Using task knowledge to guide interactor specifications analysis

José Creissac Campos

Departamento de Informática, Universidade do Minho, Campus de Gualtar
4710-057 Braga, Portugal.
Jose.Campos@di.uminho.pt

Abstract. This paper looks at how to extend the type of analysis that can be done using interactor based models of interactive systems, the i2smv tool, and SMV. Originally, the type of analysis performed with i2smv/SMV was concerned with the exhaustive exploration of all possible behaviours of a device, with little direct consideration of the tasks it should support. The paper investigates how task related knowledge can be introduced into the approach in order to extend the range of properties that can be analysed.

1 Introduction

The development of software systems has proved to be an iterative process where quality is achieved incrementally. As software evolves so does the cost of introducing changes. It is important that quality analysis be started as early as possible in the development process.

The quality of interactive devices can be measured in terms of their usability. Empirical approaches to the evaluation of systems designs attempt to analyse the designs under real world usage conditions, but are typically expensive. This is so both because the analysis is performed late in the design life-cycle, and because setting up the analysis requires a lot of resources and time. Analytic approaches to the analysis of system's models have been proposed as a means to reason about usability issues from the initial stages of design. These approaches use models to focus the analysis in specific usability issues. In recent years the use of formal (mathematically based) notations and tools in the context of reasoning about usability has been proposed [10,2]. The main motivation has been the possibility of performing thorough and repeatable analysis of usability related issues from as early as possible in the design process.

One such approach is presented in [2]. Models of interactive devices are used to reason about the usability of the envisaged system (device+user). The analysis is performed by attempting to prove that the device's behaviour exhibits certain desired properties. Models are structured using the notion of interactor [6,4] and expressed using modal action logic. The analysis is performed using SMV, a model checker. A tool has been developed to translate the interactor models into the SMV input language. The properties to be proved are written in CTL.

By analysing a model of the device we guarantee full coverage of its behaviour (as modelled), but it becomes harder to analyse how the device reacts to specific user behaviours. Put simply, an unpredictable user is assumed, that can take any action at any time. This paper looks at how to extend the approach so that specific behaviours are more easily analysed. The paper investigates how task related knowledge can be introduced and used to analyse if envisaged tasks will have the desired effect when performed correctly.

It must be stressed that the paper is not proposing formally based approaches as substitutes for more traditional usability analysis techniques. Rather as one more tool that can be used during development.

Section 2 will discuss the motivation to consider tasks during verification. Section 3 introduces the interactor language and tool, and an example of analysis. Section 4 introduces tasks, and section 5 shows how they can be used during the analysis. Finally section 6 presents some discussion and conclusions.

2 Usability analysis with formal methods

Reasoning about usability means that considerations about the user(s) must be brought to bear on the analysis. In the approach of [2] this is reflected in the choice of what properties to prove, and in the interpretation of the results. Typically, properties are expressed over the states that are reached by the device. During verification all possible behaviours of the device are considered (i.e. device behaviour is left unconstrained).

The advantage of this approach is that it forces the analysis to consider previously unforeseen behaviours that might lead to undesired states. In practice this can lead to a number of false negatives which must be investigated and dealt with in turn during the analysis. Some of these behaviors will be discarded because they appear due to the encoding of the model (for example, the use of abstraction might mean that the model exhibits more behaviours than the actual device), or because, despite being possible behaviours of the device, they are not considered plausible/relevant from a human-factors stand point (for example, if we are investigating how calls are made using some mobile phone, we might wish to disregard behaviours where the user explicitly cancels the dialling process).

Filtering out unwanted behaviours can be done either by altering the model, or the property to be proved. Safely altering the model might be difficult when dealing with complex systems. Changing the property to filter out unwanted behaviour can be complex and lead to difficult to read properties. CTL (Computational Tree Logic — see section 3.2) is well suited for expressing properties about which states a system might or not reach, it is not as suited for expressing properties over the paths leading to those states. For example, it is easy to express that a state where *prop* holds can be reached (EF *prop*), it is not as simple to write properties about the possible behaviours that lead to that state.

Some authors have proposed that user or task models should be coupled with the device model for analysis [14,5,11,7]. This increases the complexity of the final model but gives a greater focus to the analysis. Only those behaviours that are consistent with the model of the user/task are considered during the analysis. One main difference between the two approaches is that while user models try to define how user behaviour is *generated* (by representing the mental structures and activities of the user), task models directly define the behaviour that users are supposed to exhibit. User models can be specially useful when analysing how a novice user will behave when faced with the device. Unfortunately they are very hard to develop. Task models are most useful in situations where prescribed procedures are defined and should be followed.

One drawback of using a user or task model is that the scope of the analysis is narrowed. If a user model is used, it will typically cover *rational* user behaviour, and, possibly, typical classes of user error. This can leave out some unforeseen anomalous behaviour that, though unlikely, might have a negative impact on the device's usability. If a task model is used then only those behaviours that are defined as correct according to the task description will be considered during the analysis. Once again anomalous behaviours with negative impact in usability might go unnoticed. Errors may be introduced in the task model to alleviate this, but full coverage cannot be guaranteed.

By not explicitly considering a user or task model, approaches such as the one presented in [2] aim at full coverage of the device's behaviour. This allows for the detection of unexpected traces of behaviour that might jeopardise the usability of the system. The drawback in this case is that this type of approach makes it harder to analyse specific user behaviours and how they are supported by the device. There is a strong focus on the device at the modelling level.

Clearly the ideal solution would be to have a mixed approach, allowing for both exhaustive analysis of device's behaviour, and analysis of how well the device supports specific tasks.

3 Interactor analysis

This section briefly describes the language used to model interactive systems, and the tool that enables analysis of the models using SMV. For further details readers are referred to [2]. The section ends with a small example.

3.1 The interactor language

Interactors, as developed in [4], are a structuring mechanism for interactive systems' models. They help in applying general purpose specification languages to interactive systems modelling. Interactors do not prescribe a specific specification language, rather a structuring of the models that is adequate to model an interactive system.

Using interactors, models are structured around the notion of an object that is capable of rendering (part of) its state into some presentation medium. Hence, each interactor has a state (defined as a set of attributes), a number of events it can engage in (defined as a set of actions), and a rendering relation specifying which attributes/actions are perceivable/can be used by users. In the particular notation used in this paper, the behaviour of the interactor is defined using Modal Action Logic (MAL) [12].

There are four basic types of axioms to define behaviour:

- modal axioms are used to define the effect of actions in the state of the interactor — for example, axiom [newcall] ringer'=on ∧ menu'=answercall ∧ keep(dialflag, endcallflag, state) asserts that after action newcall the value of attribute ringer becomes on, the value of attribute menu becomes answercall, and attributes dialflag, endcallflag and state do not change. Priming is used to reference the value of an attribute in the state after the action has happened (non-primed attributes are calculated in the state prior to the action happening). The keep(attrib) notation is used to specify that the value of attrib does not change, and is equivalent to attrib'=attrib.
- permission axioms are used to define when actions are allowed to happen — for example, axiom per(newcall) → ¬ringer asserts that action newcall can only happen if ringer is false.
- obligation axioms are used to define that, under certain conditions, a given action must happen — for example, axiom state=sending → obl(sent) asserts that when state is sending then action sent must happen at some point in the future (it does not have to be immediately).
- initialisation axioms are used to define the initial state of the interactor — for example, axiom [] ringer=off ∧ menu=makecall ∧ dialflag=nil ∧ endcallflag=nil ∧ state=idle asserts the values of the different attributes in the initial state.

3.2 The i2smv tool

A tool has been developed that enables the automatic verification of the models described above using the SMV tool [8]. SMV is a model checker which uses CTL [3] as the logic to express properties. Models are defined as finite state machines, and CTL used to express properties over the behaviour of the models.

The properties that can be written/verified deal mainly with which states can or cannot be reached. Typical properties include:

- X is an invariant — AG(X) (X holds in all states of all behaviours);
- X is inevitable — AF(X) (for all possible behaviours, X will eventually hold);
- X is possible — EF(X) (for at least one behaviour X will eventually hold).

Different combinations of the operators can be used to express more complex properties. For example, AG(X → AF(Y)) expresses the property that it is an invariant that whenever X holds it is inevitable that Y will hold.

The i2smv [2] tool translates interactor models into the SMV input language. In this way it becomes possible to verify properties of the interactors' behaviour expressed in CTL. Because SMV does not have the notion of action, a special attribute "action" is used at the CTL level to refer to the action that has happened.

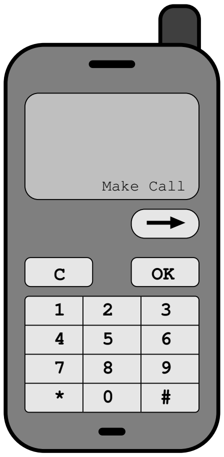

Fig. 1. Mobile phone

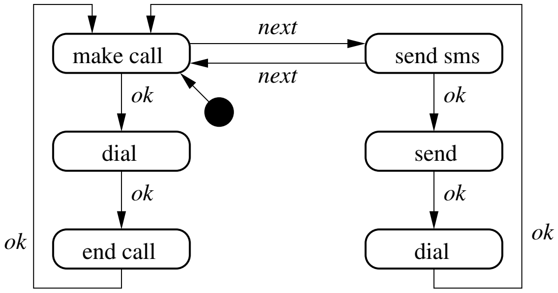

Fig. 2. Menu navigation

3.3 An example

As an example consider a simple mobile phone with capabilities for receiving and making phone calls, and receiving and sending SMS messages (see figure 1). Operation of the phone is based on a menu. At each moment the menu presents one single option to the user. Besides the usual number keys, the user interface of the phone has an OK key (to select the current menu option), a Cancel key (to reset the menu), and a key for menu navigation (advance to the next option).

The structure of menu navigation during operation of the device is presented in figure 2. In normal operation the menu option displayed will toggle between "make call" and "send sms". If, for example, the user selects "make call" (by pressing OK while that option is presented on screen), the menu changes to "dial". The user is expected to enter the number to call and press OK to dial it. Once the number is dialled, the menu option changes to "end call", which the user can select by pressing OK once more. We are not considering the possibility of the call not being established. This is a reasonable simplification, mainly since the use of voice mail is nowadays widespread.

There are three exceptions to the "normal" behaviour represented in figure 2. When a call is received the menu option displayed changes to "answer call". This option will remain active until the user accepts (by pressing OK) or rejects (by pressing Cancel) the call, or until the caller gives up. When a new SMS arrives the menu option displayed changes to "read sms". The option will remain active until the user accepts or rejects reading the sms message. Finally, the Cancel button can be used at any time to reset the dialogue.

An interactor model of a device which supports this behaviour was developed (see appendix A for the full model). The state of the device is modelled by two main attributes: ringer and state. Attribute ringer models the ringing behaviour of the phone. It starts ringing whenever a new call or sms arrives (see axioms 15 and 18). It stops ringing when the user answers the call/reads the message. If the user does none of the above it eventually stops ringing (see axioms 21 to 23). Attribute state models the over all state of the device. This is done at an high level of abstraction, nevertheless detailed enough to allow the analysis. The possible states are: *idle* (nothing is happening); *dialling* (a number is being dialled); *calling* (a call is in progress); *reading* (a sms is being displayed); *writing* (a sms is being written); and *sending* (a sms is being sent).

Two additional attributes are used to distinguish between dialling a number for a phone call or for a sms (attribute dialflag), and making or receiving a call (attribute endcallflag). This is relevant since the behaviour of the device is different in each case.

With this model it is possible to test some features of the design. For that it is first necessary to define an interactor named main:

```
interactor main
includes
  mobile via phone
test
  ...
```

To test if it is possible to make a call the following test would be made:

EF(phone.state=calling)

The answer is that the property holds. We now know that it is possible to have the phone in the calling state. We know nothing, however, of how that state can be reached. One possibility to investigate this is to verify the property:

¬EF(phone.state=calling)

This property is obviously false and the trace ok→ok is presented as a possible behaviour that leads to a call being in progress. This is expected behaviour, but there might be others. Unfortunately there is no direct way of finding out all possible behaviours that falsify the property.

Another test that can be performed is whether the phone always rings when a new call is received. The property to check would be:

$$\text{AG(phone.action=newcall} \rightarrow \text{phone.ringer=on)}$$

Again the answer is that the property is true. Note that questions such as whether the user will be aware of the phone ringing fall outside the scope of this type of approach. Nevertheless, it is useful to know that the phone works properly.

It is one thing to have the phone ringing whenever there is a new call, it is another for the user to be able to answer the call. To test whether a call can always be answered the property is:

$$\text{AG(phone.menu=answercall} \rightarrow \text{AF(phone.state=calling))}$$

This property is false and the counter example presented shows that the user can cancel the call instead of answering it. This is correct behaviour but we want to consider situations where the user wants to answer the call. To filter out the above behaviour we rework the property to be:

$$\begin{aligned}&\text{AG(phone.menu=answercall} \rightarrow \\ &\qquad\qquad \text{AF(phone.state=calling} \vee \text{phone.action} \in \{\text{cancel}\}))\end{aligned}$$

A new counter-example is produced. This time showing that the caller can give up on the call before the user answers. This is also correct behaviour but one that we do not want to consider further at this stage. Filtering out this behaviour, a further counter-example shows that a sms message can arrive before the user answers a call. In fact in the current design, the arrival of an sms message cancels any incoming or ongoing call. This is most likely unwanted behaviour and the design should be changed to address this issue.

The above analysis shows how this type of approach can be useful in detecting potential usability problems. The analysis implies going through a process where unwanted behaviour is filtered out by encoding constraints into the property. This process is useful in building an understanding of the contextual conditions under which a given property of the system holds. The process, however, can become time consuming and difficult for complex systems.

For complex devices more elaborate constraints might be necessary than those possible to encode directly in CTL formulae. It is also not easy to verify if a given user behaviour has the desired effect on the device. The properties that can be written are mostly about the state the device is in, not the behaviour the user/device system is having. Thinking about how a goal is actually achieved is done indirectly. What the constraints mostly represent is what should not happen for the goal to be reached.

If we want to reason about whether a certain specific behaviour will achieve the goal, some representation of the intended behaviour must be incorporated into the model. To address this, we will introduce tasks into the approach.

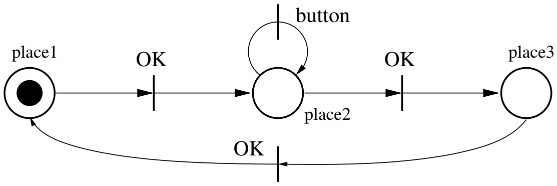

Fig. 3. C/E-system for making a call

4 Modelling tasks

The analysis proposed in the previous section enables the exploration of all possible behaviours of a device model, thus supporting the identification of unforeseen usability problems. It does not, however, enable the analysis of how well the device supports prescribed usage behaviour. This section addresses this issue by introducing task related knowledge into the interactor modelling framework. First an encoding for tasks is chosen, then the expressions of this encoding using interactors is explained.

4.1 Expressing task related knowledge

Task analysis is a well studied field. It is not the purpose of this paper to put forward a new proposal for a task description language. In fact, we will abstract away from concrete task notations and consider simply what is fundamental in the notion of task.

Whatever the specific language used, tasks will describe the valid sequences of events leading to some desired goal. To keep the approach independent from a specific task modelling language we will model these sequences of events directly using Condition/Event-systems (C/E-systems - a subclass of Petri nets where places can have at most one token). We are not proposing Petri nets as a task modelling language, rather as an intermediate notation to which other languages can be translated (cf. [9,7]).

As an example consider the task of making a phone call in the mobile phone described above. Starting from the initial state this can be achieved by pressing OK, dialling the number, and pressing OK again. To finish the call, OK must be pressed once more. This behaviour (OK→button* →OK→OK) can be represented with the C/E-system in figure 3. Actions are modelled by transitions between places, and places are created so that the valid sequences of actions are modelled by sequences of transitions' firing.

Note that we are modelling user behaviour only. It is possible to include user and system behaviour in the model also. To decide on the best approach to take it is necessary to consider the goal of the analysis. If the objective is to analyse the task structure itself, then both user and device behaviour must be included in the task model. This would allow reasoning about whether the proposed task

```
interactor making_a_call
attributes
  place1, place2, place3: boolean
actions
  OK button
axioms
  per(OK) → ((place1 ∧ ¬place2) ∨ (place2 ∧ ¬place3) ∨ (place3 ∧ ¬place1))
  (place1 ∧ ¬place2) → [OK] ¬place1' ∧ place2' ∧ keep(place3)
  (place2 ∧ ¬place3) → [OK] ¬place2' ∧ place3' ∧ keep(place1)
  (place3 ∧ ¬place1) → [OK] ¬place3' ∧ place1' ∧ keep(place2)
  per(button) → place2
  [button] keep(place1,place2,place3)
```

Fig. 4. Interactor for making a call

accomplishes the desired goal. If the objective is to analyse the device, then by including user actions only we can test how the device reacts to the user actions and whether the user will achieve the goal by performing the task. This latter approach is less normative since it leaves the device's behaviour free. This allows for the identification of situations where, despite the user following the prescribed procedures, the device deviates from intended behaviour.

4.2 Mapping tasks to interactors

Expressing tasks using interactors is now reduced to expressing the C/E-systems using the MAL logic. To translate a C/E-system into an interactor, each place is modelled by a boolean state variable representing whether the place is marked or not, and each transition is modelled by two axioms. A permission axiom stating when the transition is allowed to fire, and a modal axiom stating the effect of the transition on the marking of the net.

For example, transition OK from place1 to place2 above generates the permission axiom: $\mathrm{per(OK)} \rightarrow \mathrm{place1} \wedge \neg \mathrm{place2}$ (that is, OK can fire when place 1 is marked and place2 is unmarked), and the modal axiom: $(\mathrm{place1} \wedge \neg \mathrm{place2}) \rightarrow [\mathrm{ok}]\ \neg \mathrm{place1'} \wedge \mathrm{place2'} \wedge \mathrm{keep(place3)}$ which reads, under the conditions defined by the permission axiom (this is necessary because in this case other transitions are labelled OK also) the effect of firing OK is to leave place1 unmarked, place2 marked, and place3 unchanged.

When different transitions are associated with the same event, as is the case above, the permission axioms are joined by disjunction. The interactor expressing the task introduced above is presented in figure 4. Note that the modal axiom for button does not need a guard since there is only one transition for that event.

4.3 Linking task to device

The next step, in order to check the device model against the task model, is to link both models together. For that a new version of interactor main is developed:

```
interactor main
includes
  mobile via device
  making_a_call via task
axioms
  task.action≠nil → task.action=device.action
  task.action=nil → device.action ∉ {button,ok,cancel,Next}
```

This interactor links the task model to the model of the device that should support it. The link between the two models is established at the level of actions. The first axiom establishes that when an action occurs at the task level, then the same action must occur at the device level. The axiom can use the expression task.action=device.action since the same actions names are used in both models. The second axiom restricts the action that can happen at the device level independently from the task level. In this case they cannot be actions performed by the user. Together the two axioms restrict the behaviour of the system so that user actions can only happen according to the task description, and device actions can happen freely according to the device's semantics.

5 Revisiting the example

This section looks at how we can use task knowledge as encoded above to explore the design of the mobile phone.

5.1 Using task knowledge

Using the model above it is possible to perform more in-depth analysis of how the system reacts to intended user behaviour. A first property that could be checked is whether following the prescribed task procedure it is possible to make a call. The property to check is

$$\mathrm{EF}(\mathrm{phone.state}=\mathrm{calling} \land \mathrm{task.place3})$$

Note that it is important to check whether the task has reached place3. If only the device state was considered,we could get false positives. Consider, for example, the event sequence: device.newcall→task.ok; it is a trace that fits into the task description, and leaves the device in the calling state, without the task procedure being completely carried out (only the first step of the task would be performed). This trace can be obtained by checking: !EF(phone.state=calling).

The property above is verified by the model checker. The next step is to check whether performing the task always results in a call being established. To this end we check the property:

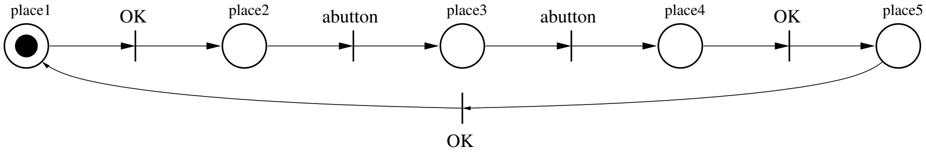

Fig. 5. C/E-system for making a call (revised version)

AF(phone.state=calling ∧ task.place3)

This property fails with the following trace: task.ok→device.newcall→device. giveup→*task continues (but phone is no longer in the expected state)...* What this show is that if the phone rings while a number is being input, then that number cannot be dialled. In fact, in the phone as modelled, the arrival of a phone call or of a sms message, preempts any task that is being carried out. For example, checking the property:

AG((phone.state=calling ∧ task.place3) →
AF(task.action=ok ∧ phone.state=idle))

(it is always possible to end an ongoing call by pressing ok) shows that if during a call a sms message arrives then the call is lost (or at least the ability to end it!).

To solve this problem a redesign of the device is needed. For now we can filter out such behaviour by writing:

AF((phone.state=calling ∧ task.place3) ∨ device.action ∈ {newcall,newsms})

Checking the property above reveals a problem with the task model. There is no upper bound on the number of times the user can press a button. This means that, according to the task description, the user can keep pressing buttons indefinitely, thus not accomplishing the goal of making a call. To solve this we could make the task description more concrete. Without loss of generality we would consider a task where a 2-digit number is dialed. The C/E-system for this version of the task is presented in figure 5.

5.2 Redesigning the device

The problem with the arrival of new calls and sms messages has been circumvented above by adapting the property being checked. This has enabled the analysis to progress, but the problem with the device remains. To solve it, it is necessary to redesign the phone. It is not the purpose of this paper to present a full specification for a mobile phone. However, for the sake of argument, a simple redesign will be presented here.

The problem with the current design is the interference of device triggered behaviour (the arrival of a new call or sms message) with user triggered behaviour

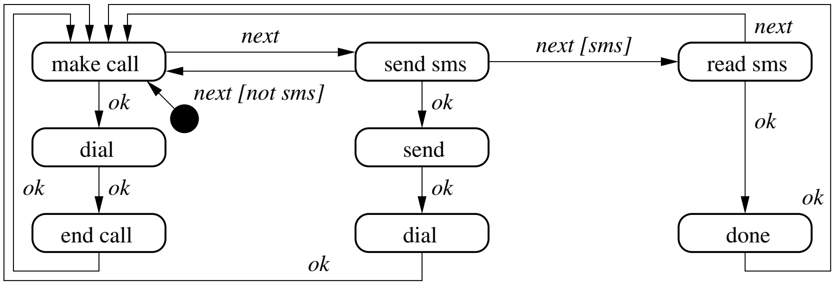

Fig. 6. Menu navigation (redesign)

(making a call or sending a sms message). The solution is to *isolate* the two types of behaviour. Assuming we attach more relevance to the user triggered behaviour we will leave it as is, and change the behaviour of the device when new calls/sms messages arrive, so that its response is not intrusive regarding the user current activity.

Regarding sms messages, the basic idea is that the phone should give an indication that new messages have arrived, but otherwise leave the state of the phone unchanged. This can be achieved by incorporating a indication of new messages on the phone screen (an icon that is turned on whenever there are unread messages). Regarding incoming calls the simplistic solution (probably too simplistic) is to reject calls when the phone is in use. With this changes the menu now must offer the possibility of accessing unread messages (see figure 6).

These design options cause some changes in the model. A new boolean attribute (sms) is introduced to represent whether the sms icon is being displayed or not. A five message queue is also introduced to hold the messages that are received. If the queue is full no new messages can be received. The queue is represented by an attribute (queue) with values in the range 0 to 5.

The receiving sms and call sections of the behaviour definition must be updated accordingly. For incoming sms messages the axioms become:

per(newsms) → queue<5
[newsms] sms' ∧ queue'=queue+1
menu=readsms → [ok] queue'=queue-1 ∧ menu'=done ∧ state'=reading
∧ keep(ringer,sms,dialflag, endcallflag)
menu=done → [ok] sms'=(queue'>0) ∧ state'=idle ∧ menu'=makecall
∧ keep(ringer,queue,dialflag, endcallflag)

for incoming calls the permission axiom becomes:

per(newcall) → state=idle ∧ ¬ringer

All other axioms in the interactor are updated to take account of the new attributes. With the resulting model it becomes possible to prove that following the prescribed task procedure always results in a call being established.

The same type of reasoning can be performed for other tasks. In the case of sending an sms message, for example, the results show that a task similar to the one above enables the user to send sms messages, except when a new call arrives before the user selects the "send sms" option.

6 Discussion and Conclusions

To guarantee the quality of designs at the lowest cost it is necessary to start analysing quality from as early as possible. Interactive systems quality can be measured in terms of their usability. A number of analytic methods have been proposed for the early analysis of interactive systems' designs, including methods based on the use of formal methods and reasoning techniques.

In [2] one such proposal is put forward that focus on the analysis of device behaviour in order to identify potentially dangerous and unexpected behaviours that might jeopardise usability. This paper presents an approach to the integration of task knowledge into the analysis of interactive systems devices found in [2]. This broadens the scope of analysis made possible with the original approach.

Other authors have proposed similar approaches, integrating knowledge of device usage into the analysis of the device's design. Fields [7] also uses separate device and task models, and uses Murϕ, a state exploration tool, to explore the behaviour resulting from the combination of both models. One advantage of using SMV is that it enables greater expressiveness in the properties that can be explored. Rushby [11] uses SMV to analyse joint models of device and user. In this case however, there is no clear separation between the two models. While Rushby refers to models of the user, it is not completely clear if they are user models of models of user activity (i.e. task models). For a review of the application of automated reasoning techniques in usability analysis see [1].

A different style of approach is proposed by Thimbleby [13]. Mathematica is used to perform analysis of a mobile phone's menu structure design. In this case, however, the analysis is not concerned with the effect of actions on the device state, or with the outcome of performing some specific task. The device is assumed to behave correctly and its interface is analysed regarding complexity. Complexity is measured in terms of the number of user actions needed to reach desired functions in the menu structure. The analysis is based on probabilistic distributions of usage of device functions and interface actions. This is a style of approach which is complementary to the one presented here.

In this paper, tasks have been used to restrict the device's behaviour to a suitable subset of all possible behaviours. In the original, device only, approach constraints on behaviour were encoded into the properties being verified since not all possible behaviours might be considered relevant. At first it might look that we are simply looking at different strategies to achieve the same goal. There is, however, a relevant difference. Task knowledge must be known from the outset, it represents the prescribed behaviour for system usage. The goal of the analysis is to see if the device adequately support such prescribed behaviour. In the device oriented approach little knowledge is assumed. The constraints emerge from the attempts made at proving properties. These constraints encode knowledge that is elicited by the analysis. It becomes clear that the two types of approach complement each other in performing early analysis of designs. This should help in reducing the number of problems found later in development.

Regarding future work, one aspect worth pursuing is the exploration of task patterns. Thus far, the equivalence between task level actions and device level actions has been done at the level of action names. It would be useful to have generic task patterns which could be instantiated as needed. For that to be possible the interactor language needs to be extended to allow actions as parameters.

Acknowledgments

The author thanks Michael D. Harrison and the reviewers for their comments.

References

1. José C. Campos. *Automated Deduction and Usability Reasoning.* D.Phil. thesis, Department of Computer Science, University of York, 1999.
2. José C. Campos and Michael D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3-4):275–310, August 2001.
3. 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.
4. David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.
5. D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Syndetic modelling. *Human-Computer Interaction*, 13(4):337–393, 1998.
6. G. Faconti and F. Paternò. An approach to the formal specification of the components of an interaction. In C. Vandoni and D. Duce, editors, *Eurographics '90*, pages 481–494. North-Holland, 1990.
7. Robert E. Fields. *Analysis of erroneous actions in the design of critical systems.* D.Phil. thesis, Department of Computer Science, University of York, 2001.
8. Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
9. Ph. Palanque, R. Bastide, and V. Senges. Task model - system model: towards an unifying formalism. In *Proceedings of HCI International conference*, pages 489–494, Yokoohama, Japan, July 1995. Elsevier.
10. Fabio D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems.* PhD thesis, Department of Computer Science, University of York, 1995. Available as Technical Report YCST 96/03.
11. John Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, February 2002.
12. Mark Ryan, José Fiadeiro, and Tom Maibaum. Sharing actions and attributes in modal action logic. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 569–593. Springer-Verlag, 1991.

13. Harold Thimbleby. Analysis and simulation of user interfaces. In S. McDonald, Y. Waern, and G. Cockton, editors, *Proc. BCS Human Computer Interaction*, volume XIV, pages 221–237, 2000.
14. Richard M. Young, T. R. G. Green, and Tony Simon. Programmable user models for predictive evaluation of interface designs. In K. Bice and C. Lewis, editors, *CHI'89 Proceedings*, pages 15–19. ACM Press, NY, May 1989.

A Mobile phone model

interactor mobile

attributes

$\boxed{\text{vis}}$ ringer: {on, off}

$\boxed{\text{vis}}$ menu: {makecall, sendsms, dial, send, endcall, readsms, answercall, done}

state: {idle, dialling, calling, reading, writing, sending}

dialflag: {nil, call, sms}

endcallflag: {nil, make, answer}

actions

$\boxed{\text{vis}}$ button ok cancel Next

newcall newsms giveup sent

axioms

# menu navigation

(1) menu=makecall $\rightarrow$ [Next] menu'=sendsms $\wedge$ keep(ringer,dialflag,endcallflag,state)

(2) menu=sendsms $\rightarrow$ [Next] menu'=makecall $\wedge$ keep(ringer,dialflag,endcallflag,state)

(3) menu $\notin$ {makecall,sendsms} $\rightarrow$ [Next] keep(ringer,menu,dialflag,endcallflag,state)

# making a call / sending a SMS

(4) menu=makecall $\rightarrow$ [ok] menu'=dial $\wedge$ dialflag'=call $\wedge$ state'=dialling $\wedge$ keep(ringer,endcallflag)

(5) (menu=dial $\wedge$ dialflag=call) $\rightarrow$ [ok] menu'=endcall $\wedge$ dialflag'=nil $\wedge$ state'=calling $\wedge$ endcallflag'=make $\wedge$ keep(ringer)

(6) (menu=endcall $\wedge$ endcallflag=make) $\rightarrow$ [ok] menu'=makecall $\wedge$ endcallflag'=nil $\wedge$ state'=idle $\wedge$ keep(ringer,dialflag)

(7) [button] keep(menu,ringer,dialflag,endcallflag,state)

(8) menu=sendsms $\rightarrow$ [ok] menu'=send $\wedge$ state'=writing $\wedge$ keep(ringer,dialflag,endcallflag)

(9) menu=send $\rightarrow$ [ok] menu'=dial $\wedge$ dialflag'=sms $\wedge$ state'=dialling $\wedge$ keep(ringer,endcallflag)

(10)(menu=dial $\wedge$ dialflag=sms) $\rightarrow$ [ok] menu'=makecall $\wedge$ dialflag'=nil $\wedge$ state'=sending $\wedge$ keep(ringer,endcallflag)

(11)per(sent) $\rightarrow$ state=sending

(12)state=sending $\rightarrow$ obl(sent)

(13)[sent] state'=idle $\rightarrow$ keep(ringer,menu,dialflag,endcallflag)

# receiving a call / SMS

(14)per(newcall) $\rightarrow$ ringer$\neq$on

(15)[newcall] ringer'=on $\rightarrow$ menu'=answercall $\wedge$ keep(dialflag,endcallflag,state)

(16)menu=answercall $\rightarrow$ [ok] menu'=endcall $\wedge$ endcallflag'=answer $\wedge$ ringer'=off $\wedge$ state'=calling $\wedge$ keep(dialflag)

(17)(menu=endcall $\wedge$ endcallflag=answer) $\rightarrow$ [ok] menu' $\in$ {makecall, readsms} $\wedge$ endcallflag'=nil$\wedge$ state'=idle $\wedge$ keep(ringer,dialflag)

(18)[newsms] ringer'=on $\wedge$ menu'=readsms $\wedge$ keep(dialflag,endcallflag,state)

```
(19)menu=readsms → [ok] menu'=done ∧ ringer'=off ∧ state'=reading
                                          ∧ keep(dialflag,endcallflag)
(20)menu=done → [ok] menu' ∈ {makecall, readsms} ∧ state'=idle
                                          ∧ keep(ringer, dialflag,endcallflag)
(21)per(giveup) → ringer=on
(22)menu=answercall → [giveup] ringer'=off ∧ menu' ∈ {makecall, readsms}
                                          ∧ state'=idle ∧ keep(dialflag,endcallflag)
(23)menu=readsms → [giveup] ringer'=off ∧ keep(menu,dialflag,endcallflag,state)
# cancel
(24)[cancel] ringer'=off ∧ menu'=makecall ∧ dialflag'=nil ∧ endcallflag'=nil ∧ state'=idle
# state at power up
(25)[] ringer=off ∧ menu=makecall ∧ dialflag=nil ∧ endcallflag=nil ∧ state=idle
```