable all valid button presses for each door state result in the corresponding connection being established. This type of causal effect can be best expressed in a temporal framework. The particular property above can be expressed as the CTL formulae defined by the following quantified expression:

$$\forall_{d \in Door, s \in accessibility(d)} \bullet$$
$$AG(door\_icon = d \rightarrow AX(buttons(s).\ >_{\texttt{action}} = press \rightarrow$$
$$(owner, s, callee) \in current))$$

This type of formula is best verified using model checking. In the present case SMV, and the compiler introduced in Chapter 4 will be used to perform the verification.

In order to be model checkable a specification must be expressible as a finite state machine, ideally with a minimum of states. For that to be true of the present ECOM model, the model must first be completed. The first step is to make all types finite. From Figure 6.1 it can be seen that there are four possible connection types (snapshot, glance, connect, and message), and also four possible door states (open, ajar, closed, and locked). Exceptions can be set for six different levels (always, when door open, when door ajar, when door closed, when door locked, never). The number of users in the system is arbitrary, three users were initially considered. The types become:

types
$User = \{user1, user2, user3\}$
$Service = \{message, snapshot, glance, connect\}$
$Door = \{open, ajar, closed, locked\}$
$Exception = \{always, never, whendooropen, whendoorajar, whendoorclosed, whendoorlocked\}$

Since all other types are defined on top of these three, they become finite by definition.

Next, Axioms 8 and 9 must be defined. This is because the property under consideration deals with establishing connections, and these two axioms have impact on that. It will be assumed that messages can always happen, snapshots can happen unless the door is locked, glances if the door is open or ajar, and connects only if the door is open. With these assumptions, Axiom 8 can be substituted by the following axioms:

(8a) $accessibility(open) = \{message, snapshot, glance, connect\}$
(8b) $accessibility(ajar) = \{message, snapshot, glance\}$

$(8c)$ $accessibility(closed)=\{message, snapshot\}$
$(8d)$ $accessibility(locked)=\{message\}$

Axiom 9 defines the exception levels. It can be substituted by the following axioms (note that they preserve the cumulative nature of exceptions):

$(9a)$ $exceplevel(always)=\{open, ajar, closed, locked\}$
$(9b)$ $exceplevel(never)=\{\}$
$(9c)$ $exceplevel(whendooropen)=\{open\}$
$(9d)$ $exceplevel(whendoorajar)=\{open, ajar\}$
$(9e)$ $exceplevel(whendoorclosed)=\{open, ajar, closed\}$
$(9f)$ $exceplevel(whendoorlocked)=\{open, ajar, closed, locked\}$

The model is now complete. In order to analyse it with SMV, however, all structured types have to be rewritten as arrays. Type $\mathbb{P}$ Conn, for example, becomes:

$PConn$ = array $user1$ .. $user3$ of
array $snapshot$ .. $message$ of array $user1$ .. $user3$ of boolean

Since the array indices must be integer values, $user1$ to $user3$ are defined as the values 1 to 3 using **define** clauses. The same process is applied to most other structured types present in the specification. Attribute *buttons* is substituted by a number of attributes, one for each service available.

The specification can now be rewritten to take into consideration these concrete definitions of the types. As an example, the new version of Axiom 7 in interactor *core* is shown next (compare with Figure 6.2):

$$
\begin{array}{l}
allowed[caller][s][callee] \leftrightarrow \\
\quad ((accessexcep[caller][s][callee] = \text{null} \rightarrow accessibility[availability[callee]][s]) \\
\quad\ \wedge \\
\quad\ (accessexcep[caller][s][callee] \neq \text{null} \rightarrow \\
\quad\quad exceplevel[accessexcep[caller][s][callee]][availability[callee]]) \\
\quad)
\end{array}
$$

To make sure the temporal property above holds, it must be verified for all possible users, and all types of connections that are valid for each door state. If it fails for any given combination, then clearly the property does not hold. A problem now arises. Attempting to model check the current specification, it quickly becomes apparent, from the runtime information given by SMV, that the generated state space is too big

for an answer to be expected in a reasonable amount of time. The invariant axioms, in particular, will leave SMV calculating for days.

There is a need, then, to reduce the complexity of the model. Ultimately what is needed is a new (smaller) model which captures all the relevant behaviour of the old model, regarding the property under analysis, but ignores irrelevant information (behaviour and attributes which have no impact on the property being considered). This poses two problems:

- how to deduce a new model, representing a subset of the original one;
- how to guarantee that the new model captures all the relevant aspects of the original one — i.e. how to simplify the model without removing the very problems the analysis is supposed to detect.

Reducing a model to an acceptable size is done by the use of abstractions. That is, simplifications of the original model that preserve intended features. As a general guideline, simplifications made to the model should preserve all possible behaviours of the original model (possibly adding more behaviour). Typically this will be done by using nondeterminism. However, simplifications can also be used to eliminate irrelevant behaviour.

Dwyer et al. (1997) identify three classes of abstractions that can be applied to user interface models. Although they are proposed as abstractions for the user interface layer only, their use can easily be generalised. Additional abstractions can be identified for different domains. Four classes will be proposed here as possible sources of simplifications:

- attribute elimination (1) — whenever two or more attributes are/can be treated as a single entity, they can be substituted by a single attribute representing their composite value (cf. Dwyer et al.'s grouped fields). This type of abstraction has already been used to represent the information transfer in a connection simply as a value indicating the type of connection (with no mention to the actual information being passed).

- attribute elimination (2) — by focusing the analysis on a specific subset of features of the system, it might happen that some attributes will behave as constants. These attributes can be eliminated by hard-coding them into the model. In the current case, attributes *accessibility*, and *exceplevel* were eliminated by *hard-coding* their definition in Axiom (5). This is valid since changes in those attributes are not being considered (i.e. they are thought of as constants).

- noninterference — when a particular feature of the system does not affect the property under analysis, then it can be removed (cf. Dwyer et al.'s independent subsystems). Here the term "feature" is being used loosely. A possible application of this type of simplification to the current case is to note that no third user can influence a connection between two other users. This means that when looking at properties of connections, only two users need to be considered.

- type reduction — on many occasions it is possible to reduce types to a subset of meaningful values (cf. Dwyer et al.'s application state). When modelling a queue, for example, it might be enough to model whether the queue is empty or not. This type of simplification will be heavily dependent on both the system being modelled and the type of property being analysed. In the current case the number of services and of door states was reduced to two. This is valid since the precise number of service types/doors is not relevant (assuming they are both greater than one).

These are only broad categories of possible simplifications. What will be possible in each concrete case will depend on the domain of analysis. It should be stressed that these simplifications must be done carefully, in order not to affect the meaning of the specification. More specifically, the simplified version of the specification must preserve all the behaviour of the original specification regarding the property being checked.

It is, then, necessary to guarantee that all relevant behaviour is included (or, at least, to defend against the possibility of not having done so). To this end, note that in all of the simplifications introduced above, something was being assumed about the behaviour of the original model. The simplifications (hence, the verification results) are only valid if those assumptions hold. Whether they hold, or not, becomes material

for further analysis. The verification, then, progresses in an assumption-commitment style.

At a more technological level, performance can be improved by maximising the effectiveness of the symbolic algorithm of the model checker. In particular, the order in which each of the variables are used in the construction of the BDD has a dramatic impact on the performance of SMV.

With the alterations above the property becomes checkable in useful time. When the following instance of the property is tried:

$$\texttt{AG}(\texttt{door\_icon} = \texttt{open} \rightarrow \texttt{AX}(\texttt{button\_connect}.>_{\texttt{action}} = \texttt{press} \rightarrow (\texttt{owner}, \texttt{connect}, \texttt{callee}) \in \texttt{current}))$$

SMV replies with[2]:

```
-- specification AG (door_icon = 1 -> AX (buttons_connect... is false
-- as demonstrated by the following execution sequence
state 1.1 :
action = nil
allowed[user1][connect][user2] = true
current[user1][connect][user2] = false
availability[user2] = open
owner = user1
door_icon = open
callee = user2

state 1.2 :
action = setavail(user2, locked)
allowed[user1][connect][user2] = false
current[user1][connect][user2] = false
availability[user2] = locked
buttons_connect.action = press
door_icon = locked

resources used :
user time : 292.7 s, system time : 0.31 s
BDD nodes allocated : 783782
Bytes allocated : 14745600
BDD nodes representing transition relation : 3911 + 892
```

It can be seen that the property does not hold. What the counter example shows is that callees might change their availability between the time a caller looks at the callee's

[2] After some editing for readability.

door state and the time the button is actually pressed. This is a somewhat obvious problem, it is interesting to notice, however, that none of the approaches in (Bellotti et al. 1996) raised this issue. Naturally, predictability is not an absolute measure of quality. So, whether this is a real problem or not will be matter for discussion. A possible suggestion to obviate this problem is to have some sort of delay between the moment the availability is changed (more precisely, shown to have changed) and the moment it actually becomes effective. Note that if there is a variable lag in the broadcast of changes to all users in the system, then the amount of time it will take for a change in availability to become effective will itself be unpredictable. This might have an impact on usability on the callee's side, specially if the lag is too high. At this stage some time based analysis of the architecture proposed for the implementation of the system would be useful.

In order for the analysis to progress at the current level, it must be assumed that the above problem will not happen. This can be done by integrating the problem situation into the property being checked. In this way, properties become scenarios of usage where conditions for a property to hold are identified. The new formulation is:

$$\begin{array}{l}\texttt{AG}(\texttt{door\_icon} = \texttt{open} \rightarrow \\ \quad \texttt{AX}((\texttt{button\_connect}.>_{\texttt{action}} = \texttt{press} \wedge >_{\texttt{action}} \neq \texttt{setavail}(\texttt{callee}, \texttt{d})) \rightarrow \\ \quad\quad (\texttt{owner}, \texttt{connect}, \texttt{callee}) \in \texttt{current}))\end{array}$$

When this property is tried, again the test fails. This time the problem state is:

$$\begin{array}{l}\texttt{state 2.2 :} \\ \texttt{action} = \texttt{setexcep}(\texttt{user2}, \texttt{connect}, \texttt{user1}, \texttt{never}) \\ \texttt{allowed}[\texttt{user1}][\texttt{connect}][\texttt{user2}] = \texttt{false} \\ \texttt{accessexcep}[\texttt{user1}][\texttt{connect}][\texttt{user2}] = \texttt{never} \\ \texttt{buttons\_connect.action} = \texttt{press}\end{array}$$

What this shows is that the callee might set an exception for the particular connection being tried. If this is done, the connection becomes "not allowed". Unfortunately the caller has access only to the callee door state (see visibility annotations in the specification), so the user is unable to predict whether a connection is going to be accepted or not.

The analysis above tells us that the system is not predictable. The user can not tell whether a request for connection will be successful or not. From the user's point

of view, this happens because the information that is displayed regarding the callee's receptiveness to connections is inappropriate. Since exceptions can override the accessibility level defined by the door state, they break the tight coupling between availability and accessibility that existed in CAVECAT. In this way, the door state (which represents the callee's general availability), no longer is an accurate representation of the accessibility of the callee regarding a prospective caller. As pointed out in (Bellotti et al. 1996) this is an instance of a break in conformance: the display does not accurately reflect the state of the system. Note how testing the design against a specific property has allowed the detection of a problem relating to a design principle.

From a cognitive standpoint, the above result has at least two implications. Since callers cannot predict whether a connection will be accepted by looking at the callee's door, they will loose faith in the door metaphor as a representation of the callee's availability. In an extreme situation the door becomes meaningless (obviously this will depend on how frequently users set exceptions, this could be the subject of a user study). Additionally, users will have difficulty in determining what connections each door state allows or denies, so they will have difficulty when trying to set their own availability level (cf. Bellotti et al. 1996). In this way, the combination of both mechanisms seems to hinder the user's capacity to set an appropriate level of availability.

In order to illustrate the above analysis, the following property can be used to demonstrated the difficulty users will have in assessing the effect of setting a particular door state:

$$\texttt{AG}(\texttt{setavail}(\texttt{user2}, \texttt{locked}) \rightarrow \texttt{AX}((\texttt{user1}, \texttt{connect}, \texttt{user2}) \notin \texttt{current}))$$

What the formula expresses is that locking the door prohibits connections of type *connect*. When the formula is tested in the model checker, it fails. However it does not fail due to any breakup between availability and accessibility as might have been expected. Instead, the counter example given by the model checker points out that the property is not true when a connection is already in progress at the time the door is locked: where a *connect* is already established, locking the door will not end that connection. This means that even by locking the door, a user will not be assured privacy. In order to see if this is acceptable behaviour, the precise characteristic of

each connection will have to be considered. For instance, how aware will the user be of the current connections? If the current situation is considered to be unacceptable, the model will have to be revised so that connections are terminated whenever they become invalidated by a change in the accessibility/availability.

With the model, as specified, the property must be revised to:

$$\texttt{AG}(\texttt{setavail}(\texttt{user2}, \texttt{locked}) \wedge (\texttt{user1}, \texttt{connect}, \texttt{user2}) \notin \texttt{current}) \rightarrow \texttt{AX}((\texttt{user1}, \texttt{connect}, \texttt{user2}) \notin \texttt{current}))$$

It still fails. This time the counter example shows that the user might have set an exception unconditionally allowing the connect. This corroborates the point made above that users will have difficulty in predicting what the effect of setting a particular door will be.

Going back to the initial problem of determining which connections are valid, what should be presented to the caller is not the general accessibility level of the callee, but the result of applying the exceptions regarding the caller to the callee's accessibility level: this could be done, for instance, by disabling inappropriate buttons. Note that this is different from what was previously considered: disabling buttons to reflect the accessibility conveyed by the door state. It could even be argued that the general accessibility level of the callee should not be displayed at all, so as to avoid callers detecting that they were being in some way *segregated*. This however still does not solve the problem of predicting what each door really means.

The problem is that two mechanisms with different philosophies are being integrated. From CAVECAT, ECOM inherited a tight coupling between availability and accessibility. Door states being used to represent both. From RAVE, ECOM inherited a mechanism for accessibility setting which is not directly coupled to availability. Determining what is the best compromise solution in the integration of both mechanisms falls outside the scope of formal/automated verification alone. What the analysis offers, is a way to study the different proposals against specific criteria of quality.

Going back to SMV's answer, and looking at it from the specification side, it can be seen that the property fails because there is a mismatch between the precondition of the core level action that establishes the connection (action *establish*), and the preconditions (in the user's head) of the user interface commands that trigger that action

(the buttons). Again, this shows how the checking process can reveal inconsistencies between the software engineering level (concerned with building the system) and the human-factors level (concerned with how users use the system).

State exploration type properties demand that the system be reduced to a finite state machine. It is clear that, as the specifications grow in complexity, this becomes increasingly hard. Moreover, even if the system can be expressed as a finite state machine, it can also happen that this machine is too big and model checking is not feasible. Checking for the satisfaction of preconditions, on the other hand, can be done by hand or using a theorem prover. It is easy to see that a necessary condition for a user interface to be predictable is that the preconditions at the two levels above match. Although this is an indirect test, it still allows for the detection of unpredictability in a system. Hence, even if the specification is too complex for model checking, it can still be analysed regarding predictability, by verifying if all preconditions at the user interface level match the corresponding system level ones. Similarly one can think of checking whether the result of system level actions matches what the user expects. This illustrates the type of interplay that takes place between properties, models, and tools.

6.3.2 Analysing exception setting

The second analysis criterion deals with exceptions, and how they are set. Axioms (7a) to (7f) of the model developed in the previous section, show that, for all exception levels, if a door is in that level, so are all door settings that are more liberal. Hence, the cumulative nature of exception levels is expressed in that model. This however, is not enough to guarantee the benefits of this feature of the system. In order for it to be useful to the user, the interface needs to adequately convey it (Doherty & Harrison 1997).

Since the cumulative nature of exception levels is important mainly in helping users determining what the effect of setting an exception will be, the criteria set forth initially can be further refined to:

> Can the user use the cumulative nature of exceptions to predict whether a connection will be allowed when setting an exception for it?

In order to analyse this, the approach described in Chapter 5, for analysis of the Gulf of Evaluation, can be used. A model of the proposed interface presentation will be developed, along with a function (ρ) which builds a presentation from the state of the abstract specification. Then, two operators capturing the relevant concepts under analysis, one for each model, are defined. The model will consider how a user determines if a connection is allowed from its current exception setting. The presentation will be appropriate if using the operator at the abstract level yields the same result as mapping the abstract state into the presentation and using the operator defined in the presentation level. Since this analysis has to do with relating two models of the system state, and not directly with behaviour. The analysis will be performed in PVS.

From the model developed in the previous section, it can be deduced that, in the presence of an exception, what influences whether a connection will succeed are the availability and exception setting of the callee. The model will thus be refined from the point of view of a potential callee setting an exception for a specific caller and service. Since the caller and connection service are fixed in advance, it becomes possible to factor them out. A model considering only the callee's relevant attributes can start to be defined in PVS. Since the aim of the chapter is to discuss verification, not specification, the specifications will not be presented in full here. The full versions of all the PVS theories used in the case study can be found in Appendix F.

```
restrictedpanel : THEORY
  BEGIN
  door : TYPE  = {open, ajar, closed, locked}
  exception : TYPE  = {never, always, whendooropen,
                       whendoorajar, whendoorclosed, whendoorlocked}
  attributes : TYPE  = [# availability : door, accessexcep : exception#]
  exceplevel : [exception → setof[door]] =
    λ (e : exception) :
      COND
      e = never → {d : door | FALSE},
      e = always → {d : door | TRUE},
      e = whendooropen → {d : door | d = open},
      e = whendoorajar → {d : door | d = open ∨ d = ajar},
      e = whendoorclosed → {d : door | d = open ∨ d = ajar ∨ d = closed},
      e = whendoorlocked → {d : door | d = open ∨ d = ajar ∨ d = closed ∨ d = locked}
      ENDCOND
  ...
  END restrictedpanel
```

Types "door" and "exceptions" are the same as previously. Similarly, operator "exce-

plevel" is also included in this model (compare with Axioms (7a) through (7f)). As said, since it is assumed that an exception is being set, the accessibility definition is not needed — the exception overrides it.

In the presence of an exception, whether or not a request for establishing the connection will succeed can be modeled by the following operator, which is added to the theory above:

```
willsucceed : [attributes → bool] =
  λ (callee : attributes) : exceplevel(accessexcep(callee))(availability(callee))
```

This definition is easily deduced from axiom 5 of *core* (remember, it is considered that the user has set an exception).

Next, a model of the actual interface that is being proposed for the system must be built. Again, the model needs only consider the callee's availability and the caller's exception setting. The presentation is formed by two radio box panels (see figure 6.1 on page 193):

```
ρrestrictedpanel : THEORY
  BEGIN
  door_icon : TYPE = {open_door, ajar_door, closed_door, locked_door}
  IMPORTING radio_box[door_icon]
  availability_panel : TYPE = radio_box[door_icon].selected_radiobox_panel
  exception_item : TYPE = {always, when_open, when_ajar,
                           when_closed, when_locked, never}
  IMPORTING radio_box[exception_item]
  exception_panel : TYPE = radio_box[exception_item].selected_radiobox_panel
  attributes : TYPE = [# availability : availability_panel, exceptions : exception_panel#]
  ...
  END ρrestrictedpanel
```

where "radio_box" is a theory for radio box widgets.

Buttons at the interface are laid out in a specific order. To model this, theory "radio_box" uses a function that associates a natural number to each button. The only restriction is that no two buttons can be associated with the same number. This is expressed in the type definition so that no invalid orderings can be specified. In order to model the order in which the buttons of the exception panel are present in the interface, the following operator is introduced:

```
exception_panel_order : [exception_item → nat] =
  λ (ei : exception_item) : COND
                            always?(ei) → 1,
                            when_open?(ei) → 2,
                            when_ajar?(ei) → 3,
                            when_closed?(ei) → 4,
                            when_locked?(ei) → 5,
                            never?(ei) → 6
                            ENDCOND
```

In order for the presentation to convey the notion that exceptions accumulate (i.e. when an exception level is set, all door states up to that one are allowed) the exception setting level must vary progressively along the column of buttons. If that happens, the user will be able to know if a request for a service will succeed based on the relative position of the current accessibility level and current exception setting level. The request is allowed if the exception setting is equal or *bigger* than the current accessibility (for example, the situation illustrated in Figure 6.1 allows for glance to happen). This is conveyed by the following operator, which compares the exception level with the accessibility setting:

```
ρwillsucceed : [attributes → bool] =
  λ (a : attributes) : (maptoexceplist(user_avail) = user_excep ∨
                        isabove(exceptions(a), maptoexceplist(user_avail), user_excep))
WHERE
user_avail : door_icon = identify_avail(availability(a)),
user_excep : exception_item = identify_excep(exceptions(a))
```

A number of assumptions is being made regarding what the user will be able to do:

- "identify_access" and "identify_excep" model the cognitive tasks of identifying which button is selected in each of the panels;
- "maptoexceplist" models the cognitive tasks of mapping a door to the corresponding accessibility level;
- "isabove" models the cognitive tasks of identifying if a button is above another button.

Whether or not these are reasonable demands on the user falls into the scope of cognitive psychology. In this way, these operators can act as starting points for discussion

between software engineers and cognitive psychologists, regarding the design of the systems and the demands it imposes on the users. For instance, how easy will it be for a user to map a door into the corresponding exception level? does the design need improvement on this aspect? how can it be improved?

The final theory, combining the previous two, can now be written. Again the complete PVS theory is presented in Appendix F, an extract is shown below:

```
ecom : THEORY
  BEGIN
  IMPORTING restrictedpanel, ρrestrictedpanel
  ρavailability : [door → availability_panel] = ...
  ρexceptions : [exception → exception_panel] = ...
  ρ : [restrictedpanel.attributes → ρrestrictedpanel.attributes] =
    λ (rp : restrictedpanel.attributes) :
      (#availability := ρavailability(availability(rp)),
         exceptions := ρexceptions(accessexcep(rp))#)
  ...
  END ecom
```

where "ρavailability" and "ρexceptions" perform the translation of each of the door states (accessibility and exceptions setting) to the corresponding array of buttons.

Trying to prove that "ρwillsucceed" is an adequate operator to check for permissions, amounts to demonstrating that:

Theorem 6.3.1 (equivalence)

$\forall$ (rp : restrictedpanel.Attributes) : willsucceed(rp) = ρwillsucceed(ρ(rp))

When an attempt to prove the theorem is made, the proof arrives at a situation where the theorem prover asks for the following to be proven:

Sequent 6.3.1

```
|  {1}   never(accessexcep(rp'))
|-------------------------------
|
```

It seems that in order for the theorem to be true, it should not be possible to set the exception level to "never".

PVS powerful proof commands can be very useful when proofs can be successfully completed, as they allow for a high degree of automation. When proofs do not succeed, however, the very automated proof rules will in many cases oversimplify the sequents,

making it difficult to detect what exactly is causing the proof to fail. In such cases, it is necessary to perform the proof with smaller steps, in order to identify where the problem lies. Doing that, the proof arrives at a point where the following sequent is shown:

Sequent 6.3.2

$$\{1\} \quad \text{isabove}((\#\text{state} := \lambda(\text{ei} : \text{exception\_item}) : \text{ei} = \text{never},\ \text{position} := \text{exception\_panel\_order}\#),\ \text{when\_open}, \text{never})$$

$$\vdash$$

That is, the prover points out that, in order for the theorem to be true, the button for "when door open" should not be above the button for "never". This is not the way the system was designed (see Figure 6.1 on page 193). The problem is that the buttons for "never" and "always" are placed the wrong way around. The order of the buttons in the interface should be:

- Never
- (only if) Door Open
- (up to) Door Ajar
- (up to) Door Closed
- (up to) Door Locked
- Always

Note that what the sequent actually shows is that there is a mismatch between the cognitive operators that were initially considered to represent the user task, and the presentation that is being proposed to support such a task. The conclusion must be that either the task definition or the representation is wrong. In this case, the cognitive analysis corroborates the notion that the cognitive operators are correct regarding how a user would expect the representation to reflect the cumulative nature of exception levels. So, the conclusion must be that the buttons are in the wrong order. If the positions of the two buttons are swapped, the proof becomes trivial.

This problem was initially reported in (Bellotti et al. 1996). There, PUM analysis was used to model the system from an human-factors perspective. Here it is shown how in the context of a more software engineering oriented approach, the selection of appropriate abstractions allows for the same conclusion to be reached.

6.3.3 Analysing redundancy

The third set of criteria set forth initially will now be looked at:

> The effect of setting the exception level to "always"/"never" cannot be achieved by setting it to some other level.

This type of analysis can be best performed by comparative analysis of system behaviour. In this case, it is necessary to compare the behaviour of the model when the "always"/"never" exceptions levels are present, with the behaviour of the model when those levels are not included. In SMV the analysis could be done having different models of ECOM in parallel so that their behaviours could be compared. Putting this into practice, however, is not a trivial task. Using PVS the proof amounts to comparing the result of "willsucceed" when applied to different attributes, which is much simpler.

The criterion above, for the "never" case, can be expressed as the following theorem:

Theorem 6.3.2 (never2)

$$\forall\ (a1 :\ \text{attributes}) : \text{accessexcep}(a1) = \text{never} \Rightarrow \neg\ \exists\ (a2 :\ \text{attributes}) : (\text{accessexcep}(a2) \neq \text{accessexcep}(a1) \wedge \text{availability}(a1) = \text{availability}(a2) \wedge \text{willsucceed}(a1) = \text{willsucceed}(a2))$$

Note that the theorem demands that both availabilities be equal, so that the situation under comparison are the same except for the value of the exception level. When attempting to prove this theorem, three sub cases are generated. The sequent for the first sub case can be simplified to:

Sequent 6.3.3

$$\begin{array}{|ll} [-1] & \text{never?}(\text{accessexcep}(a1')) \\ [-2] & \text{availability}(a1') = \text{availability}(a2') \\ [-3] & \text{whendooropen?}(\text{accessexcep}(a2')) \\ \hline [1] & \text{open?}(\text{availability}(a2')) \end{array}$$

i.e., it has to be shown that, when the exception levels of attributes "a1" and "a2" are "never" and "when door open", respectively, and knowing that the availabilities are the same, then that availability must necessarily be "open". Since there is no constraint on the availability of "a2", this sequent cannot be proved. The interesting question, then, is why this sequent appears. There seems to be a problem with setting the availability to "open". Inspection of the specification shows that when the availabilities are in fact "open", "willsucceed" will be the same for both attributes if the exception level of "a2" is "when door open". Hence depending on the availability, there are situations where the effect of "never" can be achieved by other means. This reveals a weakness in the criteria: the effect of setting an exception level, depends on the availability, but that is not taken into account. By omission the criteria are demanding that the effect of setting the exception level to "always"/"never" cannot be achieved by setting it to any other level, *for all availabilities*. However, it is enough to have one availability setting which makes "never" unique in order to make that level non redundant. Hence, the criteria must be reformulated to:

> There is at least one availability setting for which the effect of setting the exception level to "always"/"never" cannot be achieved by setting it to some other level.

This new criteria can be expressed, again for the "never" case, as the theorem:

Theorem 6.3.3 (never3)

$$\begin{array}{l}\exists\ (\mathrm{d}:\ \mathrm{door}):\\ \quad \forall\ (\mathrm{a1}:\ \mathrm{attributes}):\ (\mathrm{availability}(\mathrm{a1}) = \mathrm{d} \wedge \mathrm{accessexcep}(\mathrm{a1}) = \mathrm{never}) \Rightarrow \\ \qquad\qquad \neg \exists\ (\mathrm{a2}:\ \mathrm{attributes}):\ (\mathrm{accessexcep}(\mathrm{a2}) \neq \mathrm{accessexcep}(\mathrm{a1}) \wedge \\ \qquad\qquad\qquad\qquad \mathrm{availability}(\mathrm{a2}) = \mathrm{d} \wedge \\ \qquad\qquad\qquad\qquad\ \ \mathrm{willsucceed}(\mathrm{a1}) = \mathrm{willsucceed}(\mathrm{a2}))\end{array}$$

which is easily proved. Since it has been proved that it is possible to find at least one situation where the effect of the never exceptions level cannot be achieved by any other level, it can be concluded that, from a software engineering point of view, "never" is not redundant.

Similarly for the "always" case the following theorem can be written:

Theorem 6.3.4 (always3)

$$\exists\ (\mathrm{d}:\ \mathrm{door}):$$
$$\forall\ (\mathrm{a1}:\ \mathrm{attributes}):\ (\mathrm{availability}(\mathrm{a1}) = \mathrm{d} \wedge \mathrm{accessexcep}(\mathrm{a1}) = \mathrm{always}) \Rightarrow$$
$$\neg\exists\ (\mathrm{a2}:\ \mathrm{attributes}):\ (\mathrm{accessexcep}(\mathrm{a2}) \neq \mathrm{accessexcep}(\mathrm{a1}) \wedge$$
$$\mathrm{availability}(\mathrm{a2}) = \mathrm{d} \wedge$$
$$\mathrm{willsucceed}(\mathrm{a1}) = \mathrm{willsucceed}(\mathrm{a2}))$$

In this case the theorem cannot be proved. Consider the following sequent which is obtained by using an availability of "locked" for "a2":

Sequent 6.3.4

[−1]	$\mathrm{availability}(\mathrm{a1}') = \mathrm{locked}$
[−2]	$\mathrm{accessexcep}(\mathrm{a1}') = \mathrm{always}$
[−3]	$\mathrm{availability}(\mathrm{a2}') = \mathrm{locked}$
[1]	$\mathrm{accessexcep}(\mathrm{a2}') = \mathrm{whendoorclosed}$
[2]	$\mathrm{accessexcep}(\mathrm{a2}') = \mathrm{whendoorajar}$
[3]	$\mathrm{accessexcep}(\mathrm{a2}') = \mathrm{whendooropen}$
[4]	$\mathrm{accessexcep}(\mathrm{a2}') = \mathrm{always}$
[5]	$\mathrm{accessexcep}(\mathrm{a2}') = \mathrm{never}$

What the sequent says is that, in order for the theorem to be true, when availability is "locked", and the exception level of "a1" is "always", then the exception level of "a2" cannot be "when door locked". What this points out is that there is no difference between setting the exception level to "always" or to "when door locked". The same type of situation can be reproduced for all door states.

Since "when door locked" is the most liberal level, it can even be proved that "always" and "when door locked" have the same meaning for every availability setting. Hence, from a software engineering point of view, one of the two levels can be eliminated from the system. Note however that this decision should not be taken too lightly. On the one (software engineering) hand the question should be asked as to whether all relevant aspects of the system have been considered in the model: is there any factor that may make "always" different from "when door locked"? On the other (human-factors) hand, input from a cognitive psychology oriented analysis should be sought: it may be the case that the two buttons have different cognitive meanings at the interface. This in turn would raise an issue regarding the design of the user's task: if the user has two different tasks that are functionally identical, then perhaps the task analysis should be looked into in order to see if it is possible to eliminate one of the tasks. Since in this case the only task which is considered relevant, regarding the exception levels, is setting exceptions, it could be decided to remove one of them.

6.4 Results

The role of formal (automated) verification is assessing designs against specific criteria for quality. If all criteria are met the design is satisfactory. If, on the contrary, the design fails to meet a criterion, then redesign is needed. This case study has shown how reasoning about usability criteria can be performed in a (formal) software engineering design context. One of the problems to overcome springs from the fact that, regarding usability, the quality criteria are not traditional quantifiable assertions about the system. Rather, they involve making assumptions about users and how they will use the system. Hence, when looking at what criteria to verify it is necessary to look at human-factors related issues.

More interesting than listing what properties was possible to prove, and what properties were not provable, it is to consider how it was decided which properties deserved looking into, and what the results of their verification might mean in terms of the design. This, then, will be subject of this section. Three main issues were analysed during the case study:

- whether pressing a connection button resulted in the corresponding connection being established;
- whether the presentation layout chosen for the exception buttons could be used to determine the effect of the buttons;
- whether it was possible to duplicate the effect of the "always"/"never" buttons with any other exceptions button.

Each of these issues will now be revisited.

6.4.1 Establishing connections

The analysis addressed whether pressing a button requesting a connection would result in that connection being established. The justification behind this line of reasoning is *predictability*, a design principle. In short, psychology says that if a system is predictable users will be able to learn how to operate it more quickly, and generally feel

more comfortable using the system. In this case it was decided to analyse the request for connections, since that is one of the main tasks a user will be using the system for: connecting to people. The conclusion of the analysis was that attempting a connection was not a predictable task. This was shown to be so for (at least) two reasons:

- possible changes in the availability setting of the callee between the time the caller sees that setting, and the time the button is actually pressed;
- possible exceptions (accessibility levels) set by the callee that falsify the availability conveyed by the callee's door state.

Since the properties have a human-factors background, analysis of the results must be done in a human-factors context. In this case, depending on the actual context of usage of the system, the first problem might actually be considered as not very relevant. The second, on the other hand, seems to point to a major flaw in design: two concepts are being integrated (general availability setting, and specific exceptions setting) which do not work well together, and lead to a breakdown in the coupling between the concepts of availability and accessibility. This will make this system harder to understand and learn. It is interesting to point out, at this stage, that a further question was raised during the analysis:

- changing exceptions/door states does not affect current connections.

This could either be seen as a problem or a feature. The interesting point about this third question is that, from a purely technological point of view, it does not seem to be directly related to the other two issues raised. The first two relate to setting connections, this last one to setting exceptions/availability. In fact, its connection with the initial problem was established at the psychological level of the analysis.

The findings of the analysis would now have to be fed back into the design process, so that they could inform the design. Since the breakdown between accessibility and availability is a major design issue, this could result in an overall reassessment of the design, and a completely new design being proposed. A less drastic solution would be to convey accessibility through disabling of buttons associated with unavailable connection services. Once again, whether this solution is a good one is a matter for

further analysis. In particular, this last solution would not solve the problem of users not knowing what the exact effect of selecting a specific availability level might be, unless they are able to remember all the exceptions they have set.

Once a new design was produced, it could again be verified. Note, however, that this time much of the work regarding what properties to analyse, and how to express them, would be already done. Unless major changes in design had happened. the model could also be reused and adapted.

6.4.2 Assessing the presentation layout

The analysis addressed whether the user would be able to determine the effect of selecting an exception level, by looking at the relative position of the exception buttons in the screen. The justification behind this line of reasoning is that the system should support user tasks and reduce the cognitive load on the user. If the user could tell the effect of an action by simply looking at the relative position of a button on the screen, this would reduce the cognitive load on the user and generally make the system less prone to errors (some would say *make the user less likely to commit an error* — on the discussion of where the blame should go when an error does happen, see Woods et al. 1994).

The result of the analysis was that the "never" and "always" buttons were positioned the wrong way around. This could cause users to make mistakes since the position of the buttons is counterintuitive. In terms of redesign, the problem could be solved by swapping the two buttons.

6.4.3 Uniqueness of buttons

The analysis addressed whether the user could reproduce the effect of the "never"/"always" buttons using any other exception setting button. The justification behind this line of reasoning is that, as a general principle, redundancy should be avoided at the user interface. Redundancy will make interfaces harder to learn and understand. Additionally, it will increase the memory load on the user, since there will be more to learn/remember. This analysis was applied to this specific feature of the system, due

to an instinctive suspicion that redundancy could be present. This illustrates that the process of deciding what to analyse can be influenced, not only by design principles and guidelines, but also by input from designers and human-factors experts.

The result of the analysis was twofold. Regarding "never", it was concluded that no other button could have the same effect (block all connections). Regarding "always", it was concluded that button "when door locked" achieves the same effect (allowing all connections). Hence, a possible redesign would be to remove the "always" button.

Obviously, all the redesign suggestions presented herein are just that: suggestions. They would have to be assessed in the overall context of design.

6.5 Discussion

Two main perspectives can be used to discuss the findings of the case study. From a technological perspective, the relative merits and shortcomings of the two tools used (SMV and PVS) can be discussed. From a methodological perspective, the results of the actual analysis and how they compare with that in (Bellotti et al. 1996) are of interest.

6.5.1 On the tools

Traditionally, approaches to the use of automated reasoning in interactive systems have been based on choosing a particular tool and analysing how it can be used to perform verification (see for instance, Paternò 1995, Bumbulis 1996). No one tool has the capability to analyse all aspects of interest, so in choosing a specific tool, the analyst is restricting the field of analysis. The approach in this thesis has been to focus first on the properties and then choose the most appropriate model/tool combination. Two main tools were analysed: SMV, a model checker, and PVS, a theorem prover. Both tools proved useful/needed during the case study.

Choosing a tool

SMV, being a model checker, is naturally more adequate when reasoning about the behaviour of the system is required. The construction of an additional layer on top

of SMV's state oriented model, has enabled analysis to be done in terms of actions as well as state. This contrasts with the use of model checking in (Paternò 1995), for example, which enables reasoning about events only. PVS, on the other hand, is best suited to reason about properties of the system state, and the relationships between the different components of the state. Note however that, with an appropriate model, it is still possible to reason about some characteristics of the system behaviour in the context of theorem proving. In this case study, for instance, the conditions that would allow a connection to be established were encoded as the state attribute "allowed". This enabled the use of the theorem prover to analyse the impact of setting exception regarding future connection attempts. There is clearly a degree of flexibility when deciding what technique to use. Not so much a thin line, but rather a grey area where what technique should be used depends on what type of model is used. On the other hand, when developing the model the tool that has been chosen needs to be taken into account. The two go hand in hand.

Applying the tool

Regarding the feasibility of applying the tools, SMV has clearly a problem with state space explosion. This is to be expected in a model checker. Nevertheless, being a symbolic model checker, SMV can deal with quite big state spaces. In this case study the option, regarding SMV use, was to try and follow the original MAL specification as closely as possible. This included modelling the interactors sets and finite functions as SMV arrays. This proved not to be an ideal solution since it aggravates the state explosion problem, mainly because the arrays had a combinatorial effect on the number of state variables that were needed at the SMV level. Clearly as the systems under analysis grow in complexity a higher degree of simplification of the models is needed. This raises a problem. How to simplify the models without removing from them the very same problems and unexpected interactions between different components which the verification is supposed to detect? This seems to be a problem that is inherent in model checking. A number of possibilities to overcome this problem were discussed. Regarding the case study, it was decided to consider only two users in the final model. A connection has a caller and a callee, since no third user can influence the connection,

two users are enough to reason about connections. Obviously the noninterference of a third user is a property of the system which is being assumed. Ideally, this would be the trigger for another iteration of the verification process, this time to verify that no third party can interfere with a connection. Additionally, it was possible to eliminate state attribute "allowed" by rearranging same of the axioms.

Still regarding model checking, there are a number of parameters that can be used to tune the symbolic algorithm used by SMV. These can have quite a dramatic impact on the times taken to obtain results. As an example, reordering the sequence in which variables were analysed by SMV, reduced the time taken to perform a proof from fifty-five minutes down to five. Choosing the best ordering, however, is down to heuristics. For example, keep tightly coupled variables together in the ordering, and put variables with large domains first.

Unlike SMV, PVS could work at the original specification level, and was more flexible in relation to the style of models which could be written. This does not necessarily imply that such models were easy to analyse. In PVS the proof process is not a fully automated step as in SMV. The user has to decide on the best strategy to perform the proof and guide the theorem prover where appropriate. The available proof commands and strategies, however, have proved very powerful. Especially when, as it was the case in the current analysis, all the types are enumerated types, and many of the proofs amount to the analysis of all the possible situations that the different combinations of values might produce. In fact, most of the theorems, when true, could be proved with a single proof command: `grind`, with appropriately set parameters. When a proof fails, however, these powerful commands can become counterproductive: they tend to simplify the sequent to an extent that makes the reason for failure no longer apparent. In those cases a less automated, more laborious, attempt at the proof has to be made in order to identify the causes for failure. In part due to this need to be more involved in the proof process, PVS provides better feedback regarding the specification than that provided by SMV. Additionally, the type correction conditions (TCCs) which are automatically generated, will point out inconsistencies within the specification.

Unfortunately, PVS does not have an internal model of objects and behaviour. So,

	SMV	PVS
Behavioural reasoning	✓	×
State based reasoning	×	✓
Model development	×	✓
Automated analysis	✓	∼
Tool learning	✓	×
Expressive power	×	✓

Table 6.1: SMV vs. PVS

reasoning about the behaviour of the system could only be done indirectly, by encoding information about the behaviour of the system as state attributes (cf. the user of attribute "allowed" to model whether a connection will be permitted). Additionally, learning how to use PVS, not only the specification language (which is more complex than SMV's) but specially the proof commands (which commands to use where, and how), has a steep learning curve.

When comparing the use of PVS to a manual proof process, is it clear that PVS helps in performing the proofs. The proof commands, once learnt, allow much of the proof to be automated, and it is easy to explore the application of different strategies and commands. A manual proof relies on the skill of the person performing the proof, PVS enables this dependency to be lessened.

SMV vs. PVS

As a conclusion it can be said that the idea that model checking is an automated reasoning tool, is something of a myth. In this respect, the main difference between model checking and theorem proving is not so much whether the process is fully automated, but rather, where work has to be done. In SMV work goes mainly into developing the model and tuning the checking process. In PVS work goes mainly into guiding the theorem prover through the proofs. It must be said that in the current case study more work was performed in the first case (SMV) than in the second (PVS). This is due to the particular type of properties and model that were being analysed/used. In the analysis in Chapter 5, for instance, PVS needed more guidance. Table 6.1 summarises the conclusions regarding the tools.

Finally it is worth noticing that during the thesis an interactor for buttons was repeatedly used in the case study, a PVS theory for radio boxes was also developed. This suggests that toolkits of widget models could be developed both for SMV and PVS, which could then be used when developing the models for verification. These models could have already proved properties that could be used during the verification process. Additionally, these widget models could be used as a basis for the development of the actual implementation of the user interface in the style of (Hussey & Carrington 1998) or (Bumbulis et al. 1996).

6.5.2 On the analysis

The results of the analysis, in terms of the specific design and properties being analysed, have already been addressed in Section 6.4. This section, looks at the analysis from a more general point of view. One possible criticism of the current case study, is that most of the results have already been achieved by other means in (Bellotti et al. 1996). This however, would be missing a number of important issues. This section discusses those issues, and notes the major points that the case study has illustrated.

Feasibility

In (Bellotti et al. 1996) the design is analysed by four different teams of analysts. So, this is a well thumbed case study. The main goal then was to see if the results could be reproduced. When issues like the analysis of the exceptions buttons layout are analysed, the point is not that a theorem prover would be needed to reach the conclusion. The relevant point is that a theorem prover *could* be used to reach the conclusion. At the same time, automating part of the analysis has had the effect of providing relevant triggers for the expertise of other disciplines to be incorporated into the software engineering of interactive systems. Obviously it would be interesting to perform a comparative study between this and other approaches. This would imply finding a suitable case study (complex enough to be representative, but not so complex that it would make the effort of performing a set of different analyses prohibitive), and setting up teams with the appropriate expertises and availability to engage in the case

study. Controlled experiments of this kind are hard to come by and results depend very much on individual differences of participants.

Additionally, having a number of models each built with a different aspect of the system in mind, can be viewed as a problem. How is it possible to guarantee that all the models are consistent? The case study shows how models can be built in relation to each other, each taking a more detailed view of a particular aspect of the system. Each model can be viewed as a fish-eyed image of the system, where some aspects are presented in great detail, while less important issues are only roughly sketched. All models must overlap consistently in what is common between them. So, the model used to analyse consistency in Section 6.3.2 uses only some of the state attributes of the model used to analyse predictability in Section 6.3.1, but the definitions of those attributes are consistent. At the same time the model adds more detail regarding the actual user interface, and again the visibility modalities used in Section 6.3.1 can be seen to be consistent with this new, more detailed, view of the user interface. Additionally, the analysis in Section 6.3.2 also shows how the development of different models can be used to detect inconsistencies between different views of the system.

Complementarity

Although the thesis has focused specifically on the use of automated reasoning, it is not argued that this type of approach should replace other approaches to analysis and verification of interactive systems. On the contrary, automated reasoning should be seen as one more tool available to perform analysis.

A software engineering approach

In (Bellotti et al. 1996), most of the analysis was performed by psychologists; the case study shows that similar results can be obtained inside a (formal) software engineering context. More specifically, the case study shows how, by using automated reasoning tools to analyse properties derived from design principles and/or from input from the designers and from human-factors experts, it is possible to reach conclusions regarding the usability of the system. By identifying where human-factors plays a role, the

process potentiates a better integration of the user related concerns of human-factors approaches with the rigour and analytic power of formal proof. This integration is further developed at the stage of results analysis. It would be dangerous to think that once a property is proved (or not) the verification process has ended. Particularly when a properties turns out to be false, the consequences of it, and the reasons for it, must be analysed and that again can only be done in cooperation with designers and psychologists. Note, for instance, how the discussion about both "never" and "when door locked" buttons having the same meaning (at the system level), did not necessarily meant that they should be replaced by just one button.

Formal techniques and tools allow for precise conclusions to be reached regarding a design from very early in development. Human-factors approaches address user related concerns that fall outside the scope of traditional formal methods. Bridging the gap between the two areas should contribute to a better design process.

Scalability

A major concern with approaches to design is their scalability — how well they will cope with design of increasing size and complexity. One potential problem with the analysis in (Bellotti et al. 1996) is that it was performed manually and somewhat informally (although in the context of formal modelling). Although the case study is realistic in the sense that it comes from a real design context, it is not *big*. Automated reasoning has better potential regarding scalability than human analysis, which becomes more error prone as the complexity and size of the systems grows. Being partly automated, the proof process is less dependent on the skill of the analyst, hence more robust. Additionally, the possibility of performing a compositional style of reasoning, and of reusing proofs and properties, further helps in controlling complexity.

Still regarding scalability, it is worth noticing that the results in (Bellotti et al. 1996) came about mainly from insights gained during the modelling stage. This insight is obviously one of the main advantages of using formal modelling, however it cannot be guaranteed that problems will be found at this stage. The modelling process facilitates the finding of problems, but they can be, and often are, overlooked. During the case study, this aspect of modelling was intentionally omitted in order to determine

whether the automated reasoning process could detect the problems even if they had gone unnoticed previously. This aspect also becomes more and more important as the complexity of the systems grows. In this regard it is worth noting how an issue arose about how current messages should be dealt with when changing accessibilities, despite this not being the main concern of the analysis.

6.6 Conclusions

This chapter has shown how automated reasoning techniques can be used to help the formal analysis of interactive systems design. Two main techniques have been used: model checking, and theorem proving. Traditionally, theorem proving is considered more difficult to use, however the case study shows that this is not necessarily always the case. It is true that theorem proving demands mathematical knowledge about logic and proofs, while the proof process in a model checker is completely automated. However, making a model model checkable can be a task which also requires quite an amount of expertise.

The analysis in Section 6.3.1 was based on a generic design principle: predictability. It analysed whether the commands to establish connection were predictable. To perform this analysis an interactor based model using Modal Action Logic was developed, and SMV, a model checker, was used. It was concluded that *the commands for establishing connections were not predictable* since there was a mismatch between the actual accessibility settings of a user and the information made available to the community. During the analysis of this result it was also uncovered that *current connections are not affected by changes in accessibility and/or availability.* Once a problem, and its causes, are identified, it usually falls out of the scope of the formal approach to decide the best solution. That solution will have to be reached in collaboration with other user centred approaches. What formal methods have to offer is the possibility of, given a set of quality measures, comparing the different proposals.

Section 6.3.2 investigated whether the proposed presentation is coherent with the underlying semantics of the door states. This analysis was done in PVS, a theorem

prover. The conclusion was that *the ordering of the exceptions buttons was inadequate.* More specifically, the "never" and "always" buttons were in reversed positions.

Finally, the properties analysed in Section 6.3.3 are the result of applying another design principle to the proposed system. As a generic principle, redundancy at the interface should be avoided. A formal analysis, performed with the PVS theorem prover, detected a situation where in fact there was redundancy: the "when door locked" and "always" buttons had the same meaning; and another where that didn't happen: both "never" and "when door open" buttons were needed. It is also worth pointing out how in this latter case the theorem prover was used, not to detect a problem but to certify that a given design option was indeed correct.

Several approaches to the analysis of the ECOM system were used in (Bellotti et al. 1996). Comparing the present analysis with those, it can be seen that the present approach can be used to complement the formal system analysis performed there. Here automated tools are used exactly to perform that kind of analysis. Regarding the cognitive user modelling approaches (PUM and CTA), it can be seen that it was possible to reach similar results. However, the present results were obtained while still in the context of a software engineering approach. PUM and CTA demand human-factors expertise. It is clear how the approach can be complementary to those in the sense that they can be used to define a set of quality measures which can then be rigorously analysed using formal methods. Furthermore, they can help in interpreting the results of such analytic process. Finally the issue regarding what should happen to current connections when accessibilities or availabilities change, although mentioned as something that could be done in the formal analysis discussion of (Bellotti et al. 1996), was not actually looked into. This might be because the analysts were not concerned with those aspects of the system. It is interesting to notice that the issue surfaced in the automated reasoning, even when it was not initially being considered during the modeling and analysis stages. In other words, it can be said that it was not the analyst, but the tool, that noticed that this aspect was an issue.

This chapter has presented a case study were the different approaches to verification described previously were brought together. The next chapter reflects on what was accomplished, and identifies some lines of possible future work.

Chapter 7

Conclusions and Future Work

All the work has now been presented. Automated deduction and usability reasoning are two rich areas of research. This chapter discusses the motivation behind the particular line of work taken, reviews the contributions made, and compares them with other work in the area. Finally, some pointers for further work are put forward.

7.1 Introduction

This thesis has addressed the applicability of automated deduction techniques in performing usability reasoning. Usability relates to how easily users will be able to operate a system. This includes aspects such as learning and remembering the system. By addressing user related issues, usability reasoning is closely linked with applied sciences such as psychology. In these sciences, results are obtained primarily by looking at the world. Hence, much of usability reasoning is done by placing users in front of the system and analysing how they perform (or, from another perspective, how the system performs). Automated reasoning, on the other hand, is based on mathematics, and its aim is to enable precise reasoning about the world, from a model of that world. Hence, the analysis is done without direct observation. This duality has lead, from the early stages of work, to a tension between two competing problems. On the one hand, research on what it means to formally verify usability (for example, how to express it), and how to go about doing it. On the other hand, the need to research and develop technology suitable for the verification of interactive systems models.

Regarding the first issue, many properties seen in the literature seem rather artificial, and their connection to actual usability rather vague. If usability issues are not considered, however, all that is left is little more than the verification of traditional systems. Attempting to formalise and define what is meant by usability, on the other hand, would render a thesis which would address much of human-factors concern but little of computer science (at least in terms of automated deduction).

Regarding the latter issue, it is the case that technology is still not available to conveniently verify, for example, object oriented models of systems (as is the case with interactor based model of interactive systems). The integration of the available technologies (model checking and theorem proving) in order to gain the benefits of both approaches to verification is also an interesting area of research (see, for example, Rajan et al. 1995, Owre et al. 1997). Taking this route, however, would result in a thesis on automated deduction with little connection with the proposed theme.

Between these two sets of issues, a balance had to be struck. The approach taken has been to study how available automated deduction tools can be used in the verification of usability related properties.

7.2 Contributions

As just explained, instead of concentrating primarily on the development of tool support for verification, or on the development of a deeper understanding of what usability means, both of which are worthwhile undertakings, a more pragmatic approach was followed, concentrating on using currently available tools to reason about properties that are recognised as having bearing on usability. Since no review of the area was available, the first step was to do such a review. From this resulted the first contribution of this thesis:

> A review of the main approaches to the verification of interactive systems specifications (Chapter 2).

From the review came the realisation that no single approach will have the expressive and analytic power needed to tackle all the relevant issues raised by interactive

systems development. What is more, the approaches were mostly based around a specific tool, and, as already mentioned, the link between usability and the properties being verified not always clear. A shift was needed from a tool centric approach to a feature centric approach. In this ways the analyses is no longer tied to a particular method or tool. This becomes the main contribution of the thesis:

> A proposal for a closer integration of verification into the interactive systems development life-cycle, based on a move to a feature centric approach to verification (Chapter 3, Section 3.4).

Where the proposed approach differs from traditional iterative approaches to software development is that the iterations are not based around a model of the system that is successively refined but on system features for which appropriate models are derived. Hence each iteration will potentially look at different aspects of the system, instead of each interaction being a new look at the system as a whole. For example, in Chapter 6 each of the analysis performed corresponds to a complete iteration of the proposed process, and each one of them is looking at distinct system features and properties.

Even though the thesis has not to embarked on a deep study of the psychological factors of usability, there was nevertheless the need to identify those features which have an impact on usability. A characterisation of what a usability related property might involve was needed. Such a characterisation has been developed in order to compare the different approaches described in the review. This is, then, a further contribution of the thesis:

> A framework identifying what is involved in usability related properties (Chapter 3, Section 3.2).

Defining how verification should integrate with development provides a general framework. There is now the need to study how available technology can be used in the context of such a framework. Chapters 4 and 5 discuss how this can be done, using model checking and theorem proving, respectively. Chapter 4, in particular, presents the next major contribution of the thesis:

A tool to enable the verification of interactor models in SMV (Chapter 4, Section 4.2).

Between the two, these chapters give examples of reasoning about perception, mode complexity, the relationship between system and interface, and the effects of actions on the system and how they are perceived by the user. They also present a number of smaller contributions. They show how the framework in Chapter 3 can be used to assess the meaningfulness of properties in terms of user issues (Chapter 5). They show how, by embodying those user related issues, properties become models of specific sets of user characteristics (sets of assumptions made about the user). Additionally they illustrate how, in the context of usability analysis, property formulations are not static entities from which a definite answer is obtained. Rather they evolve during verification assuming the role of scenarios of interaction that can be used to illustrate why a property is not valid, or alternatively, under which conditions a property is valid, at the same time raising questions for further analysis and interdisciplinary discussion. Verification is no longer simply a proof process, but an exploration of the model, from a given perspective (the property) which is based both in system behaviour, and expectations about the user. This has lead to another major contribution of the thesis:

The possibility of performing reasoning about human-factors related issues in a primarily software engineering oriented context.

Finally, Chapter 6 presents a larger case study where the approaches discussed previously are put together in the analysis of a audio-visual communications system. A problem commonly pointed out in relation to the use of multiple models of the same system, is the possibility of incoherence between those models. This chapter shows how this can be overcome by the use of an assumption-commitment style of reasoning. The chapter also shows how this has the additional advantage of encouraging the use of compositional reasoning in verification. Additionally, it becomes possible to use different tools to discard each assumption and property. For example, theorem proving can be used to discard an assumption, and model checking to check a property based on that assumption. Hence, this promotes the integration of theorem proving and model checking, not at the level of the tools, but at the level of the process.

7.3 Analysis

This thesis has proposed a new way of thinking about verification of interactive systems. The focus of attention is shifted from the tools to the properties of the system being developed. Since the main approaches discussed in Chapter 2 are based around specific tools, a direct and global comparison between each of them and the work in this thesis is not appropriate. This thesis proposes, above all, a process to integrate verification into interactive systems development. In this process each of the reviewed approach can take a part, if that is deemed useful.

On the other hand, the thesis also discusses and gives examples of using both a model checker (SMV) and a higher-order logic theorem prover (PVS) to perform verification in the context of the proposed approach. At this level a comparison can be made between the results obtained here, and the results obtained in the literature. It should be clear, however, that it is not proposed that the two tools discussed herein will cover all of the relevant aspects of interactive systems design (although it was shown that they can be used to cover some of the relevant issues).

7.3.1 Main approaches (revisited)

Keeping the above in mind, some comments can be made regarding how the techniques described in Chapters 4 and 5 (specially the compiler developed in Chapter 4) fair against the main approaches described in Chapter 2:

- SMV — Abowd et al.'s (1995) approach analyses simple PPS models only. The compiler developed in Chapter 4 uses a similar type of approach, but extends it to enable the model checking of more complex interactor based models.

- Lite — Regarding technology, the model checking approach in Chapter 4 allows for state attributes as well as actions to be used in the model and in verification, Paternò's (1995) approach works only with actions. Although actions can be used to encode relevant state changes, this is rather artificial. There is also the problem of knowing, in advance, which state changes are relevant and which are not. Using both actions and attributes helps solve this problem. Regarding

methodology, Paternò's (1995) approach is clearly task based. The proposal in Chapter 3 does not define a specific style of development. This is so that it remains flexible enough to be integrated in whatever development approach is being used. Hence, a task based approach could be (one of the approaches) used to generate properties for verification.

- HOL — Bumbulis et al.'s (1996) approach deals with properties of the interface at the device level. In this case a theorem prover is used. This type of approach is rather restrictive in the properties that can be verified. In this thesis the analysis starts much sooner and can be performed as the development progresses. Chapter 5 shows how theorem proving can be used to perform a more powerful analysis of the actual interface being built. This is accomplished by analysing the relationship between user interface devices, underlying system state, and the perception the users might have of the system.

- Lesar — Concerning the technology used, the main distinguishing characteristic of d'Ausbourg et al.'s (1996) approach is the ability to reason about data flow models. However, by using only boolean flows, the approach is in fact limited to reasoning about changes in those flows, which become roughly equivalent to actions. Although none of the approaches presented here enables reasoning about continuous interaction, Chapter 4 does show how continuous information can be transformed into discrete information. Regarding the method, like (Bumbulis et al. 1996), d'Ausbourg et al. (1996) perform the verification from a description of the actual interface, the same comments as above apply.

One of the main differences between these approaches and the work in this thesis is the role properties play in verification. On all of the approaches, properties are mainly seen as expressing some fact about the system which, if verified, should guarantee the system to be usable. Although it is said that the proof process will either validate the property or give a counter example, properties are globally seen as static entities. In this thesis, properties are used not only to encode information about the system, but also about the user. In this way, they act as user models at the *micro level* of specific system features. Additionally, they are seen to change as the verification step

progresses, acting then as scenarios of usage. Hence they become dynamic entities.

7.3.2 Recent work (revisited)

Doherty et al. (1999) apply HyTech, a tool for reachability analysis in hybrid automata, to the analysis of the flight deck instrumentation concerning the hydraulics subsystem of an aircraft. The use of hybrid automata to analyse specific (continuous) aspects of a system fits into the overall approach proposed in this thesis. While Doherty et al. (1999) propose that a model of the user be explicitly built, the user model in the case study is very simplistic. It corresponds simply to all the actions that can be performed by the user. Hence, when analysing task related issues, the strategies to accomplish a task are instead encoded in the verification steps to be performed. This is somewhat similar to the approach followed on this thesis of modelling user related issues into the properties to be checked.

Rushby (1999) also uses an explicit model of the user. In this case the model is built into the previously developed model of the system. This approach has already been discussed in Chapter 4. Basically, the model is built around specific sequences of events. This begs the question of how to select the appropriate sequences of events. In Chapter 4 similar results were obtained using a more generic model of the system under analysis and using user expectations about the system as predicates to be verified.

Finally, there is also some recent work being developed at NASA, Langley Research Center concerning the analysis of mode complexity. These approaches are similar to the mode complexity analysis in Chapter 4. A model of the system is built and properties expressing user related concerns are verified. The models relate to the inner working of the system's mode logic, while in Chapter 4 the model was built around the user interface of the mode logic subsystem. While this latter approach might be criticised from the point of view that not all mode logic information is presented at the interface, it allows better exploration of the interaction between the user and the system, and not simply of how the system reacts to commands. Reasoning about user-system interaction was, after all, the main goal of the thesis. Additionally, the relation between user interface and internal state can also be explored, as Chapter 5 has shown. The work as NASA can be seen as complementary to the work presented

in this thesis, as it explores the applicability of different verification technology to the analysis of a specific aspect of a type of system.

7.4 Future work

The present thesis has setup a framework for the verification of interactive systems. This, however, is only the first stage in the development of a practical approach to interactive systems verification. This section presents some proposals for future work. They are divided into two main groups:

- work on making verification more feasible;
- work on making verification more accessible.

7.4.1 Making verification more feasible

In order for a realistically usable approach to be developed, two lines of work are immediately obvious:

- on the human-factors side — the study of what properties make specific types of system more usable. The framework in Chapter 3 gives a general characterisation of what is involved in usability related properties. This is done at an abstract level, since the idea was not to focus on a specific type of system. Chapters 4 and 5 give example of possible analysis which can be performed. To facilitate the applicability of the approach, important properties should be identified and studied for different types of systems. This work should investigate what makes a particular type of system more usable than another, and how such usability factors might be expressed as verifiable properties. The envisaged result would be a toolkit of usability factors, organised by type of system.
- on the software-engineering side — the study of verification techniques for different types of systems/models/properties. Chapters 4 and 5 address the use of model checking and higher-order logic theorem proving in general. It is shown how they can be used to reason about relevant issues pertaining to usability.

However, the main motivations behind the work has been to analyse how the proposed verification process could be applied. Issues such as time or continuous interaction have not been specifically addressed, as they fall outside the usual capabilities of traditional tools. As reported, some work has been done in this area, but clearly more research is needed. Obviously this research should link with the result of the item above. Work on the verification technology itself needs also to continue. For example, regarding the verification of object-oriented models. Hence, this line of work can be subdivided into: using available tools to reason about different aspects of interaction; working on the development of new tools.

The two major lines of work above relate to what can be seen as the macro level of the research area. At the level of the particular work presented in this thesis, some lines of further development can also be identified. One such line is to further develop the interactor compiler. At the moment, modalities are not reflected in the generated SMV code. Although it has been shown how modalities can be used to drive the formulation and interpretation of properties, it would also be possible to model them as predicates on the SMV state. One probable drawback of this is that models would become bigger, hence more difficult to model check. The relative merits of encoding modalities in SMV should be investigated. Regarding the language used to write axioms, it can be noted that variables are only allowed as action parameters, and that they are implicitly universally quantified. At the same time, constants and attributes cannot be used as action parameters. This could be made more consistent if quantification was made explicit, and the use of actions, constants and attributes generalised to all expressions. Additionally, at the moment the $>_{\texttt{action}}$ operator means each state can only be reached by an action. The use of a boolean $>$ operator, parameterised by actions, should also be investigated. This would allow a state to be reached by any number of different actions, hence potentially decreasing the size of the state space. A potential drawback is that it would make it more difficult to keep track of what actions caused which transition. On the other side, it would allow for actions to occur concurrently inside an interactor. This late issue would ask for a redefinition of the semantics of "nochg". If "nochg(a)" is interpreted as mandatory, then it is impossible for another action to

change attribute "a". Hence, "nochg" would need a less stricter semantics.

Additional work can also be done in better exploring the compositional style of proof that the approach favours. This would involve the realisation of further case studies. Finding suitable case studies is a difficult task on its own. All of the case studies so far were taken from the literature. Although care has been taken to avoid interference from factors such as hindsight, ideally it should be possible to apply the approach to an ongoing project, so that its results could be judge more accurately.

It would be of little use to prove that a model represent an usable system, if it could not also be proved that the reification process preserves the relevant properties of the model. Chapter 5 has shown how the relationship between functional core and user interface can be analysed. This could be used as a basis for an approach to the formal reification of interactive systems (in a style close to that proposed in Doherty & Harrison 1997). This represents a further area of future work.

7.4.2 Making verification accessible

Developing an understanding of usability and verification technology to go with it, is not enough to guarantee an useful and usable approach to verification. There is also the need to support the designer/analyst in performing such verification. A possibility for this is to build layers on top of the tools which make the concepts involved more readable to someone not fluent in formal methods. The use of graphical notations might be a useful possibility. Graphics, however, can also make things more confusing so this needs research. The area opens several lines of work. One is research on interfaces for the verification tools. STeP, for example, uses diagrams to represent proof strategies. Another is the need to support the formulation of properties. Can graphical notations be of use here? Where properties are expressed as goals, maybe graphical representation of start and target interface states could be used. This is only a possibility, work needs to be carried out to determine how best to address this issue. The toolkits of predefined properties mentioned above could be of use here. Regarding reification, and as suggested in Chapter 6, a toolkit of widget models could be useful in building and analysing models of interface implementations.

Epilogue

"Beware of bugs in the above code; I have only proved it correct, not tried it."
attributed to Donald Knuth (not formally verified)

The thesis is now complete. Ultimately, usability cannot be proved. Only time will be a judge of the quality of the interaction between users and a system. Nevertheless quality (usability) can be improved. Formal methods and tools can be useful here. They can help in the detection of bad design.

A new approach to the use of formal verification techniques for usability reasoning has been proposed. The approach favours a closer integration of verification into development. The use of model checking and higher-order logic theorem proving in the context of the approach has been demonstrated. This thesis is a step towards a usable method for the verification of interactive systems during development, future lines of work have been proposed.

Bibliography

Abowd, G. D. (1991), Formal Aspects of Human-Computer Interaction, PhD thesis, University of Oxford. Also available as Technical Report YCS 161, Department of Computer Science, University of York.

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

Baader, F. & Nipkow, T. (1998), *Term Rewriting and All That*, Cambridge University Press.

Barnard, P. J. (1987), Cognitive resources and the learning of human-computer dialogs, *in* J. M. Carroll, ed., 'Interfacing Thought: Cognitive Aspects of Human-Computer Interaction', A Bradford Book, The MIT Press, chapter 6.

Barnard, P. & May, J. (1995), Interactions with advanced graphical interfaces and the deployment of latent human knowledge, *in* F. Paternò, ed., 'Interactive Systems: Design, Specification, and Verification', Focus on Computer Graphics, Springer, pp. 15–49.

Bellotti, V., Blandford, A., Duke, D., MacLean, A., May, J. & Nigay, L. (1996), 'Interpersonal access control in computer-mediated communications: A systematic analysis of the design space', *Human-Computer Interaction* **11**, 357–432.

Bellotti, V., Shum, S. B., MacLean, A. & Hammond, N. (1995), Multidisciplinary modelling in HCI design ...in theory and in practice, *in* 'Human Factors In Com-

puting Systems: CHI '95 Conference Proceedings', ACM Press, Addison-Wesley, pp. 146–153.

Berezin, S., Campos, S. & Clarke, E. C. (1998), Compositional reasoning in model checking, *in* de Roever et al. (1998), pp. 81–102.

Bjørner, N. et al. (1996), *STeP User's Manual (Educational Release)*, Computer Science Department, Stanford University. Version 1.2-α.

Blandford, A., Butterworth, R. & Good, J. (1997), Users as rational interacting agents: formnalising assumptions about cognition and interaction, *in* Harrison & Torres (1997), pp. 45–60.

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

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

Bryant, R. E. (1986), 'Graph-based algorithms for boolean function manipulation', *IEEE Transactions on Computers* **C35**(8), 477–.

Bumbulis, P. (1996), Combining Formal Techniques and Prototyping in User Interface Construction and Verification, PhD thesis, University of Waterloo.

Bumbulis, P., Alencar, P. S. C., Cowan, D. D. & Lucena, C. J. P. (1996), Validating properties of component-based graphical user interfaces, *in* Bodart & Vanderdonckt (1996), pp. 347–365.

Burch, J. R., Clarke, E. M. & McMillan, K. L. (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.

Butterworth, R. & Blandford, A. (1997), Programmable user models: The story so far, PUMA Working paper WP8, School of Computing Science, Middlesex University. *http://www.cs.mdx.ac.uk/puma/publ/WP8.html

Butterworth, R., Blandford, A. & Duke, D. (1998*a*), The role of formal proof in modelling interactive behaviour, *in* Markopoulos & Johnson (1998), pp. 87–101. also available as (Butterworth et al. 1998*b*).

Butterworth, R., Blandford, A. & Duke, D. (1998*b*), The role of formal proof in modelling interactive behaviour, PUMA Working paper WP12, School of Computing Science, Middlesex University.
*http://www.cs.mdx.ac.uk/puma/publ/WP12.html

Campos, J. C. & Harrison, M. D. (1997), Formally verifying interactive systems: A review, *in* Harrison & Torres (1997), pp. 109–124.

Campos, J. C. & Harrison, M. D. (1998), The role of verification in interactive systems design, *in* Markopoulos & Johnson (1998), pp. 155–170.

Campos, J. C. & Harrison, M. D. (1999*a*), From interactors to SMV: A case study in the automated analysis of interactive systems, Technical Report YCS-99-317, Department of Computer Science, University of York.
*ftp://ftp.cs.york.ac.uk/reports/YCS-99-317.ps.gz

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

Campos, J. C. & Harrison, M. D. (n.d.), Detecting interface mode complexity with interactor specifications. submitted to Automated Software Engineering.

Campos, J. C. & Martins, F. M. (1996), Context sensitive user interfaces, *in* Roast & Siddiqi (1996).
*htpp://www.springer.co.uk/eWiC/Workshops/FACHI/

Card, S. K., Moran, T. P. & Newell, A. (1983), *The Psychology of Human Computer Interaction*, Lawrence Erlbaum Associates.

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

Clarke, E. M., Wing, J. M. et al. (1996), 'Formal methods: state of the art and future directions', *ACM Computing Surveys* **28**(4), 626–643.

Crow, J., Owre, S., Rushby, J., Shankar, N. & Srivas, M. (1995), 'A tutorial introduction to PVS', Presented at WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques.
*http://www.csl.sri.com/sri-csl-fm.html

d'Ausbourg, B. (1998), Using model checking for the automatic validation of user interfaces systems, *in* Markopoulos & Johnson (1998), pp. 242–260.

d'Ausbourg, B., Durrieu, G. & Roché, P. (1996), Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour, *in* Bodart & Vanderdonckt (1996), pp. 105–122.

d'Ausbourg, B., Seguin, C., Durrieu, G. & Roché, P. (1998), Helping the automated validation process of user interface systems, *in* 'Proceedings of the International Conference on Software Engineering, ICSE '98', IEEE, pp. 219–227.

de Roever, W.-P. (1998), The need for compositional proof systems: A survey, *in* de Roever et al. (1998), pp. 1–22.

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

Degani, A. (1996), Modeling Human-Machine Systems: On Modes, Error, and Patterns of Interaction, PhD thesis, Georgia Institute of Technology.

Dill, D. L. (1996), The Murφ verification system, *in* R. Alur & T. A. Henzinger, eds, 'Computer Aided Verification, CAV '96', Vol. 1102 of *lncs*, Pringer-Verlag, pp. 390–393. murphy.

Dix, A. (1990), Non determinism as a paradigm for understanding the user interface, *in* Harrison & Thimbleby (1990), chapter 4, pp. 97–127.

Dix, A. & Abowd, G. (1996), 'Modelling status and event behaviour of interactive systems', *Software Engineering Journal* **11**(6), 324–346.

Dix, A., Finlay, J., Abowd, G. & Beale, R. (1993), *Human-Computer Interaction*, Prentice-Hall.

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

Doherty, G., Campos, J. C. & Harrison, M. D. (2000), 'Representational reasoning and verification', *Formal Aspects of Computing* (in press).

Doherty, G. & Harrison, M. D. (1997), A representational approach to the specification of presentations, *in* Harrison & Torres (1997), pp. 273–290.

Doherty, G. J. (1998), A Pragmatic Approach to the Formal Specification of Interactive Systems, PhD thesis, Department of Computer Science, University of York.

Doherty, G., Massink, M. & Faconti, G. (1999), Using hybrid automata to support human factors analysis in a critical system, *in* 'Proceedings of ERCIM workshop on Formal Methods in Industrial Critical Systems'. to appear.

Drayton, L., Chetwynd, A. & Blair, G. (1992), 'Introduction to LOTOS through a worked example', *Computer Communications* **15**(2), 70–85.

Duke, D., Barnard, P., Duce, D. & May, J. (1998), 'Syndetic modelling', *Human-Computer Interaction* **13**(4), 337–393.

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

Duke, D., Fields, B. & Harrison, M. D. (1999), 'A case study in the specification and analysis of design alternatives for a user interface', *Formal Aspects of Computing* **11**(2), 107–131.

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

Duke, D. J. & Harrison, M. D. (1994*a*), A preliminary FSM analysis of the CERD, Technical Report System Modelling: SM/IR8, Amodeus Project.

Duke, D. J. & Harrison, M. D. (1994*b*), A theory of presentations, *in* 'FME'94: Industrial Benefit of Formal Methods', number 873 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 271–290.

Duke, D. et al. (1995), The Amodeus system reference model, Technical Report System Modelling/D9, Amodeus Project.

Dwyer, M. B., Carr, V. & Hines, L. (1997), Model checking graphical user interfaces using abstractions, *in* M. Jazayeri & H. Schauer, eds, 'Software Engineering — ESEC/FSE '97', number 1301 *in* 'Lecture Notes in Computer Science', Springer, pp. 244–261.

Engberg, U. (1994), *TLP Manual (release 2.5a)*, preliminary edn.

Engberg, U. (1995), Reasoning in the Temporal Logic of Actions, PhD thesis, Department of Computer Science, University of Aarhus. submitted version.

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

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

Fields, B., Merriam, N. & Dearden, A. (1997), DMVIS: Design, modelling and validation of interactive systems, *in* Harrison & Torres (1997), pp. 29–44.

Garland, S. J. & Guttag, J. V. (1991), A guide to LP, the larch prover, Research Report 82, Digital Equipment Corporation Systems Reaserch Center, Palo Alto, California.

Gaver, B., Moran, T., MacLean, A., Lovstrand, L., Dourish, P., Carter, K. & Buxton, B. (1992), Realising a video environment: Europarc's RAVE system, *in* 'CHI '92: ACM Conference on Human Factors in Computing Systems', ACM Press, Addison-Wesley, pp. 27–35.

Guttag, J. V., Horning, J. J. et al. (1993), *Larch: Languages and Tools for Formal Specification*, Texts and Monographs in Computer Science, Springer-Verlag.

Halbwachs, N., Caspi, P., Raymond, P. & Pilaud, D. (1991), 'The synchronous data flow language LUSTRE', *Proceedings of the IEEE* **79**(9), 1305–1320.

Hall, A. (1996), 'Using formal methods to develop an ATC information system', *IEEE Software* **13**(2), 66–76.

Harel, D. (1987), 'Statecharts: A visual formalism for complex systems', *Science of Computer Programming* **8**, 231–274.

Harrison, M. D. & Duke, D. (1995), A review of formalisms for describing interactive behaviour, *in* R. Taylor & J. Coutaz, eds, 'Software Engineering and Human Computer Interaction', number 896 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 49–75.

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

Harrison, M., Fields, R. & Wright, P. C. (1996), The user context and formal specification in interactive system design (invited paper), *in* Roast & Siddiqi (1996). *htpp://www.springer.co.uk/eWiC/Workshops/FACHI/

Harrison, M. & Thimbleby, H., eds (1990), *Formal Methods in Human-Computer Interaction*, Cambridge Series on Human-Computer Interaction, Cambridge University Press.

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

Heller, D. & Ferguson, P. M. (1994), *Motif Programming Manual*, Vol. 6A of *X Window System Seris*, second edn, O'Reilly & Associates, Inc.

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

Henzinger, T. A., Ho, P.-H. & Wong-Toi, H. (1997), HyTech: A model checker for hybrid systems, *in* O. Grumberg, ed., 'Computer Aided Verification, CAV '97', number 1254 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 110–122.

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.

Hussey, A. & Carrington, D. (1998), Which widgets? Deriving implementations from formal user-interface specifications, *in* Markopoulos & Johnson (1998), pp. 206–224.

Hutchins, E. (1995), 'How a cockpit remembers its speed', *Cognitive Science* **19**, 265–288.

Jones, C. B. (1980), *Software Development: A Rigorous Approach*, Prentice-Hall International Series in Computer Science, Prentice-Hall International.

Jones, C. B. (1986), *Systematic Software Development Using VDM*, Prentice-Hall International Series in Computer Science, Prentice-Hall International.

Jones, C. B. (1992), The search for tractable ways of reasoning about programs, Technical Report UMCS-92-4-4, Department of Computer Science, University of Manchester.

Lamport, L. (1994), 'The temporal logic of actions', *ACM Transactions on Programming Languages and Systems* **16**(3), 872–923.

Lamport, L. (1995), Proving possibility properties, Research Report 137, Digital Equipment Corporation Systems Reaserch Center, Palo Alto, California.

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

Loeckx, J. & Sieber, K. (1984), *The Foundations of Program Verification*, Wiley-Teubner Series in Computer Science, John Wiley & Sons Ltd. and B. G. Teubner, Stuttgart.

Lüttgen, G. & Carreño, V. (1999), Analyzing mode confusion via model checking, Technical Report NASA/CR-1999-209332, National Aeronautics and Space Administration, Langley Research Center, Hampton, Virginia 23681-2199, USA.

Mañas, J. A. et al. (1992), *Lite User Manual*, LOTOSPHERE consortium. Ref. Lo/WP2/N0034/V08.

Manna, Z. & Pnueli, A. (1995), *Temporal Verification of Reactive Systems: Safety*, Springer.

Mantei, M. M., Baecker, R. M., Sellen, A. J., Buxton, W. A. S., Milligan, T. & Wellman, B. (1991), Experiences in the use of a media space, *in* 'Human Factors in Computing Systems: CHI '91 Conference Proceedings', ACM Press, Addison-Wesley, pp. 203–208.

Markopoulos, P. & Johnson, P., eds (1998), *Design, Specification and Verification of Interactive Systems '98*, Springer Computer Science, Eurographics, Springer-Verlag/Wien.

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

Meseguer, J. (1992), 'Conditional rewriting as a unified model of concurrency', *Theoretical Computer Science* **96**, 73–155.

Mezzanotte, M. & Paternò, F. (1996), Verification of properties of human-computer dialogues with an infinite number of states, *in* Roast & Siddiqi (1996). *htpp://www.springer.co.uk/eWiC/Workshops/FACHI/

Miller, S. P. & Potts, J. N. (1999), Detecting mode confusion through formal modeling and analysis, Technical Report NASA/CR-1999-208971, National Aeronautics

and Space Administration, Langley Research Center, Hampton, Virginia 23681-2199, USA.

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

Nelson, G. (1989), 'A generalization of dijkstra's calculus', *ACM Transactions on Programming Languages and Systems* **11**(4), 517–561.

Newman, W. M. & Lamming, M. G. (1995), *Interactive System Design*, Addison-Wesley.

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

Nicola, R. D. & Vaandrager, F. (1990), Action versus state based logics for transition systems, *in* I. Guessarian, ed., 'Proc. Ecole de Printemps on Semantics of Concurrency', number 469 *in* 'Lecture Notes in Computer Science', Springer, Berlin, pp. 407–419.

Nigay, L. & Coutaz, J. (1993), A design space for multimodal interfaces: concurrent processing and data fusion, *in* S. Ashlund, A. Henderson, E. Hollnagel, K. Mullet & T. White, eds, 'Human Factors in Computing Systems: INTERCHI '93', IOS Press, pp. 172–178.

Norman, D. E. (1988), *The Psychology of Everyday Things*, Basic Book Inc.

Owre, S., Rushby, J. M. & Shankar, N. (1992), PVS: A prototype verification system, *in* D. Kapur, ed., 'Automated Deduction — CADE-11', number 607 *in* 'Lecture Notes in Artificial Intelligence (subseries of Lecture Notes in Computer Science)', Springer-Verlag, pp. 748–752.

Owre, S., Rushby, J. & Shankar, N. (1997), Integration in PVS: Tables, types and model checking, *in* 'Tools and Algorithms for the Construction and Analysis of

Systems (TACAS'97)', Lecture Notes in Computer Science, Springer-Verlag. To appear.

Palanque, P., Paternò, F., Bastide, R. & Mezzanote, M. (1996), Towards an integrated proposal for interactive systems design based on TLIM and ICO, *in* Bodart & Vanderdonckt (1996), pp. 162–187.

Palanque, P. & Paternò, F., eds (1998), *Formal Methods in Human-Computer Interaction*, Formal Approaches to Computing and Information Technology, Springer-Verlag London.

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

Paternò, F. D. (1995), A Method for Formal Specification and Verification of Interactive Systems, PhD thesis, Department of Computer Science, University of York.

Paternò, F. & Mezzanotte, M. (1995), Formal analysis of user and system interactions in the cerd case study, Technical Report SM/WP48, Amodeus Project.

Preece, J. et al. (1994), *Human-Computer Interaction*, Addison-Wesley.

Rajan, S., Shankar, N. & Srivas, M. (1995), An integration of model-checking with automated proof checking, *in* 'Computer Aided Verification, CAV '95', number 939 *in* 'Lecture Notes in Computer Science', Springer Verlag, pp. 84–97.

Roast, C. R. & Siddiqi, J. I., eds (1996), *Formal Aspects of the Human Computer Interface*, *electronic* Workshops in Computing, Springer-Verlag London.
*htpp://www.springer.co.uk/eWiC/Workshops/FACHI/

Rushby, J. (1995), 'Model checking and other ways of automating formal methods', Position paper for panel on Model Checking for Concurrent Programs, Software Quality Week, San Francisco.

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.

Rushby, J. M. & Stringer-Calvert, D. W. J. (1996), A less elementary tutorial for the PVS specification and verification system, Technical Report CSL-95-10, Computer Science Laboratory, SRI International.
*http://www.csl.sri.com/pvs/examples/elementary-tutorial/csl-95-10.html

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

Sage, M. & Johnson, C. W. (1999), A declarative prototyping environment for the development of multi-user safety-critical systems, *in* 'International System Safety Conference 1999'. to appear.

Sufrin, B. & He, J. (1990), Specification, analysis and refinment of interactive processes, *in* Harrison & Thimbleby (1990), chapter 6, pp. 154–200.

Tanner, P. P. & Buxton, W. A. S. (1985), Some issues in future user interface management systems (UIMS) development, *in* G. E. Pfaff, ed., 'User Interface Management Systems', Springer-Verlag, pp. 67–79.

Thimbleby, H. (1990), *User Interface Design*, Frontier Series, ACM Press.

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

Wang, H.-M. & Abowd, G. D. (1994), A tabular interface for automated verification of event-based dialogs, Technical Report CMU-CS-94-189, Department of Computer Science, Carnegie Mellon University.

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

Young, R. M., Green, T. R. G. & Simon, T. (1989), Programmable user models for predictive evaluation of interface designs, *in* K. Bice & C. Lewis, eds, 'Human Factors in Computing Systems: CHI '89 Conference Proceedings', ACM Press, Addison-Wesley, pp. 15–19.

Appendix A

SMV model for the e-mail client

This is the SMV code resulting from compiling the e-mail client interactor models discussed in Chapter 4 with the `i2smv` tool.

A.1 Single mail agent window case

This is the model for the single mail agent window case.

```
#!/usr/theory/smv/smv
-- File generated by i2smv 1.4.1
-- 58 lines compiled.
MODULE window
VAR
  action: {show, hidew, unmap, nil, seen, update, map};
  newinfo: boolean;
  visible: boolean;
  mapped: boolean;
TRANS next(action)=nil -> ((next(newinfo)=newinfo &
      next(visible)=visible & next(mapped)=mapped))
TRANS next(action)=map -> (next(mapped) & next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=unmap -> (!next(mapped) & !next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=hidew -> (next(mapped) & !next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=show -> (next(mapped) & next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=update -> (next(mapped)=mapped & next(newinfo))
TRANS next(action)=seen -> (next(mapped)=mapped & !next(newinfo))
```

```
TRANS next(action)=map -> (!mapped)
TRANS next(action)=unmap -> (mapped)
TRANS next(action)=hidew -> (mapped & visible)
TRANS next(action)=show -> (mapped & !visible)
TRANS next(action)=seen -> (newinfo)
INIT !mapped & !visible & !newinfo & action=nil
-- ------------------------------
MODULE main
VAR
  action: {nil};
  mail: window;
  others: window;
DEFINE
  mail_changevis:= next(mail.action) in {map, unmap, show};
  others_changevis:= next(others.action) in {map, unmap, show};
TRANS next(action)=nil -> 1
TRANS !(mail.mapped & others.mapped) ->
      ((next(mail.visible)=mail.visible &
      next(others.visible)=others.visible ) | mail_changevis |
      others_changevis )
INIT action=nil
SPEC
  AG(mail.action=update -> mail.visible)
-- ------------------------------
```

A.2 Pop-up window case

This is the model for the pop-up case.

```
#!/usr/theory/smv/smv
-- File generated by i2smv 1.4.1
-- 76 lines compiled.
MODULE window
VAR
  action: {show, hidew, unmap, nil, seen, update, map};
  newinfo: boolean;
  visible: boolean;
  mapped: boolean;
TRANS next(action)=nil -> ((next(newinfo)=newinfo &
      next(visible)=visible & next(mapped)=mapped))
TRANS next(action)=map -> (next(mapped) & next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=unmap -> (!next(mapped) & !next(visible) &
```

```
      next(newinfo)=newinfo)
TRANS next(action)=hidew -> (next(mapped) & !next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=show -> (next(mapped) & next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=update -> (next(mapped)=mapped & next(newinfo))
TRANS next(action)=seen -> (next(mapped)=mapped & !next(newinfo))
TRANS next(action)=map -> (!mapped)
TRANS next(action)=unmap -> (mapped)
TRANS next(action)=hidew -> (mapped & visible)
TRANS next(action)=show -> (mapped & !visible)
TRANS next(action)=seen -> (newinfo)
INIT !mapped & !visible & !newinfo & action=nil
-- ------------------------------
MODULE popup
VAR
  action: {show, hidew, unmap, seen, dummy, map, nil, update};
  newinfo: boolean;
  visible: boolean;
  mapped: boolean;
TRANS next(action)=nil -> ((next(newinfo)=newinfo &
      next(visible)=visible & next(mapped)=mapped))
TRANS next(action)=nil -> 1
TRANS next(action)=map -> (next(mapped) & next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=unmap -> (!next(mapped) & !next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=hidew -> (next(mapped) & !next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=show -> (next(mapped) & next(visible) &
      next(newinfo)=newinfo)
TRANS next(action)=dummy -> (next(mapped)=mapped & next(newinfo))
TRANS next(action)=seen -> (next(mapped)=mapped & !next(newinfo))
TRANS next(action)=map -> (!mapped)
TRANS next(action)=unmap -> (mapped)
TRANS next(action)=hidew -> (mapped & visible)
TRANS next(action)=show -> (mapped & !visible)
TRANS next(action)=seen -> (newinfo)
TRANS next(action)=update -> (next(mapped) & next(newinfo) &
      next(visible))
INVAR !action=dummy
INIT !mapped & !visible & !newinfo & action=nil
INIT action=nil
-- ------------------------------
MODULE main
VAR
  action: {nil};
```

```
  alert: popup;
  mail: window;
  others: window;
DEFINE
  mail_changevis:= next(mail.action) in {map, unmap, show};
  others_changevis:= next(others.action) in {map, unmap, show};
  alert_changevis:= next(alert.action) in {map, unmap, show};
TRANS next(action)=nil -> 1
TRANS !((mail.mapped & others.mapped) | (mail.mapped & alert.mapped) |
      (others.mapped & alert.mapped) ) ->
      ((next(mail.visible)=mail.visible &
      next(others.visible)=others.visible &
      next(alert.visible)=alert.visible ) | mail_changevis |
      others_changevis | alert_changevis )
INIT action=nil
-- ------------------------------
```

Appendix B

Checkable interactor model for the MCP

This is the MCP model discussed in Chapter 4. This text can be translated with the `i2smv` tool for analysis in SMV.

```
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)
  [enterAC] pitchMode'=ALT_CAP & !ALT'
  (ALT & pitchMode!=ALT_CAP & (ALTDial.needle - plane.altitude)<=2) \
                                                    -> obl(enterAC)
  pitchMode=VERT_SPD -> plane.climbRate=crDial.needle
  pitchMode=IAS -> plane.airSpeed=asDial.needle
  pitchMode=ALT_HLD -> plane.climbRate=0
  pitchMode=ALT_CAP -> plane.climbRate=1
  ALTDial.needle < 5
  (pitchMode=ALT_CAP & plane.altitude=ALTDial.needle) -> obl(enterAH)
  [] plane.altitude = 0
fairness
  !action=nil
```

Appendix C

SMV model for the MCP

This is the SMV code resulting from compiling the MCP interactor model discussed in Chapter 4 with the `i2smv` tool. The original interactor model is presented in Appendix B.

```
#!/usr/theory/smv/smv
-- File generated by i2smv 1.4.1
-- 76 lines compiled.
--  omitted...
MODULE dialVelocity
VAR
  action: {set_5, nil, set_0, set_1, set_2, set_3, set_4};
  needle: {0, 1, 2, 3, 4, 5};
TRANS next(action)=nil -> ((next(needle)=needle))
TRANS next(action)=set_0 -> (next(needle)=0)
TRANS next(action)=set_1 -> (next(needle)=1)
TRANS next(action)=set_2 -> (next(needle)=2)
TRANS next(action)=set_3 -> (next(needle)=3)
TRANS next(action)=set_4 -> (next(needle)=4)
TRANS next(action)=set_5 -> (next(needle)=5)
INIT action=nil
-- ------------------------------
MODULE airplane
VAR
  action: {nil, fly};
  climbRate: {-1, 0, 1};
  altitude: {0, 1, 2, 3, 4, 5};
  airSpeed: {0, 1, 2, 3, 4, 5};
TRANS next(action)=nil -> ((next(climbRate)=climbRate &
      next(altitude)=altitude & next(airSpeed)=airSpeed))
```

```
TRANS (altitude>0 & altitude<5)  -> (next(action)=fly ->
      (((next(altitude)>=altitude - 1 & next(altitude)<=altitude + 1) &
      (next(altitude)<altitude -> next(climbRate)<0) &
      (next(altitude)=altitude -> next(climbRate)=0) &
      (next(altitude)>altitude -> next(climbRate)>0))))
TRANS altitude=0  -> (next(action)=fly -> (((next(altitude)>=altitude &
      next(altitude)<=altitude + 1) & (next(altitude)=altitude ->
      next(climbRate)=0) & (next(altitude)>altitude ->
      next(climbRate)>0))))
TRANS altitude=5  -> (next(action)=fly ->
      (((next(altitude)>=altitude - 1 & next(altitude)<=altitude) &
      (next(altitude)<altitude -> next(climbRate)<0) &
      (next(altitude)=altitude -> next(climbRate)>=0))))
INIT action=nil
FAIRNESS
  !action=nil
-- ------------------------------
MODULE dialClimbRate
VAR
  action: {set_-1, nil, set_0, set_1};
  needle: {-1, 0, 1};
TRANS next(action)=nil -> ((next(needle)=needle))
TRANS next(action)=set_-1 -> (next(needle)=-1)
TRANS next(action)=set_0 -> (next(needle)=0)
TRANS next(action)=set_1 -> (next(needle)=1)
INIT action=nil
-- ------------------------------
MODULE main
VAR
  action: {enterAC, nil, toggleALT, enterIAS, enterAH, enterVS};
  plane: airplane;
  ALTDial: dialAltitude;
  asDial: dialVelocity;
  crDial: dialClimbRate;
  ALT: boolean;
  oblenterAH: boolean;
  pitchMode: {VERT_SPD, IAS, ALT_HLD, ALT_CAP};
  oblenterAC: boolean;
TRANS next(action)=nil -> ((next(ALT)=ALT & next(pitchMode)=pitchMode)
      | next(asDial.action)=set_0 | next(asDial.action)=set_1 |
      next(asDial.action)=set_2 | next(asDial.action)=set_3 |
      next(asDial.action)=set_4 | next(asDial.action)=set_5 |
      next(crDial.action)=set_-1 | next(crDial.action)=set_0 |
      next(crDial.action)=set_1 | next(ALTDial.action)=set_0 |
      next(ALTDial.action)=set_1 | next(ALTDial.action)=set_2 |
      next(ALTDial.action)=set_3 | next(ALTDial.action)=set_4 |
      next(ALTDial.action)=set_5)
```

```
TRANS next(asDial.action)=set_0 ->
      (next(pitchMode)=IAS & next(ALT)=ALT)
TRANS next(asDial.action)=set_1 ->
      (next(pitchMode)=IAS & next(ALT)=ALT)
TRANS next(asDial.action)=set_2 ->
      (next(pitchMode)=IAS & next(ALT)=ALT)
TRANS next(asDial.action)=set_3 ->
      (next(pitchMode)=IAS & next(ALT)=ALT)
TRANS next(asDial.action)=set_4 ->
      (next(pitchMode)=IAS & next(ALT)=ALT)
TRANS next(asDial.action)=set_5 ->
      (next(pitchMode)=IAS & next(ALT)=ALT)
TRANS next(crDial.action)=set_-1 ->
      (next(pitchMode)=VERT_SPD & next(ALT)=ALT)
TRANS next(crDial.action)=set_0 ->
      (next(pitchMode)=VERT_SPD & next(ALT)=ALT)
TRANS next(crDial.action)=set_1 ->
      (next(pitchMode)=VERT_SPD & next(ALT)=ALT)
TRANS next(ALTDial.action)=set_0 ->
      (next(pitchMode)=pitchMode & next(ALT))
TRANS next(ALTDial.action)=set_1 ->
      (next(pitchMode)=pitchMode & next(ALT))
TRANS next(ALTDial.action)=set_2 ->
      (next(pitchMode)=pitchMode & next(ALT))
TRANS next(ALTDial.action)=set_3 ->
      (next(pitchMode)=pitchMode & next(ALT))
TRANS next(ALTDial.action)=set_4 ->
      (next(pitchMode)=pitchMode & next(ALT))
TRANS next(ALTDial.action)=set_5 ->
      (next(pitchMode)=pitchMode & next(ALT))
TRANS next(action)=enterVS ->
      (next(pitchMode)=VERT_SPD & next(ALT)=ALT)
TRANS next(action)=enterIAS -> (next(pitchMode)=IAS & next(ALT)=ALT)
TRANS next(action)=enterAH -> (next(pitchMode)=ALT_HLD & next(ALT)=ALT)
TRANS next(action)=toggleALT ->
      (next(pitchMode)=pitchMode & next(ALT)=!ALT)
TRANS next(action)=enterAC -> (next(pitchMode)=ALT_CAP & !next(ALT))
TRANS next(action)=enterAC ->
      ((ALT & (ALTDial.needle - plane.altitude)<=2))
TRANS next(action)=enterAC -> !next(oblenterAC)
TRANS next(action)!=enterAC -> next(oblenterAC)=((next(ALT) &
      next(pitchMode)!=next(ALT_CAP) &
      (next(ALTDial.needle) - next(plane.altitude))<=2)  | oblenterAC)
TRANS next(action)=enterAH -> !next(oblenterAH)
TRANS next(action)!=enterAH ->
      next(oblenterAH)=((next(pitchMode)=next(ALT_CAP) &
      next(plane.altitude)=next(ALTDial.needle))  | oblenterAH)
```

```
INVAR pitchMode=VERT_SPD -> plane.climbRate=crDial.needle
INVAR pitchMode=IAS -> plane.airSpeed=asDial.needle
INVAR pitchMode=ALT_HLD -> plane.climbRate=0
INVAR pitchMode=ALT_CAP -> plane.climbRate=1
INVAR ALTDial.needle < 6
INIT plane.altitude = 0 & action=nil & !oblenterAC & !oblenterAH
FAIRNESS
  !action=nil & !oblenterAC & !oblenterAH
-- ------------------------------
MODULE dialAltitude
VAR
  action: {set_5, nil, set_0, set_1, set_2, set_3, set_4};
  needle: {0, 1, 2, 3, 4, 5};
TRANS next(action)=nil -> ((next(needle)=needle))
TRANS next(action)=set_0 -> (next(needle)=0)
TRANS next(action)=set_1 -> (next(needle)=1)
TRANS next(action)=set_2 -> (next(needle)=2)
TRANS next(action)=set_3 -> (next(needle)=3)
TRANS next(action)=set_4 -> (next(needle)=4)
TRANS next(action)=set_5 -> (next(needle)=5)
INIT action=nil
-- ------------------------------
```

Appendix D

perceptualASI PVS theory

This is the complete PVS theory "perceptualASI" used in Section 5.3 (Chapter 5). Note that "SalmonExtent" is now an interpreted constant.

```
perceptualASI : THEORY
  BEGIN
  ASSUMING
   IMPORTING ASI
   ENDASSUMING

  Angle : TYPE = nonneg_real
  ASINeedle : TYPE = [# posn : Angle#]
  ASISpeedBug : TYPE = [# posn : Angle, extent : Angle#]
  ASISpeedBugs : TYPE = sequence[ASISpeedBug]
  ASIScale : TYPE = [# interpret : [Angle → nonneg_real] #]
  ASIInstrument : TYPE = [# needle : ASINeedle,
                            bugs : ASISpeedBugs,
                            salmonbug : ASISpeedBug,
                            scale : ASIScale
                          #]

  ScaleFactor : posreal
  BugExtent : Angle
  SalmonExtent : Angle = Ssafe / ScaleFactor

  bug_posn_extent : AXIOM ∀ (bug : ASISpeedBug) : posn(bug) − extent(bug) ≥ 0

  asi : VAR ASIInstrument
  abs_asi : VAR AbstractASI
  inv_ASIInstrument : AXIOM ∀ (i, j : nat) :
    i < j ⇒ (posn(bugs(asi)(i)) > posn(bugs(asi)(j)) ∧
              posn(bugs(asi)(i)) − extent(bugs(asi)(i))
                > posn(bugs(asi)(j)) + extent(bugs(asi)(j)))
```

```
cc_bugs : AXIOM ∀ (i : nat) :
    ((posn(bugs(asi)(i)) × ScaleFactor ≤ Vc(abs_asi)) ∧
       (¬ ∃ (j : nat) :
          j < i ∧ (posn(bugs(asi)(j)) × ScaleFactor ≤ Vc(abs_asi)))) ⇒ i = Cc(abs_asi)

rho_Needle(v : Speed) : ASINeedle = (#posn := v / ScaleFactor#)

rho_BugSeq(s : Speeds) : ASISpeedBugs =
  λ (i : nat) : (#posn := s(i) / ScaleFactor, extent := BugExtent#)

rho_Salmon(v : Speed) : ASISpeedBug =
                (#posn := v / ScaleFactor, extent := SalmonExtent#)

rho_Scale : ASIScale = (#interpret := λ (a : Angle) : ScaleFactor × a#)

ρ(a : AbstractASI) : ASIInstrument = (#needle := rho_Needle(Vc(a)),
                                      bugs := rho_BugSeq(Smm(a)),
                                      salmonbug := rho_Salmon(Vref(a)),
                                      scale := rho_Scale
                                    #)

in_arc(needle : ASINeedle, astart, aend : Angle) : bool =
          astart ≤ posn(needle) ∧ posn(needle) ≤ aend

next_counterclockwise : [[ASINeedle, ASISpeedBugs] → ASISpeedBug]
next_counterclockwise : AXIOM
    ∀ (needle : ASINeedle, bugs : ASISpeedBugs, bug : ASISpeedBug) :
     (∃ (i : nat) : posn(bugs(i)) ≤ posn(needle)) ⇒
       (next_counterclockwise(needle, bugs) = bug ⇔
         (∃ (i : nat) : bugs(i) = bug ∧ posn(bugs(i)) ≤ posn(needle) ∧
           (∀ (j : nat) : j < i ⇒ posn(bugs(j)) > posn(needle))))

getCurrentBug(needle : ASINeedle, bugs : ASISpeedBugs) :
  ASISpeedBug = next_counterclockwise(needle, bugs)

salmonBugCheck(needle : ASINeedle, bug : ASISpeedBug) : bool =
  in_arc(needle, posn(bug) − extent(bug), posn(bug) + extent(bug))

configBugCheck(needle : ASINeedle, bug : ASISpeedBug) : bool =
  in_arc(needle, posn(bug), posn(bug) + Smargin / ScaleFactor)

asiConfigCheck(asi : ASIInstrument) : bool =
  configBugCheck(needle(asi), getCurrentBug(needle(asi), bugs(asi)))

END perceptualASI
```

Appendix E

Checkable interactor model for ECOM

This is the interactor model for the ECOM case-study in Chapter 6. The text presented is the input code for the `i2smv` tool.

```
define
  null = 0
  user1 = 1
  user2 = 2
  message  = 1
  connect  = 2
  never      = 1
  always     = 2
  whenopen   = 3
  whenlocked = 4
  open   = 1
  locked = 2
types
  User = {user1, user2}
  Service = {message, connect}
  Door = {open, locked}
  Exception = {never, always, whenopen, whenlocked}
#  Conn = User x Service x User
  PConn = array user1..user2 of array message..connect of \
                                      array user1..user2 of boolean
# PConn = P Conn
  UserToDoor = array user1..user2 of {open, locked}
# UserToDoor = User -> Door
```

```
  ConnToException = array user1..user2 of \
                          array message..connect of \
                                array user1..user2 of \
                                      {never, always, whenopen, \
                                                    whenlocked, null}
# ConnToException = Conn -/-> Exception
  ExceptionToPDoor = array never..whenlocked of \
                           array open..locked of boolean
# ExceptionToPDoor = Exception -> P Door
  DoorToPService = array open..locked of \
                           array message..connect of boolean
# ------------------------------------------------------------
interactor core
attributes
  current: PConn
  allowed: PConn
  availability: UserToDoor
  accessexcep: ConnToException
actions
  establish(User,Service,User) close(User,Service,User) \
  setexcep(User,Service,User,Exception) setavail(User,Door)
axioms
# (1)
  allowed[caller][s][callee] -> [establish(caller,s,callee)] \
      current[caller][s][callee]' = true \
    & current<caller><s><callee>'=current<caller><s><callee> \
    & nochg(availability[], accessexcep[])
  !allowed[caller][s][callee] -> [establish(caller,s,callee)] \
      nochg(current[], availability[], accessexcep[])
# (2)
   current[caller][s][callee] -> [close(caller,s,callee)] \
      current[caller][s][callee]' = false \
    & nochg(current<caller><s><callee>, availability[], accessexcep[])
   !current[caller][s][callee] -> [close(caller,s,callee)] \
      nochg(current[], availability[], accessexcep[])
# (3)
  [setavail(u,d)] availability[u]'=d \
                & nochg(availability<u>, current[], accessexcep[])
# (4)
  [setexcep(o,s,t,e)] accessexcep[t][s][o]'=e \
               & nochg(accessexcep<t><s><o>, current[], availability[])
# (5)
  allowed[user1][message][user1]
  allowed[user1][message][user2]
  allowed[user1][connect][user1]
  allowed[user1][connect][user2] <-> \
      ((accessexcep[user1][connect][user2]!=null \
```

```
        -> (  (accessexcep[user1][connect][user2]=always) \
            | (accessexcep[user1][connect][user2]=whenopen \
               & availability[user2]=open) \
            | (accessexcep[user1][connect][user2]=whenlocked) \
           ) \
       ) \
       & \
       (accessexcep[user1][connect][user2]=null \
        -> (availability[user2]=open) \
       ) \
      )
  allowed[user2][message][user1]
  allowed[user2][message][user2]
  allowed[user2][connect][user1] <-> \
      ((accessexcep[user2][connect][user1]!=null \
        -> (  (accessexcep[user2][connect][user1]=always) \
            | (accessexcep[user2][connect][user1]=whenopen \
               & availability[user1]=open) \
            | (accessexcep[user2][connect][user1]=whenlocked) \
           ) \
       ) \
       & \
       (accessexcep[user2][connect][user1]=null \
        -> (availability[user1]=open) \
       ) \
      )
  allowed[user2][connect][user2]
# (6) hard-coded...
# (7) hard-coded...
  []  !current[user1][message][user1] \
    & !current[user1][message][user2] \
    & !current[user1][connect][user1] \
    & !current[user1][connect][user2] \
    & !current[user2][message][user1] \
    & !current[user2][message][user2] \
    & !current[user2][connect][user1] \
    & !current[user2][connect][user2] \
    & availability[user1]=open & availability[user2]=open \
    & accessexcep[user1][message][user1]=null \
    & accessexcep[user1][message][user2]=null \
    & accessexcep[user1][connect][user1]=null \
    & accessexcep[user1][connect][user2]=null \
    & accessexcep[user2][message][user1]=null \
    & accessexcep[user2][message][user2]=null \
    & accessexcep[user2][connect][user1]=null \
    & accessexcep[user2][connect][user2]=null
```

```
interactor button
attributes
  enabled: boolean
actions
  press
axioms
  per(press) -> enabled
  [press] enabled' = enabled

interactor main
importing
  core
includes
  button via buttons_message
  button via buttons_connect
attributes
  owner: User
  callee: User
  door_icon: Door
actions
  select(User)
axioms
# (1)
  callee=user1 -> (door_icon=availability[user1])
  callee=user2 -> (door_icon=availability[user2])
# (2)
  callee=user1 -> \
    ((  buttons_message.action=press \
      & allowed[user1][message][user1]) -> action=establish_1_1_1)
  callee=user1 -> \
    ((  buttons_connect.action=press \
      & allowed[user1][connect][user1]) -> action=establish_1_2_1)
  callee=user2 -> \
    ((  buttons_message.action=press \
      & allowed[user1][message][user2]) -> action=establish_1_1_2)
  callee=user2 -> \
    ((  buttons_connect.action=press \
      & allowed[user1][connect][user2]) -> action=establish_1_2_2)
# (3)
  [select(u)] callee'=u \
            & nochg(current[], availability[], accessexcep[])
# (4)
  [establish(caller,s,cal)] nochg(callee)
# (5)
  [close(caller,s,cal)] nochg(callee)
# (6)
  [setavail(u,d)] nochg(callee)
```

```
# (7)
  [setexcep(o,s,t,e)]  nochg(callee)
# initial state
  [] buttons_message.enabled & buttons_connect.enabled & callee=user2
  owner = user1
test
  AG(door_icon=open -> AX( buttons_connect.action=press -> \
        (  (callee = user1 -> current[user1][connect][user1]) \
         & (callee = user2 -> current[user1][connect][user2]))))
```

Appendix F

PVS theories for ECOM

These are PVS theories for the ECOM case-study in Chapter 6.

```
ecom : THEORY
  BEGIN

  IMPORTING restrictedpanel, rho_restrictedpanel

  rho_availability : [door → availability_panel] =
    λ (d : door) :
      COND
      d = open →
      (#state := λ (di : door_icon) : di = open_door,
         position := availability_panel_order#),
      d = ajar →
      (#state := λ (di : door_icon) : di = ajar_door,
         position := availability_panel_order#),
      d = closed →
      (#state := λ (di : door_icon) : di = closed_door,
         position := availability_panel_order#),
      d = locked →
      (#state := λ (di : door_icon) : di = locked_door,
         position := availability_panel_order#)
      ENDCOND

  rho_exceptions : [exception → exception_panel] =
    λ (e : exception) :
      COND
      e = never →
      (#state := λ (ei : exception_item) : ei = never,
         position := exception_panel_order#),
      e = whendooropen →
      (#state := λ (ei : exception_item) : ei = when_open,
         position := exception_panel_order#),
      e = whendoorajar →
```

```
      (#state := λ (ei : exception_item) : ei = when_ajar,
         position := exception_panel_order#),
      e = whendoorclosed →
      (#state := λ (ei : exception_item) : ei = when_closed,
         position := exception_panel_order#),
      e = whendoowlocked →
      (#state := λ (ei : exception_item) : ei = when_locked,
         position := exception_panel_order#),
      e = always →
      (#state := λ (ei : exception_item) : ei = always,
         position := exception_panel_order#)
      ENDCOND

   ρ : [restrictedpanel.attributes → rho_restrictedpanel.attributes] =
     (λ (rp : restrictedpanel.attributes) :
       (#availability := rho_availability(availability(rp)),
          exceptions := rho_exceptions(accessexcep(rp))#))

   equivalence : THEOREM
      ∀ (rp : restrictedpanel.attributes) : willsucceed(rp) = rho_willsucceed(ρ(rp))

   equivalence_correct : THEOREM
      ∀ (rp : restrictedpanel.attributes) : willsucceed(rp) = rho_willsucceed(ρ(rp))

   END ecom


restrictedpanel : THEORY
   BEGIN

   door : TYPE = {open, ajar, closed, locked}

   exception : TYPE = {never, always, whendooropen,
                       whendoorajar, whendoorclosed, whendoorlocked}

   attributes : TYPE = [# availability : door, accessexcep : exception#]

   exceplevel : [exception → setof[door]] =
     λ (e : exception) :
      COND
      e = never → {d : door | FALSE},
      e = always →
      {d : door |
      d = open∨
        d = ajar ∨ d = closed ∨ d = locked},
      e = whendooropen → {d : door | d = open},
      e = whendoorajar → {d : door | d = open ∨ d = ajar},
      e = whendoorclosed →
      {d : door |
      d = open ∨ d = ajar ∨ d = closed},
      e = whendoorlocked →
      {d : door |
      d = open∨
```

```
      d = ajar∨
        d = closed ∨ d = locked}
    ENDCOND

  willsucceed : [attributes → bool] =
    λ (callee : attributes) : exceplevel(accessexcep(callee))(availability(callee))

  END restrictedpanel


rho_restrictedpanel : THEORY
  BEGIN

  door_icon : TYPE = {open_door, ajar_door, closed_door, locked_door}

  IMPORTING radio_box[door_icon]

  availability_panel : TYPE = radio_box[door_icon].selected_radiobox_panel

  exception_item : TYPE = {always, when_open, when_ajar, when_closed, when_locked, never}

  IMPORTING radio_box[exception_item]

  exception_panel : TYPE = radio_box[exception_item].selected_radiobox_panel

  availability_panel_order : [door_icon → nat] =
    λ (di : door_icon) :
      COND
      open_door?(di) → 1,
      ajar_door?(di) → 2,
      closed_door?(di) → 3,
      locked_door?(di) → 4
      ENDCOND

  exception_panel_order : [exception_item → nat] =
    λ (ei : exception_item) :
      COND
      always?(ei) → 1,
      when_open?(ei) → 2,
      when_ajar?(ei) → 3,
      when_closed?(ei) → 4,
      when_locked?(ei) → 5,
      never?(ei) → 6
      ENDCOND

  isabove : pred[[exception_panel, exception_item, exception_item]] =
    λ (ep : exception_panel, i1 : exception_item, i2 : exception_item) :
      position(ep)(i1) < position(ep)(i2)

  attributes : TYPE = [# availability : availability_panel, exceptions : exception_panel#]

  maptoexceplist : [door_icon → exception_item] =
    λ (di : door_icon) :
```

```
    COND
    open_door?(di) → when_open,
    ajar_door?(di) → when_ajar,
    closed_door?(di) → when_closed,
    locked_door?(di) → when_locked
    ENDCOND

  identify_avail : [availability_panel → door_icon] =
    λ (ap : availability_panel) :
    COND
    state(ap)(open_door) → open_door,
    state(ap)(ajar_door) → ajar_door,
    state(ap)(closed_door) → closed_door,
    state(ap)(locked_door) → locked_door
    ENDCOND

  identify_excep : [exception_panel → exception_item] =
    λ (ep : exception_panel) :
    COND
    state(ep)(always) → always,
    state(ep)(when_open) → when_open,
    state(ep)(when_ajar) → when_ajar,
    state(ep)(when_closed) → when_closed,
    state(ep)(when_locked) → when_locked,
    state(ep)(never) → never
    ENDCOND

  rho_willsucceed : [attributes → bool] =
    λ (a : attributes) :
      (maptoexceplist(user_avail) = user_excep∨
        isabove(exceptions(a), maptoexceplist(user_avail), user_excep))
        WHERE
        user_avail : door_icon = identify_avail(availability(a)),
        user_excep : exception_item = identify_excep(exceptions(a))

  END rho_restrictedpanel
```