Formally Verifying Interactive Systems: A Review*

José C. Campos & Michael D. Harrison

Human-Computer Interaction Group
Department of Computer Science, University of York
Heslington, York YO1 5DD, U.K.
e-mail: {jfc,mdh}@cs.york.ac.uk

Abstract. Although some progress has been made in the development of principles to guide the designers of interactive systems, ultimately the only proven method of checking how usable a particular system is must be based on experiment. However, it is also the case that changes that occur at this late stage are very expensive. The need for early design checking increases as software becomes more complex and is designed to serve volume international markets and also as interactions between operators and automation in safety-critical environments becomes more complex. This paper reviews progress in the area of formal verification of interactive systems and proposes a short agenda for further work.

1 Introduction

Although some progress has been made in the development of principles to guide the designers of interactive systems (see for example principles suggested in [25,8]), ultimately the only proven method of checking how usable a particular system is must be based on experiment. Successful systems have evolved over time, using experiments with prototypes through trial and error [17]. Part of the reason for this is, of course, that systems must be judged in work context. The effect of a particular design using particular interaction principles can in the end only be judged when the system is used in a typical work environment.

However, it is also the case that changes that occur at this late stage are very expensive and any early testing of a design through verification against design principles may have the effect of reducing the cost of changes late in the design process. In practice it is very difficult to check that a system captures properties that correspond to the design principles. This paper reviews progress in this area and proposes a short agenda for further work. The need for early design checking increases as software becomes more complex and is designed to serve volume international markets and also as interactions between operators and automation in safety-critical environments becomes more complex. The challenge then is to build interactive systems that are "correct by design".

Proving that the design of a software system is correct is not possible in abstract since correctness is a relative concept. What we can do is formally verify that the specification has some required properties. Work in formal verification of software has been

* Published In M. D. Harrison and J. C. Torres, editors, *Design, Specification and Verification of Interactive Systems '97*, Springer Computer Science, pages 109–124. Eurographics, Springer-Verlag/Wien, June 1997.

traditionally concerned with two issues: the verification of implementations against their specifications and, particularly in the context of concurrent systems, that certain properties of the specification hold — that the system is free from deadlock, that the axioms of the specification are consistent and so on. Two main techniques for proving these properties are supported by automation: Model Checking and Theorem Proving. Interactive Systems have interesting characteristics that mean that both general system specification techniques and specific techniques relevant to dealing with concurrency properties are appropriate, but they pose a set of new and specific problems. We can think of an Interactive System as a heterogeneous system. On one side of the interface we have software with a fixed and predetermined behaviour while on the other we have humans, with flexible, adaptable and ultimately non-deterministic behaviour. It is the coupling of these two distinct entities that give Interactive Systems their special nature.

In this paper we start by analysing (section 2) what type of properties are of interest in Interactive Systems and establish a framework for the classification of such properties. Having done so, we go on to analyse the available approaches to the formal verification of Interactive Systems' specifications. In all, we have identified four approaches that use automated techniques: three using Model Checking (section 3) and one using Theorem Proving (section 4). We compare and relate these different approaches and establish an agenda for further work with particular emphasis on the role of hybrid specification techniques such as those developed at York [10] (section 5).

2 A Framework for the Classification of User Interface Properties

Formal verification techniques have been used in program verification, as well as in specification verification. While the tools used are basically the same, the two approaches tackle different aspects of the formal development of software. Program verification starts with a program and its specification and, given a formally defined semantics for the programming language, tries to prove that the program satisfies the specification. Specification verification has to do with proving that the specification itself has desired properties. The research on the formal verification of Interactive Systems builds on results from these fields, our particular interest being focused on the latter issue. Interactive Systems, however, raise a set of new problems and questions. The fundamental factor that differentiates Interactive Systems from other software systems is the human factor. An Interactive System is usually a mediator between humans and an underlying physical system (or some logical representation of it). Typically the humans (the users) will want to influence the underlying system, and will do it through the Interactive System. In order to enable this, the Interactive System must: support users in the execution of their tasks, present users with accurate representations of the underlying system and of the interface state (for example, mode), and minimize the interference of the interface on the performance of tasks.

These general requisites may be refined to more concrete properties that can be verified of an Interactive System. In the end, however, some properties will be more appropriate for some systems, while other properties will be more appropriate for other systems. So, what we really should look for is a framework identifying the entities involved and the classes of general properties that will make them more usable or less human error prone.

We have already seen that we have *users* interacting with an *underlying system* through a *user interface*. These will be our three entities of concern. In order to analyse the interaction between them, we must identify the mechanisms used in the interaction. Two basic mechanisms of interaction are *events* and *status phenomena* [7]. They are used by the interface to provide information to the user, and by the user to manipulate the interface. Typically the user will invoke specific combinations of events and/or status phenomena in order to achieve its goals. The set of strategies available to achieve a given goal is called a *task*. The way the User Interface reacts to events or status phenomena might change depending on its state, and in some cases one Interactive System will provide access to more than one underlying system. When this happens, we say the User Interface has different *modes* of interaction. We can now identify four interaction mechanisms: Events, Status Phenomena, Tasks and Modes.

Finally, what properties should we consider when analysing an Interactive System specification? We want to keep a high level of abstraction, not going into too much detailed analysis, in order to be able think about Interactive Systems in general. We will identify three different classes of properties. The first class we call *visibility*. Visibility concerns what is shown at the user interface, how it is shown, and how the users perceive it. Visibility includes questions like: "*do events have appropriate feedback?*", or "*will the user correctly perceive the displayed information?*".

The second class of properties we call *reachability*. Reachability 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 users' way of doing it. Reachability includes questions like: "*can the effects of actions be undone?*" or "*how does the way a task is modelled at the user interface match the users' mental model of the same task?*".

These two classes of properties have to do with what can be seen, and what can be done (and how). We are also interested in the behaviour of the Interactive System and in properties of its state like: "*does the same event always have the same effect in a given mode?*" or "*does some predicate on the state of the system always hold?*". This type of questions does not directly analyses the interaction between users and user interface, but how the user interface and the underlying system work. We will call these the *reliability* properties.

In figure 1 we summarize the framework. Basically its shows what classes of properties we want to prove of the interaction between the different entities of a given Interactive System specification.

Entities	User	User Interface	Underlying System	
Interaction Mechanisms	Events	Status phenomena	Tasks	Mode
Properties	Visibility	Reachability	Reliability	

Fig. 1. The framework

3 Model Checking

The first approaches to formal verification of Interactive Systems, where based on Model Checking technology (see [1] and [23]).

Model checking was originally proposed as an alternative to the use of theorem provers in concurrent program verification [5]. 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. By using an algorithm to perform exhaustive state space analysis, the two main drawbacks of theorem provers were avoided:

- the analysis is fully automated (as opposed to a 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 somewhat diminishes this problem. Avoiding the explicit representation of states and exploiting state space structural regularity, enables the analysis of state spaces that might be as big as 10^{20} states [4]. Unfortunately software specifications are usually not as regular as hardware ones. Furthermore, the problem remains that some systems might not be specifiable by a finite state machine at all.

We will now briefly describe and compare the two above mentioned approaches: Abowd, Wang & Monk's approach [26,1] and Paternó's approach [23]. In order to describe, and afterwards compare, the two approaches, we will be focusing on three main aspects: how the user interface is specified, how that specification translates into some kind of finite state machine, and finally how the resulting finite state machine description can be analysed.

3.1 Using SMV

In this approach, Abowd, Wang and Monk [1] combine the simplicity of Action Simulator (AS) with the power of the Symbolic Model Verifier tool (SMV). The user interface is specified using AS and then translated into the SMV input language, then the specification is analysed in SMV using Computational Tree Logic (CTL) formulae [5].

The Dialogue Specification As we said above, the user interface is specified using Action Simulator. AS is a spreadsheet package that allows for the PPS[1] specification of dialogues in a tabular fashion. The tool additionally has dialogue simulation capabilities: the specification can be executed allowing the designer to observe its behaviour.

To specify a dialogue in PPS, we must identify the user actions (the events) and a set of fields (or conditions). Each field represents some information on the system state and at each state a field can only have one value. Thus, input is event based, while output is status based.

In figure 2 (adapted from [20]) we can see the specification of a very simple photocopier: rows correspond to events and columns to fields, the first row is the state of the system.

[1] Propositional Production System.

	Conditions			No. rules = 6 No. conds = 3
	A4	*Black*	*Single copy*	
State	TRUE	TRUE	TRUE	
Request A3 ********	TRUE FALSE			
Request A4	FALSE TRUE			
Request Red ********		TRUE FALSE		
Request Black		FALSE TRUE		
Req. >1 copies ********			TRUE FALSE	
Reset Copies			FALSE TRUE	

Fig. 2. PPS specification using Action Simulator

The dialogue is specified by associating with each event pre- and post-condition pairs. Pre-conditions are written on the top side of the rows, and post-conditions on the bottom side. The pre-condition of an event defines the combination of fields' 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. Blank fields in pre-/post-conditions mean, respectively, don't care/don't change values. In our example, event $RequestA3$ is enabled, as its pre-condition is verified by the present state. If the user chooses to generate this event, the field $A4$ will be set to $false$ and the other fields will be left unchanged (see post-condition of $RequestA3$).

We can view a PPS specification 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 $T \subseteq A \times (\Sigma \to \Sigma)$, in which each member specifies a rule;
- $P : \Sigma \to \mathcal{V}^F$ 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 T that the events are labels on the transitions from state to state.

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 relations between states, and must be a total relation;
- $P : S \to 2^{AP}$ assigns to each state the set of atomic propositions (AP) true in that state.

From this definition it follows that in the translation process from PPS to SMV, the transitions' labels are lost: the transitions in the CTL machine are not labeled. This would mean losing the information on which event caused which state change. The problem is overcome by including the events as state information. This way each CTL state represents the state of the system and a possible next event in the dialogue. This means that each state in the PPS specification will be represented by n states in the CTL machine, one for each of the n possible events in the original PPS state (see figure 3). This means the notion of state in the PPS is different from the same notion in CTL, so we must be careful when talking about PPS states during verification in SMV.

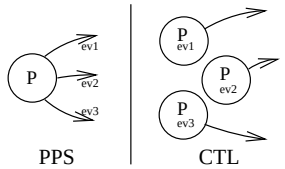

Fig. 3. From one PPS state to n CTL states

Additionally, as R must be total, dialogues with deadlocks cannot be represented in CTL machines. 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.

In [26] Wang & Abowd present an algorithm for the automatic translation from PPS to CTL, they also say that they have developed a tool to make this translation an automatic process.

Checking the Specification Once the PPS specification is translated into a CTL machine, the SMV tool can be used to verify (or not) the validity of CTL formulae in the machine. Basically the CTL machine is a finite state transition machine and the CTL formulae allow us to ask questions over the execution paths of that machine. For the syntax of CTL see [26], besides the usual propositional logic connectives, CTL allows for operators over paths that enable us to write formulae of the type:

- a property is universal, inevitable, possible or impossible;
- a property must/may hold at the next step;
- property $p1$ will/may hold until property $p2$ holds.

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.

The authors propose a set of templates for testing properties with this approach. The questions that are proposed are of the type: "*can a rule somehow be enabled?*" or "*is it true that the dialogue is deadlock free?*" or "*can the user find a way to accomplish a task from initialisation?*".

Looking at the approach in the light of our framework, we start by noticing 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. If we were to think of the fields as representing only the interface, then we would have no explicit connection with the underlying system. But even then, because the user interface must reflect the underlying system in some way, and because only events manipulate the interface state, the PPS specification would be implicitly enforcing the behaviour of the underlying system. Because of this *unifi cation* between interface and underlying state, and also the lack of a mechanism for structuring the specification, there is no way of thinking about visibility properties (everything is visible), or making distinctions between actions that affect the underlying state as against actions that affect the interface.

The main properties dealt with are reachability properties and tasks are only vaguely defined as some target state or action. Reliability conditions can also be analysed, although the level of specification and SMV do not allow for very detailed analysis.

Finally, no user model is included in the specification, neither is the notion of mode of interaction.

3.2 Using The Lite Tool Set

PPS supports very simple state transition descriptions, we will now look at how a more powerful notion can be used. In [23] Paternó proposes the use of the Lite tool set to translate Interactor based specifications written in LOTOS into a finite state machine, and then analyse the finite state machine using Action-based Temporal Logic (ACTL) formulae (for an introduction to ACTL see [21]).

The Dialogue Specification In this approach the specification of the user interface is based on the interactor architecture presented in figure 4.

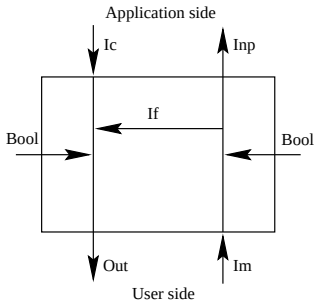

Fig. 4. LOTOS Interactor Architecture

An Interactor conveys information from the user side to the application side through channels Im and Inp (internal dialogue), and from the application side to the user side through channels Ic and Out (external dialogue). The boolean gates are triggers 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). The interactors

can be composed to construct a hierarchy, allowing for a modular specification of the interface.

After defining the architecture of the user interface using interactors, each interactor is specified in LOTOS. Because of the non deterministic way in which channels might behave, a control process may have to be introduced, at this stage, in order to constrain the dynamic behaviour of the specification.

This specification notation has clearly 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 [9].

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 by the Auto tool [18], 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 will be lost, as well as the boolean guards used to constrain behaviour. Suppose we have an OK button in a dialogue box that should only be enabled after all the entries in the dialogue box have been edited. We could model this behaviour putting a boolean guard in the button's specification. However, when translating from Full to Basic LOTOS, the guard is lost and the button becomes available at all times.

The loss of data type information does not seem to be a serious problem. Different gates can be automatically created for each data type thus creating necessary distinction, and the approach is not concerned with the system response to particular data values, but with overall behaviour.

The loss of conditional guards, on the other hand, will cause the Basic LOTOS version of the specification to be a superset of 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 [23] 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 [22] it is shown that a manual translation from Full to Basic LOTOS might further obviate this problem, but at the cost of loosing some automation.

Checking the Specification Once the specification has been translated into a FSM, we can use the Logic Checker tool [18] to analyse it. Logic Checker uses Action-based Temporal Logic (ACTL). ACTL is a branching time temporal logic that allows reasoning about the actions that a system can take. ACTL formulae can be interpreted over a Labelled Transition System - $LTS = (Q, Ac \cup \{\tau\}, \rightarrow, Q_0)$ - 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 we can see that the transitions are labeled by the actions. It should also be noticed that, as there is no explicit state information, we can only refer to dialogue properties that involve state implicitly. For instance, in ACTL we do 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?*"

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. The system integrity properties have to do with the system architecture. These properties are more directly connected with technical aspects of the specification and its consistency, than they are with properties of the user interface that is being specified, so we will discuss them no further.

Regarding user interface properties, templates are proposed for a number of properties. The properties fall into three broad classes: Reachability, Visibility, and those that are Task related. Reachability is defined in [23] as: "... given a user action, (...) it is possible to reach an effect which is described by a specific action.".

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

Relative to 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. Minimal errors are those actions after which "it is possible to get to the next state by performing one action useful for the current task" [23]. Recoverable errors are those after which several actions must be performed before a useful action for the task is possible. Unrecoverable errors are those after which it is not possible to perform the task anymore. The concept of 'task useful action', however, is not defined. This kind of information must be obtained elsewhere, possibly in the task specification.

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. 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 we cannot characterise states, we cannot express that the system returns to the previous state, this in turn means we have to rely on the notion of action that cancels effects without really formalising it.

Referring to our framework, we see that users are still not considered in the specification. The approach, however, separates the user interface from the underlying system, this enables the verification of visibility properties. However, this separation is done in such a way that there is no way to reason about the underlying system's state and its reliability properties: the underlying system's specification is not part of the Interactive System specification.

The approach is heavily based on the notion of event, and as such has no way to handle status phenomena. 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. This does not correspond to our understanding of what task is, a set of strategies available to achieve a desired goal. It is easy to see that we could have two se-

quences of actions with the same final action but corresponding to different tasks, for example: `<select_landing_mode, engage_auto_pilot>` or `<select_goaround_mode, engage_auto_pilot>`. Finally, and as in Abowd's et al. approach, mode is not explicitly represented.

3.3 Comparison

The main difference between both approaches comes from the specification notations used. Abowd et al. 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, whereas Action Simulator was designed for very high level abstract specifications [20]. At this level some of the useful properties that we want to investigate might even be not yet present. It is doubtful that we should give up so much for ease of use when the verification process, in itself, might be a complex task.

Paternó 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 of using 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. In the next section we will see how Bumbulis uses an approach that avoids this problem.

At the verification level, the main difference to be noted between both approaches has to do with the temporal logics that each approach uses. While CTL focuses on the states and the transition between states, ACTL focuses on events and the sequences of events that can be generated.

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. encode events as state information, the former analysis can also be done.

Different foci also raise different problems. ACTL has problems when we want 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 we wish 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 we can actually express 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 Paternó's approach is 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.

As a final note on the use of model checking techniques for the formal verification of interactive systems, a last approach should be mentioned. It has been proposed by d'Ausbourg et al. in [6] and uses the data flow language Lustre. This approach is similar to that of Paternó in that it uses the notion of interactor to model the user interface. In this case, however, interactors are derived from UIL descriptions, and modelled in Lustre. Verification is achieved by augmenting the interface with Lustre nodes modelling the intended properties, and using the tool Lesar to traverse the automaton generated from this new system. Generating an automaton for the conjunction of system and property specifications means that only that part of the system relevant for the property being verified is actually considered. This allows for the analysis of bigger systems than if a complete automaton was required.

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 Paternó's approach, nothing is said about how data types are handled (in [6] only boolean values are considered). Thus, some problems remain. However, the data flow nature of Lustre means status phenomena are better dealt with than in Paternó's approach. Lustre makes no mention of user related issues like tasks or mode, so it doesn't seem possible to deal with properties relating to those.

4 Theorem Proving

Having seen how Model Checking is being used in the formal verification of User Interfaces, let us now turn to the alternative approach to system verification: Theorem Proving.

Theorem Proving is a deductive approach to the verification of systems. Available systems can differ considerably regarding the way they can be used and the facilities they provide. Theorem provers range from fully interactive systems to systems that, given a proof, check if the proof is correct with no further interaction from the user. While some systems provide only a basic set of methods for manipulating the logic, giving the user full control over the proof strategy, others include complex tactics and strategies, meaning the user might not know exactly what has been done. Another commonly made distinction is that between first order and higher order logic deduction systems.

Due to this *mechanical* nature, we can trust a proof done in a theorem prover to be correct, as opposed to the recognisedly error prone manual process. While this is an advantage, it also means that doing a proof in a theorem prover can be more hard, as *every little bit* must be proved.

4.1 Using HOL

In [3] Bumbulis et al. show how they are using HOL (a Higher Order Logic Theorem Prover) in the verification of User Interface specifications.

They use a language - IL (the Interconnection Language) - to specify User Interfaces as sets of connected interface components. These specifications can then be implemented in some toolkit as well as modelled in 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 (at least) at the same level of expressiveness of the formalism used to write the specification. So, we can anticipate we will not have the translation problems of Paternó's approach.

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, output or observer), and different components can be connected through their input and output ports, much in the same way as widget's methods and callbacks 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.

A IL description (taken from [3]) of a window with a dial and a slider is shown on figure 5. The `Main` component defines the global User Interface.

```
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
}
```

Fig. 5. A IL description

The behaviour of a single component is described by a collection of code fragments defining the behaviour of its ports, and the overall behaviour of the instances of the component. The semantics of the language used to write the code fragments is defined by HOL predicates for each of the constructs of the language.

After having the User Interface specified by a set of components, a HOL model of the specification can be generated. This is done by modelling each component with a predicate. These predicates will consist of a series of conjuncts specifying the behaviour of each of the input ports and observers and also of the instances of the component. Next we show the predicate modeling the `Slider` component:

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

$$\wedge \ (c = \exists\, v.\mathsf{atomic}(\mathsf{numEv}ev)(set\ v)))$$

For a full explanation of this predicate see [3], here it is enough to say that i is the initial state, and c the behaviour of an instance of Slider.

Verifying the Specification Given the definition of all the components we can then think of verifying properties. Properties to be verified are expressed as predicates over sets of runs. A run being a sequence of event/resulting state pairs. Verifying that a model has a given property P amounts to proving the following theorem:

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

where Main is a predicate modelling the component with the same name. What the formula expresses is that for every possible initial state (i), and for every possible behaviour (c), P is a valid property of the execution of the behaviour from the initial state. That is, P is a universal property of the dialogue. It should be noticed that this is a safety property [19], and as such can, in general, be proved by a Model Checker if there are a finite number of states.

In this approach, if we are to be able to perform proofs, a logic that allows us to reason about the properties of interest must be mechanised. In the cited paper, 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.

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 a user model. Although we can imagine 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 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's et al. and visibility properties 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, reachability is also a problem.

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, and the verification process is complex.

5 Moving On...

Until now we have been looking at how the formal verification of Interactive Systems is currently approached. In figure 6 we summarize the results of our analysis.

		Abowd et al.	Paternó	d'Ausbourg et al.	Bumbulis et al.
Entities	Users	×	×	×	×
	User Interface	$\sim^a$	✓	✓	✓
	Underlying System	$\sim^a$	×	×	×
Interaction Mechanisms	Events	$\sim^b$	✓	✓	$\sim^b$
	Status phenomena	$\sim^c$	×	✓	$\sim^c$
	Task	$\sim^d$	$\sim^d$	×	×
	Mode	×	×	×	×
Properties	Visibility	×	✓	✓	×
	Reachability	✓	✓	✓	×
	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.

Fig. 6. Summary of the comparison

We will now look at what can be done in order to increase our capabilities of reasoning about specifications of Interactive Systems. We will first consider what one such specification should deliver, and then what type of tools we should use in order to analyse it.

A first conclusion to draw from Abowd's et al. 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 [14]. From the other approaches we can see that, although necessary, this separation must not be done in such a way that we loose information about the underlying systems. It is also self evident that, if we want to reason about interesting aspects of an Interactive System, a sufficiently expressive notation must be used. Aspects that have not yet deserved necessary attention are task, mode and visibility issues. Also, 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 notions of event or status phenomena, little or no attention being payed to task or mode. Further, it is important that some sort of user model be included in the specification in order to enable the analysis of the interactive system against the users' needs and capabilities.

It is our belief that York Interactors [10] are capable of delivering the expressiveness that will enable us to address these problems. York Interactors enable the homogeneous specification of both the user interface and the underlying system, and some work has been done to include models of the user in the specification [13,11].

From a technological viewpoint, we can expect to have problems if we intend to use model checking, as we have seen in Paternó's approach. On the other hand, the traditional first- and higher-order logics used by most theorem provers, do not seem to have the expressiveness that we need, namely when dealing with dynamic aspects of the dialogue between users and user interface. We intend, then, to study the use of Temporal Logic theorem provers and how they can be used in the analysis of York Interactor based

specifications. At the moment we are considering the use of TLP [12] (an extension of the Larch Prover [15] to include TLA, the Temporal Logic of Actions [16]), PVS [24], and STeP (the companion system of the book by Manna and Pnueli [19]). These two last systems, in particular, by combining the power of theorem proving with the automated analysis of model checking, seem prajanomising.

In short, we hope that by using a expressive specification formalism (York Interactors) to specify an Interactive System as a whole, modeling system and interface behaviour, and the users' needs and capabilities, and by using the analytic power of theorem proving coupled with the expressiveness of temporal logic, we will be able to improve our ability to reason about Interactive Systems in order to correctly predict/verify how they will be integrated in a real work situation.

6 Conclusions

In this paper, we started by introducing a framework for Interactive Systems properties. This framework allows us to look at techniques for the formal verification of Interactive Systems and see how they handle the different aspects that must be considered.

Four approaches were identified. Three use Model Checking (one by Abowd et al. [1], one by Paternó [23], and one by d'Ausbourg et al. [6]), the latter uses automated Theorem Proving (by Bumbulis et al. [3]).

Using our framework we were able to identify the strengths and weaknesses of the different approaches. These could be found either at the level of the specification notations, which affect what type of properties can be expressed, as well as at the level of the verification techniques, which affect what type of properties can be verified.

At the first level we identified a need to express how the user interface relates to the underlying system, users, and to better address the interaction mechanisms of the user interface. At the second we identified the need to combine the expressive capabilities of model checking and theorem proving.

We hope that, by using York Interactors based specifications, and the combined power of theorem proving and temporal logic reasoning, we will be able to achieve these objectives. This is ongoing work.

Acknowledgements

José Creissac Campos is supported by grant PRAXIS XXI/BD/9562/96. We are grateful to Gavin Doherty who made comments on earlier drafts of this paper. We also wish to thank the anonymous reviewers for their comments, and HCM network on Interactionally Rich Systems for financial support under contract ERBCHRXCT930099.

References

1. Gregory D. Abowd, Hung-Ming Wang, and Andrew F. Monk. A formal technique for automated dialogue development. In *Proceedings of the First Symposium of Designing Interactive Systems - DIS'95*, pages 219–226. ACM Press, August 1995.
2. F. Bodart and J. Vanderdonckt, editors. *Design, Specification and Verification of Interactive Systems '96*, Springer Computer Science. Springer-Verlag/Vien, June 1996.

3. Peter Bumbulis, P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena. Validating properties of component-based graphical user interfaces. In Bodart and Vanderdonckt [2], pages 347–365.
4. J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic In Computer Science*, pages 428–439. IEEE Computer Society Press, June 1990.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
6. Bruno d'Ausbourg, Guy Durrieu, and Pierre Roche. Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour. In Bodart and Vanderdonckt [2], pages 105–122.
7. Alan Dix and Gregory Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):324–346, November 1996.
8. Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction*. Prentice-Hall, 1993.
9. David Duke, Michael Harrison, Jöelle Coutaz, Laurence Nigay, Daniel Salber, Giorgio Faconti, Menica Mezzanotte, Fabio Patern´o, and David Duce. The Amodeus system reference model. Technical Report System Modelling/D9, Amodeus Project, June 1995.
10. David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.
11. D.J. Duke, P.J. Barnard, J. May, and D.A. Duce. Systematic development of the human interface. In *Asia Pacific Software Engineering Conference*, pages 313–321. IEEE Computer Society Press, December 1995.
12. Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In *Computer Aided Verification, Proceedings of the Fourth International Workshop, CAV'92*, number 663 in Lecture Notes in Computer Science, pages ???–???, 1992.
13. Bob Fields, Peter Wright, and Michael Harrison. A method for user interface development in safety-critical applications. Human-Computer Interaction Group, University of York (unpublished), 1996.
14. Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):243–275, July 1986.
15. John V. Guttag, James J. Horning, et al. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
16. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
17. Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, Inc., 1995.
18. Jos´e A. Mañas et al. *Lite User Manual*. LOTOSPHERE consortium, March 1992. Ref. Lo/WP2/N0034/V08.
19. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
20. Andrew F. Monk and Martin B. Curry. Discount dialogue modelling with Action Simulator. In G. Cockton, S. W. Draper, and G. R. S. Weir, editors, *People and Computer IX - Proceedings of HCI'94*, pages 327–338. Cambridge University Press, 1994.
21. R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7):761–778, February 1993.
22. Philippe Palanque, Fabio Patern´o, R´emi Bastide, and Menica Mezzanote. Towards an integrated proposal for interactive systems design based on TLIM and ICO. In Bodart and Vanderdonckt [2], pages 162–187.

23. Fabio Patern´o. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995.
24. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In *Computer-Aided Verification, CAV '95*, number 939 in Lecture Notes in Computer Science, pages 84–97. Springer Verlag, July 1995.
25. Harold Thimbleby. *User Interface Design*. Frontier Series. ACM Press, 1990.
26. Hung-Ming Wang and Gregory D. Abowd. A tabular interface for automated verification of event-based dialogs. Technical Report CMU-CS-94-189, Department of Computer Science, Carnegie Mellon University, July 1994.