The role of verification in interactive systems design[⋆]

José C. Campos and Michael D. Harrison

Human-Computer Interaction Group
Department of Computer Science, University of York
Heslington, York YO10 5DD, U.K.
e-mail: {Jose.Campos,Michael.Harrison}@cs.york.ac.uk

Abstract. In this paper we argue that using verification in interactive systems development is more than just checking whether the specification of the system has all the required properties; and that changing the focus from a global specification into partial, property oriented, specifications can provide a number of advantages and make verification act as an aid to decision making. We also present a compiler that allows for the verification of interactor specifications to be done in SMV, as well as a simple case study where verification is used to inform a design decision.

1 Introduction

The introduction of automation in safety critical environments raises the question of system correctness. When an automated system is responsible for the control of a sensitive process, even a *small* failure might have unacceptable outcomes. This problem is particularly relevant in reactive systems, as it becomes hard to comprehend and predict all the different interactions the system might have with its environment.

The use of formal mathematical models during development can impact system design at two levels [21]:

– the use of rigorous mathematical concepts and notations can help in the organization and communication of ideas;
– mathematical models can allow us to reason rigorously about properties of the system being designed.

The latter, although sometimes seen just as a side effect of formal specifi cation, is particularly relevant when we think of guaranteeing the correctness of a system, and has long been an object of study [16].

Formal reasoning, however, tends to be a delicate, detailed, and time consuming process. 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 identifi ed:

– algorithmic methods (i.e. model checking – see [5]): these are fully automated and, given a suitable system description and property, are capable of determining if the property is valid in the system;

[⋆] Published In P. Markopoulos and P. Johnson, editors, *Design, Specification and Verification of Interactive Systems '98*, Springer Computer Science, pages 155-170. Eurographics, Springer-Verlag/Wien, 1998.

– deductive methods (i.e. theorem proving): these are semi-automated methods where a more traditional mathematical proof is performed by a tool under user guidance (examples of theorem provers are PVS [6] and the Larch system [13]).

Interactive systems are special cases of reactive systems, and have growing application in safety critical environments (aeroplanes and nuclear power plants being two of the most commonly cited examples). These systems have raised interest in the application of verification techniques to their development. In the last four years a number of approaches to verification have been proposed. For example, Abowd et al. [1] use Action Simulator [19] to represent very abstract, action level, descriptions of interactive systems. These descriptions can then be translated into the SMV model checker for verification. Paternó [20] also uses a model checker, but the specification is more detailed. He uses Lotos interactors, to derive a specification of the system from its task description. Bumbulis et al. [3] and d'Ausbourg et al. [7], on the other hand, use specifications built at a lower level of abstraction: buttons, mouse clicks, etc. While the first authors use the HOL theorem prover, the second use a model checking related technique. For a review of these approaches see [4].

In the remainder of this paper we will identify some of the problems that arise when trying to apply these approaches and propose a means of integrating verification into the development process of interactive systems. In section 2 we will argue that a shift in focus is needed when thinking of the verification of interactive systems, and in section 3 we will present some work that is being done in this context. In section 4 we illustrate the previous discussion with an example. And finally, in section 5 we draw some conclusions, and propose some further work.

2 Changing the focus

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 1). Some authors have proposed templates for interesting

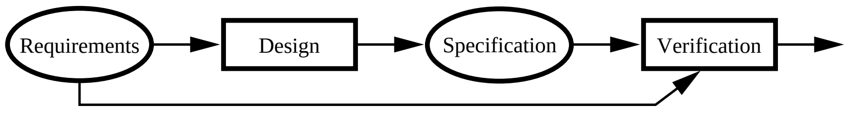

Fig. 1. Verification as a final step in development

properties that can be verified with their approaches (see [1] and [20]). So in Abowd et al., templates deal mainly with what user interface states can be reached. On the other hand, in Paternó, 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 the model checking will allow (what meaning can be attributed to checkable properties), rather than being driven by the requirements needs.

On the role of models All the above cited approaches are based around the notion of a single specification. However, given the complex and semantically rich nature of interactive systems, it becomes impossible to make sure that we have tied down all relevant details of a system in a specification at the outset.

In fact, given the diversity of perspectives we can take on an interactive system, it has been argued, by Fields et al. [12], that no available single specification notation is powerful enough to capture all relevant details of a given system. Instead the authors argue that, if we are to be able to validate a system design against an heterogeneous set of requirements, then a number of specialised models will be needed. Each model taking an appropriate viewpoint on the system, vis-à-vis the requirements being considered.

We can start to identify a trend towards a type of approach to verification in which we are not restricted to a particular verification tool or technique.

On the role of properties By defining a set of property templates for a given class of systems, we 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.

First of all, we do not yet have a 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 that we can have 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 specification being used. 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 specification is done. If we choose to consider only individual interactions between the machine and its user, then the template should not be used: in each interaction, we want the user to be able to perform only one task. However, if we take a broader look at the system, and choose to consider the successive interactions of different users, then the property is relevant as we want the user to 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 specification that is used. We can conclude that we must exercise care when looking for what properties we should prove of specifications.

On what exactly is being proved Seeing the verification step as a final step in the development process, and trying to use *off the shelf* properties, might lead us to end up looking for properties of the specification instead of the system. Although this is not, in itself, negative (we might want 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 tells us those target states are always reachable. How much does it tell us about the system? It is quite different if we need ten actions or one hundred actions to achieve task completion. No information about this is obtained from verifying the property though.

On the applicability of the techniques The number of different perspectives that we might have on interaction means that if we aim to have a global unified specification encompassing all the relevant information, we will get a very complex model. This is specially problematic as we still lack tools comprehensive enough to enable the verification of systems as rich as interactive systems. As has been pointed out in [21], 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, further work is needed in the area.

Even if a powerful enough tool 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 our specification is not reducible to a finite state machine, we simply cannot use model checking on it[1]. If we do use a finite state machine, however, then theorem proving becomes of little use. 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 we can use a theorem prover 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 ask for different styles of reasoning.

All this means that the specification should be written with the type of proof that we intend to perform in mind.

On changing how verification is applied It is now clear that a shift in focus is needed. Instead of focusing verification on the specification, we must focus on the system itself and its properties. This way, specification and validation become part of the same development process, where validation is used to inform design decisions. This 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 specifications (or models – cf. [12]) each taking a particular viewpoint on the system.

This way, we can achieve a number of benefits:

- by focusing on properties, we use verification to validate the choices that are made in relation to what is important of the system, not its specification;

[1] To help obviate this problem, some work has been done on model checking non-finite state machines. For an example of the application of this type of technique to HCI see [18].

- by using a number of models, instead of a single specification, we are able to apply the most appropriate verification technique to each property;
- conversely, by using a number of models, we can develop each model in the most suitable formalism, regarding the tool that will be used;
- using models that focus on properties also means the models will be simpler and easier to manage and verify, so we will be able to verify properties that otherwise would be too difficult to check;
- another possibility raised by this approach is the reuse of properties and their proofs: because we focus on properties, we might be able to reuse the proofs when thinking of related properties of different systems.

As pointed out in [12], 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 towards the system being modelled. We tend to agree with the authors in that these issues are not necessarily disadvantages or even specific to the approach, and that, in the end, these concerns have to do more with the designer than with the tool.

Nevertheless, it is also the case that an appropriate tool should support the designer in dealing with these concerns. Although we do not address the issue in the present paper, it does deserves further consideration.

3 An Interactors to SMV compiler

From what has been said so far, it can be seen that we will need a toolbox of different verification strategies tailored to work with our specifications. We have been experimenting with different techniques and tools, and, in this context, we are developing a compiler to allow the verification of interactor based specifications with SMV.

The notion of Interactor [20,9] is basically that of an object capable of making its state perceivable to its users. York interactors do not prescribe a specific specification notation to describe the interactor's state and behaviour. Instead, they act as a mechanism for structuring the use of standard specification techniques.

In SMV [17] the state of the system is defined by a set of variables (boolean or enumerated). The behaviour is defined as a transition relation on states. This is done, either by a set of logic expressions asserting which transitions of the state are valid, or by a set of firing rules.

In [1] Abowd et al. presented an application of SMV to the verification of interactive systems specifications. Their approach, however, is based on a simple propositional production system based description of the interactive system written in Action Simulator. This means only very simple abstract specifications can be checked, and, in fact, the authors propose the approach to be used at a very high level of abstraction in early interactive system design.

With interactors, however, specifications are built compositionally. The system is broken down into a number of meaningful components, and each component is specified by an interactor. These interactors communicate between themselves and/or with the user as appropriate.

It seems clear that this style of specification is potentially more expressive than a PPS, and will not, in most cases, be adequately translatable into a single PPS. We need then investigate how to express such specifications in SMV.

Specifying one interactor Several different formalisms have been used to specify interactors, including Z [9], modal action logic (MAL) [10], and VDM [14].

The fact that we want to be able to model our interactor specifications in SMV, influences the choice of notation to describe interactors. When defining exactly what kind of interactor to use, we have to decide on:

- how the state of the interactor is defined;
- how the behaviour of the interactor is specified;
- how communication between interactors is modelled.

In order for the interactor specification to be reducible to a finite state machine, SMV deals only with boolean and enumerated type variables. This restriction will be carried forward to our interactor notation. Although this is a major restriction, we feel the approach will still be useful because we are interested in building partial, property oriented, models. And, in fact, this is a restriction common to all approaches using model checking techniques.

As to the modelling of interactor behaviour, of the different approaches to modelling with interactors, the use of MAL seems to be a good candidate for modelling interactor behaviour. MAL augments propositional logic with the notion of action and three operators:

- OBL — $\text{OBL}(ac)$ means action ac must happen next;
- PER — $\text{PER}(ac)$ means action ac can happen next;
- $[\,]$ — $[ac]prop$ means after action ac proposition $prop$ holds.

The operators OBL and PER correspond to some form of quantification over the actions that may be initiated by the user in a given state:

- $\text{OBL}(ac) \equiv \forall_{ac_1} \bullet [ac_1]true \Rightarrow ac = ac_1$
- $\text{PER}(ac) \equiv \exists_{ac_1} \bullet ac = ac_1 \wedge [ac_1]true$

where $[ac_1]true$ means "it is possible to perform action ac_1 in the current state" (i.e. there is, at least, one trace from the current state that starts with action ac_1). Although we do not use these two operators in this paper, the potential for asking questions about obligation and permission are of interest to us.

The formula

$$[]prop$$

indicates that $prop$ holds before any action is taken (i.e. $prop$ holds in the initial state of the system). Formula

$$prop_1 \Rightarrow [ac]prop_2$$

means that when proposition $prop_1$ holds, performing action ac makes proposition $prop_2$ true.

If we interpret propositions $prop_1$ and $prop_2$ above as pre- and post-conditions of action ac, we can use this type of formula to define a production system in a similar way to Action Simulator. The difference is that each interactor defines its own finite state machine, while with Action Simulator we have only one global finite state machine.

We follow loosely Dearden and Harrison's syntax in [8]. As an example consider the following which represents a window defining whether it is mapped on the screen or not:

interactor $window$
attributes
  $mapped : boolean$
visible
  $mapped$
actions
  $map, unmap$
axioms
1. $[]\neg mapped$
2. $\neg mapped \Rightarrow [map]\mathsf{next}(mapped)$
3. $mapped \Rightarrow [unmap]\neg\mathsf{next}(mapped)$

The state of the interactor is defined by a set of attributes. These attributes can then be associated with modalities to specify how they should be rendered in the presentation. In the present case, the only information in the state is whether the interactor is mapped on the screen or not, and this information must be visible in the presentation of the interactor (cf. **visible** clause).

The behaviour of the interactor is defined by a set of axioms on the available actions. These axioms define all possible behaviours of the interactor. In the present case we have two actions (to toggle the state on or off), and three axioms. Axiom 1 states that the initial state of the interactor is unmapped. Axioms 2 and 3 define when the actions are available and their effect on the state. Axiom 2, for instance, states that if the interactor is not mapped then action map can take place and gives rise to a new state where the interactor is mapped. As we assume the axioms to define all possible behaviours, if the interactor is not mapped, only the map actions can occur. Note that $\mathsf{next}(mapped)$ means the value of $mapped$ in the next state.

The algorithm to translate one interactor into the SMV input language is very similar to that presented in [1]. The major difference being that we include in each state, not the next action to be executed, but the last action that was executed (the action that led into the present state instead of the action that leads out of the present state). This avoids the *explosion* of the initial states (simplifying the composition of interactors) but makes the detection of deadlocks more difficult.

The algorithm has been implemented in a prototype compiler. If we run the example above through the compiler we get the following SMV module (where `POSTnil` represents the post-condition of the stuttering step):

```
MODULE window
VAR
mapped: boolean;
action: {nil, map, unmap};
DEFINE
POSTnil := (next(mapped)=mapped);
```

```
INIT
!mapped & action=nil
TRANS
  (mapped & next(action)=unmap & !next(mapped))
| (!mapped & next(action)=map & next(mapped))
| (next(action)=nil & POSTnil)
FAIRNESS
!action=nil
```

The structure of the module is similar to that of the interactor. The state is defined by a set of variables. The variable `action` was introduced to hold the value of the action that led into the present state. Because this variable is an enumerated type, we cannot have parametrised actions. To overcome this limitation we can *explode* parametrised actions into a set of actions, one per parameter value.

`INIT` defines the initial state of the module and was generated from axiom `1` in the interactor. `TRANS` defines the behaviour of the module and is generated from the axioms of the interactor, excepting those defining the initial state. Axioms of type

$$pre \Rightarrow [ac]post$$

are translated into

```
pre & next(action)=ac & next(post)
```

and *ored* together with the stuttering formula:

```
next(action)=nil & POSTnil
```

Note that the intuitive translation would be:

```
(pre & next(action)=ac) -> next(post)
```

However, this would allow `next(post)` to hold even if the pre-condition and action did not. Because we are assuming that axioms specify all possible behaviours, we use the first translation instead.

The fairness condition shown is a default condition and can be overridden in the interactor definition (see below).

Note that the modality (**visible**) has not been included in the module. At this stage the modalities are not directly represented in the SMV code and have to be taken into account informally during the verification process.

Besides the clauses shown in the example, our interactor notation allows for three additional clauses:

- **importing** — allows one interactor to reuse the state and behaviour of another interactor, in SMV this is done by simply repeating the code in the new module;
- **fairness** — this clause allows the definition of a fairness expression to be used by SMV;
- **define** — enable us to give names to expressions as can be done in SMV, the names can then be used instead of the corresponding expressions.

Multiple interactors So far, we have only considered the one interactor situation. Now we will see how to compose interactors and how to model their communication.

Interactors are organised in a hierarchy, where one interactor (the parent) can have other interactors (the children) as part of its state.

Parents can access the state of children using the notation:

$$child.attribute$$

By convention, a parent should only access attributes that have an appropriate modality.

Parents can also *send* actions to their children, either explicitly:

$$child.\mathsf{action} = ac$$

or implicitly, by imposing conditions on the next state of the child. Consider the following excerpt:

```
interactor $main$
attributes
  $mail : window$
...
axioms
  $\mathcal{P}rop \Rightarrow \mathsf{next}(mail.mapped)$
  ...
```

The axiom effectively implies that, if $\mathcal{P}rop$ is true and the window $mail$ is not mapped, then an action to map it must be performed.

In order to translate these hierarchies of interactors into SMV, we use the notion of module. So each interactor will be a module in SMV, and we demand that an interactor named `main` exists.

In SMV, if a module is declared a process, then it progresses independently from other modules in the system. This would allow for the different interactors in the systems to progress independently of each other (modulo their interactions). For this to work, however, the modules must be completely independent. In our case, because parents can influence the behaviour of their children, there is no guarantee that the modules will be independent (in fact its very unlikely that they will) so declaring modules as processes has no effect. That is the reason why explicit stuttering had to be introduced through the use of the `nil` action.

Finally, in order to test properties of the specifi cation a new clause was introduced: **test**. It can only be declared in the main interactor, and it is used to specify a CTL formula whose validity is to be verifi ed by SMV.

4 An example

In this section we will show how the ideas presented above can be applied in practice. We will use as example 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, we will focus on whether or not the user is made aware of new mail. Following the ideas put forward in section 2, we will take a fish-eyed view of the system, and we will build models that concentrate on what is relevant for the issues at hand. Although the example will be kept very simple, we feel that it is still representative at two levels.

– it shows how, by focusing on a specific view of the system, we can build 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[2].

As we consider 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 we are only concerned with visibility issues, we will build our models of windows 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;
– 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.

Were we to be interested in a different aspect of the system, the set of attributes to include in the specification would be different. In the context of our 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
  $map, unmap, hide, show, update, seen$

The complete definition of the `window` interactor can be seen in appendix A.

Two mechanisms for announcing new mail were considered. We will now describe the specification and analysis performed for each of them.

[2] Note that we are not saying we will be able to perform all types of analysis this way. The possibility of using different formalisms and different verification tools, as appropriate, is one of the advantages of building specialised models.

Using a simple mail agent window The first possibility considered was 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 analysis that we want to perform, however, we do not need to know exactly how it is done, so we will use the simple window interactor described above (see also appendix A).

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

– $mail$ — 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 (see appendix B). Once the specification is complete, we can start to explore it. The first test is done just to increase our confidence in the soundness of the specification:

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

it simply verifies if we can have both windows mapped (thus verifying that the finite state machine is not empty - note that initially both windows are unmapped). Next we try to see if after a new message arrives that information is available to the user:

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

The answer to this test is no and we get the following counter example[3]:

```
state 1.1:                    state 1.2:
others.mapped = 0             mail.newinfo = 1
mail.newinfo = 0              mail.action = update
mail.visible = 0
mail.mapped = 0
```

What this shows is that if the e-mail window is not visible when the new message arrives, then the user has no way of knowing about this new message.

This prompts us to think about the next mechanism we will consider: using a pop up window to announce new mail. But first, we do another test: what happens when the e-mail window is visible and a new message arrives? Will the window still be visible?

$$\mathsf{AG}(mail.visible \rightarrow \mathsf{AX}(mail.\mathsf{action} = update \rightarrow mail.visible))$$

The answer here is that it is possible that the user might not be able to notice the new message, as some other window might hide the e-mail window.

Using a pop-up window Having identified a problem with the previous approach, we now try to devise a new design that might solve it. We will consider 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 $update$ event is redefined (at present, redefining the behaviour of an event is still done is a rather clumsy manner, future versions of the compiler are expected to allow for a better syntax):

[3] Note that only attributes of the state that change are shown. Also, we stripped the counter-example of irrelevant attributes.

interactor $popup$
importing
  $window[dummy/update]$
actions
  $update$
axioms
  $\mathsf{!action} = dummy$
  $[update]\mathsf{next}(mapped)\&\mathsf{next}(newinfo)\&\mathsf{next}(visible)$

The main specification now includes three windows:

– $mail$ — 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 the following axiom:

$$mail.\mathsf{action} = update \leftrightarrow alert.\mathsf{action} = update$$

Additionally, if the user resets the e-mail client, the alert window must be reset:

$$mail.\mathsf{action} = seen \rightarrow !alert.newinfo$$

The complete specification is presented in appendix C. Once the specification is complete, we can repeat the same tests as previously. First we check the three windows can be mapped:

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

Once again this serves only to enhance our confidence in the specification itself.

The first real test is whether the pop up window becomes visible when a new message arrives:

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

In this case the answer is yes.

This step accomplished, we can increase our demands upon the system. One thing is the alert window to become visible, another is for the user to see it. To test that the window will always be visible until the user sees it, we test that it can only disappear by direct action of the user (that is, the user has to *see it* before it goes away):

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

Unfortunately in this case the answer is no, and the counter example shows us that some other window might hide the alert window before the user gets a chance to see it.

From this analysis we can conclude that this approach, although better than 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 we can conclude that, if we want to maximize user awareness regarding the arrival of new mail, we must use some kind of permanent window displaying the status of the mail box. This is not a terribly ground breaking or surprising result. In fact, we could easily reach the same conclusion 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 us to reach conclusions about the systems that are not so obvious.

The example, however, does not illustrate all the aspects of the matter. In this case, we are only using model checking. The properties we are interested in regard reachability questions, so model checking is the appropriate tool. Were we, for instance, to consider how the list of messages is kept sorted by date, model checking would probably not be the right tool. In that case we would have to use a theorem prover. That, however, does not have any implications in the analysis performed above, as the specifications used in each case would be independent.

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, we may then combine its results with the results of the present analysis to make a choice.

5 Conclusions

We have argued that using verification in (interactive) systems development is more than just checking whether the specification of the system has all the required properties. Global specifications tend to be too complex for verification, and different types of properties ask for different proof styles/techniques (hence, different specification styles). In this context, we propose 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: we can have greater confidence in that we are checking properties which are relevant to the system (not only of its specification); the specifications to verify become simpler; we can think of reusing proofs on systems with similar requirements.

Furthermore, the properties we want 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. In the paper, we present a simple case study where verification is used to inform a design decision.

In order for this type of approach to be feasible, we will need an ensemble of verification tools tailored to work with our specifications. We present an interactor compiler which is being developed as part of a wider study on the applicability of different techniques to the verification of interactor specifications. The compiler allows for specific type of interactors to be checked by SMV and is used in the example.

Although we think that the use of verification as an aid to development helps in making verification more usable and useful, a number of problems remain open.

The interactor notation accepted by the compiler has a number of restrictions. We need to apply it to *real life* systems in order to access its potential. Although the sheer size of a system might not be a problem (the use of interactors should allow the specifi -cations to scale up well), the restrictions on the variables and on the specifi cation of behaviour limit the expressiveness of the approach. Another point to consider is whether it is possible to include the modality annotations in the SMV generated code. At the moment they have to be taken into account *informally*.

The use of theorem provers has not been addressed in this paper, but suitable representations of interactor specifi cations have to be developed that can be verifi ed with that type of tool.

At the methodological level, the use of multiple models raises the questions of consistency and veracity. In [12] it is argued that consistency can be exploited in benefi t of the development process, and that, regarding veracity, this problem is not exclusive of a multiple models approach to development.

Finally, in the example we looked at a very simple system property. In more realistic applications, what should guide the use of verifi cation? In what areas should we apply it, and what are the roles of notions like tasks, mode of interaction, or modality of interaction? We envisage that some form of task model, or scenario driven design (cf. [11]) might be useful here.

Acknowledgements

We wish to thank the anonymous reviewers for their comments. José Campos is supported by Fundação para a Ciência e a Tecnologia (FCT, Portugal) under grant PRAXIS XXI/BD/9562/96.

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. Jos´e C. Campos and Michael D. Harrison. Formal verification of interactive systems: A review. In Harrison and Torres [15], pages 109–124.
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. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques, April 1995. http://www.csl.sri.com/sri-csl-fm.html.
7. 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.
8. A. M. Dearden and M. D. Harrison. Risk analysis, impact and interaction modelling. In Bodart and Vanderdonckt [2], pages 229–247.

9. David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.
10. 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.
11. Bob Fields, Michael D. Harrison, and Peter Wright. THEA: Human error analysis for requirements definition. Technical Report YCS 249, Department of Computer Science, University of York, Heslington, York, YO1 5DD, England, 1997.
12. Bob Fields, Nick Merriam, and Andy Dearden. DMVIS: Design, modelling and validation of interactive systems. In Harrison and Torres [15], pages 29–44.
13. John V. Guttag, James J. Horning, et al. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
14. M. Harrison, R. Fields, and P. C. Wright. The user context and formal specification in interactive system design (invited paper). In C. R. Roast and J. I. Siddiqi, editors, *BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*, electronic Workshops in Computing. Springer, September 1996. http://www.springer.co.uk/ewic/workshops/FAHCI/.
15. M. D. Harrison and J. C. Torres, editors. *Design, Specification and Verification of Interactive Systems '97*, Springer Computer Science. Springer-Verlag/Vien, June 1997.
16. C. B. Jones. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, Department of Computer Science, University of Manchester, June 1992.
17. K. L. McMillan. *The SMV system*. Carnegie-Mellon University, draft edition, February 1992.
18. M. Mezzanotte and F. Patern´o. Verification of properties of human-computer dialogues with an infinite number of states. In C. R. Roast and J. I. Siddiqi, editors, *BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*, electronic Workshops in Computing. Springer, September 1996. http://www.springer.co.uk/ewic/workshops/FAHCI/.
19. 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.
20. Fabio Patern´o. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995.
21. John Rushby. Model checking and other ways of automating formal methods. Position paper for panel on Model Checking for Concurrent Programs, Software Quality Week, San Francisco, May/June 1995.

A A Window Interactor specification

interactor $window$
attributes
$mapped : boolean$
$visible : boolean$
$newinfo : boolean$
visible
$mapped\ visible\ newinfo$
actions
$map, unmap, hide, show, update, seen$
axioms
$[]!mapped\ \&\ !visible\ \&\ !newinfo$
$!mapped \Rightarrow [map]\mathsf{next}(mapped)\ \&\ \mathsf{next}(visible)\ \&\ \mathsf{next}(newinfo){=}newinfo$
$mapped \Rightarrow [unmap]!\mathsf{next}(mapped)\ \&\ !\mathsf{next}(visible)\ \&\ \mathsf{next}(newinfo){=}newinfo$

$mapped$ & $visible \Rightarrow [hide]$next$(mapped)$ & !next$(visible)$
& next$(newinfo)$=$newinfo$
$mapped$ & !$visible \Rightarrow [show]$next$(mapped)$ & next$(visible)$
& next$(newinfo)$=$newinfo$
$[update]$next$(mapped)$=$mapped$ & next$(newinfo)$
$newinfo \Rightarrow [seen]$next$(mapped)$=$mapped$ & !next$(newinfo)$

B Specification of the simple mechanism

interactor $main$
attributes
$mail : window$
$others : window$
define
$mail\_changevis$:= next($mail$.action) in $\{map, unmap, show\}$
$others\_changevis$:= next($others$.action) in $\{map, unmap, show\}$
axioms
♯ How visibility might change...
!($mail.mapped$ & $others.mapped$)
$\rightarrow$ ((next($mail.visible$)=$mail.visible$ & next($others.visible$)=$others.visible$
) | $mail\_changevis$ | $others\_changevis$)

C Specification of the pop up window

interactor $popup$
importing
$window[dummy/update]$
actions
$update$
axioms
!action=$dummy$
$[update]$next$(mapped)$ & next$(newinfo)$ & next$(visible)$

interactor $main$
attributes
$mail : window$
$others : window$
$alert : popup$
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 $\{map, unmap, show\}$
axioms
$mail$.action=$update \leftrightarrow alert$.action=$update$
$mail$.action=$view \Rightarrow$!$alert.newinfo$
!(($mail.mapped$ & $others.mapped$) | ($mail.mapped$ & $alert.mapped$)
| ($others.mapped$ & $alert.mapped$))

$\rightarrow$(($\mathsf{next}(mail.visible)$=$mail.visible$ & $\mathsf{next}(others.visible)$=$others.visible$
& $\mathsf{next}(alert.visible)$=$alert.visible$
) | $mail\_changevis$ | $others\_changevis$ | $alert\_changevis$)