ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

C.R. Roast and J.I. Siddiqi, Sheffield Hallam University, UK (Eds)

BCS-FACS Workshop on Formal Aspects of the Human Computer Interface

Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, Sheffield Hallam University, 10-12 September 1996

Verification of Properties of Human-Computer Dialogues with an Infinite Number of States

M. Mezzanotte and F. Paternó

Published in Collaboration with the
British Computer Society

Verification of Properties of Human-Computer Dialogues with an Infinite Number of States

Menica Mezzanotte & Fabio Paterno'

CNUCE - C.N.R.
Pisa, Italy

Abstract

One of the advantages of using formal methods in human-computer interfaces is the possibility to reason about user interface properties. Model checking techniques provide a useful support for this purpose. This paper discusses the problem of verifying the properties of user interfaces when the dialogue specification has an infinite number of states. Recent techniques allow designers to get some results even in this particular case and we discuss when such results are useful for verifying the user interface properties.

1 Introduction

There are many reasons for using formal notations: they allow designers to obtain unambiguous descriptions of the desired functionalities, to clarify aspects of their design and to reason about the specifications performed.

A wide set of formal notations are available which often differ in terms of their expressive power, and the distinct aspects that they describe best.

It is important for the development of a complex user interface to have automatic tools which allow designers to apply some transformations on the specifications and to verify specific requirements and properties, for example usability properties.

The possibility of rigorous reasoning is one of the main advantages of using formal notations. This can be carried out either by model checking or by theorem provers (such as HOL or PVS). In the former case the specification represents the model against which properties can be checked, the latter allows one to move from the theory provided by the specification to an underlying model in which the properties can be checked. Theorem provers are more difficult to use.

Formal verification has been successfully used in hardware design where it is important to check that some properties are satisfied before implementing the specification into hardware. The human-computer interaction field is more challenging for verification methods and tools, since the specification of human-computer dialogues may be more complex than the hardware specifications.

Verification techniques are rapidly improving. They are able to address specifications with millions of states [4]. Some specifications can reach an infinite number of states because of the dynamic nature of the related applications. This is the case of some human-computer dialogues. However, some new verification techniques are being developed to address these cases and in this paper we discuss their application to user interfaces.

We mainly consider environments for process-based notations such as LOTOS [10] and CCS [12].

2 Related Works

The first results of the application of tool-supported, formal verification to check properties of basic interaction objects and user interfaces were described in [14]. Since then other works have been carried out: Palanque and Bastide [15] use Petri Nets to specify Interactive Systems and to reason about them. Petri Nets is a powerful notation supporting parallelism and time, however the specifications performed with it may have a low modularity. Some research work [18] is being developed to investigate whether it is possible to integrate this approach with our TLIM (Tasks, LOTOS, Interactors Modelling) method [16]. Abowd and Wang [1] have used similar tools to those used by us to verify properties expressed in CTL, which has the same power as ACTL but describes state modifications rather than actions performance, and, in order to simplify the use of a formal notation they use the user interface of Monk's Action Simulator to express the properties which have to be verified.

We have applied our approach which consists in developing a LOTOS specification of the Interactive System considered and then verifying ACTL [6] properties by model checking in various case studies. For example, we

considered the user interface of an air traffic controller [17]. In that case, we verified the possibility of undesirable events which, especially for that type of application, can have very serious consequences.

Recent work [2] has been developed to apply the HOL theorem prover in order to verify mechanically specific requirements imposed on the user interface. Duke and Harrison used modal action logic for reasoning about user interactions [9] but in that case so far there is no automatic support for reasoning. Johnson [11] used temporal logic as a means to specify and prototype user interfaces rather than for reasoning about a more detailed specification.

3 Expressive power of Specification Notations

We can consider Interactive Systems as a subclass of Reactive Systems. In fact, they share the main feature of reactive entities: the continuous reaction to events externally generated. The main distinctive feature is that one component is a human being whose input devices are eyes and ears, and output devices are hands and mouth. User behaviour is driven by cognitive systems which have specific mechanisms and requirements which differ from those of software architectures.

Whereas in computer science there is a strong tradition in comparing notations in terms of their expressive power, this is not part of the typical user interface designer background.

This is demonstrated, for example, by the use in many industrial sites of state transition diagrams to represent user interface dialogues. However, this type of notation can be used to describe very simple, not detailed examples. In fact, user interfaces, especially multimodal user interfaces, can perform many actions in parallel and activate and deactivate interaction techniques. These features are difficult to represent with finite transition diagrams. On the other hand, process-based notations, such as CCS, CSP and LOTOS, are suitable for describing these features, as they are mathematical models developed to capture in an immediate way the features of concurrent systems. In fact, if we want to describe the possibility of interleaving between two tasks, for example editing and printing a file, each one composed by the sequence of three actions, this is immediately expressed in LOTOS ($\parallel$ is the LOTOS operator indicating the possibility of interleaving among actions of the two considered elements):

```
Task1[open_file, modify_file, save_file]  $\parallel$  Task2[ select_file, select_parameters, print_file]
```

where

```
process Task1[open_file, modify_file, save_file]:noexit:=
open_file; modify_file; save_file; Task1[open_file, modify_file, save_file]
endproc
```

```
process Task2[select_file, select_parameters , print_file]:noexit:=
select_file; select_parameters; print_file; Task2[select_file, select_parameters, print_file]
endproc
endspec
```

The corresponding labelled transition system is very complex and is represented in Figure 1. It was obtained by automatically transforming the previous LOTOS expression. The starting state is indicated by a double circle.

Figure 1: The automaton associated with the two interleaving tasks.

It is important to be aware of the expressive power of a notation to understand both whether it is able to describe the desired behaviour and whether it is possible to reason about the specification. In fact, in the second case, if the specification corresponds to a model with an infinite number of states then properties are more difficult to verify.

To avoid this problem it is important to be able to understand whether specific subsets of a given notation have different expressive powers too, so that we can find out if and when they provide a specification corresponding to a finite state system.

For this purpose Fantechi and Gnesi [8] compared the power of different subsets of LOTOS operators. They showed that it is possible to identify a set of LOTOS basic operators that has the expressive power of regular languages, that is, a set of operators which generates a language whose expressions can be described by a finite state automaton. One possible set is composed of the basic LOTOS expression obtained with the operators: choice, alternative choice between two possible behaviours; “;” which is placed between two sequential actions; stop, which indicates that the process can no longer be involved in interactions; and process instantiation, which provides the definition of a process behaviour.

In [8] some other cases of the use of LOTOS operators which produce expressions corresponding to a finite state automaton are indicated.

4 Examples of Human Computer Dialogues which require different expressive powers

We consider two simple examples of user interactions: one which can be described by a finite state machine and the other which corresponds to a non-finite automaton.

Suppose we want to model a simple interaction with a window containing text and the scrollbar associated with it. We have the situation graphically described in Figure 2.

Figure 2: An interaction with scrollbar and text (corresponding architecture and presentation).

We describe the objects illustrated in Figure 2, using LOTOS, as follows:

```

process Interaction [.....]:noexit:=
(Mouse[move_mouse, ..., cursor] |[move_in_scrollbar]| Scrollbar[move_in_scrollbar, ..., scrollbar_mod])
|[move_in_text, thumb_move]| Text[move_in_text, ..., up_text_pres]

where
process Mouse [move_mouse, move_in_scrollbar, move_in_text, cursor]: noexit:=
move_mouse; (move_in_scrollbar; cursor; Mouse [move_mouse, ..., cursor]
    [] move_in_text; cursor; Mouse [move_mouse, ..., cursor])
endproc

process Scrollbar [move_in_scrollbar, thumb_move, scrollbar_mod]:noexit:=
move_in_scrollbar; thumb_move; scrollbar_mod; Scrollbar [move_in_scrollbar, ..., scrollbar_mod]
endproc

process Text[move_in_text, save_text, thumb_move, previous_text, up_text_pres]:noexit:=
move_in_text; up_text_pres; Text[move_in_text, ..., up_text_pres]
[] thumb_move; up_text_pres; Text[move_in_text, ..., up_text_pres]
[] previous_text; up_text_pres; Text[move_in_text, ..., up_text_pres]
endproc
endspec
    
```

This LOTOS expression is regular because it uses a set of operators which have been proved to generate regular expressions. Thus the behaviour of this expression can be verified because it can be transformed into a finite state automaton.

The second example considers the case of a text editing application: an icon is active, and when it is selected a window available to include the file to edit is activated. The initial icon has two buttons included: one to activate help on-line (which we will not consider in our specification), the other to quit the application.

Figure 3: The text editing example.

In Figure 3 both the interactor-based description and layout of the example are provided. Here the possible actions are: select the Edit icon to open a text window (*select_ed* action), select the Quit button to close the application (*select_quit* action), and edit a text in the window which has been activated (*int_text* action). In the figure on the left dashed arrows indicate interactor enabling or disabling.

The behaviour of the interactors illustrated in Figure 3 is described, using the LOTOS notation, in the following expression. Note that when the Edit process terminates (*exit* action) it activates itself, by the enabling operator ($\gg$), in interleaving composition ($\parallel$ operator) with a new interactor that manages a new text window.

```

Application[ ... ] [> Quit [select_quit, feed_quit]
where
process Application [ ... ] :noexit:=
Edit [select_ed, feed_ed] >> Application [ ... ] ||| Text[int_text, ... up_pres_text]
endproc
process Edit [select_ed, feed_ed] :noexit:=
select_ed; feed_ed; exit
endproc
process Text[int_text, text, up_pres_text]:noexit:=
int_text; text; up_pres_text; Text[int_text, text, up_pres_text]
endproc
    
```

```

process Quit [select_quit, feed_quit] :noexit
select_quit; feed_quit; exit
endproc
endspec

```

This LOTOS expression cannot be described using a regular expression. This is because the process Application is defined in such a way that it is called recursively, in interleaving composition ($\parallel$ operator) with another process (the Text process). This means that for each recursion a new instance of the Text process can be generated. Thus we are not able to build an automaton associated with this behaviour and, consequently, we cannot verify its properties with the usual tools.

This type of behaviour is common in many dynamic user interfaces: when we have to describe objects which, when interacting with the user, can activate new objects without disactivating the previously created objects.

5 Verification of User Interface Properties

The idea of using model checking techniques to reason about user interface properties was introduced by Paterno' in [14]. The notation used is Action-based Temporal Logics, a branching-time temporal logic which allows designers to reason about the actions a system can perform.

In our TLIM method we start from the task specification of the application considered and then we derive the LOTOS specification of the Interactive System software inheriting the relationships of the task model. The description of the software architecture is mainly obtained by associating a LOTOS process with each interactor. Some control processes may be added in order to control further the dynamic behaviour of the resulting specification.

If we want to reason better about user interaction we can describe the possible behaviour by associating a LOTOS expression for each cognitive subsystem and then compose them [13].

One of the main advantages of LOTOS specifications is that we can reason about their properties by applying automatic tools for model checking. This means that the LOTOS specification is automatically translated into a corresponding labelled transition system which represents the model against which user interface properties expressed in Action-based temporal logics are automatically verified.

The sort of properties that we deal with are:

- task-related properties, because we can identify the actions in the interactor-based specification which are associated with the performance of a task;
- user-related properties, because we can reason about how the actions of the user interface interact with the actions of the human cognitive system;
- user interface properties which allow us to identify the relationships between user actions and system actions such as the possibility to reach some modification of the presentation or to have continuous feedback.

In [16] there is a detailed discussion of a large set of properties. It is clear that although these are general properties they can still be tailored to the requirements of the particular system being specified.

6 Classification of Properties

As we mentioned before, most verification environments are based on the hypothesis that a system can be modelled as a Labelled Transition System (LTS). That is, they give no modality to deal with a non finite-state LTS. In the Jack environment [3] when a specification has to be verified, first a test is made to check whether the expression is finite and whether the model can be built for the verification.

A new method [7] has recently been proposed that allows properties expressed in ACTL and on CCS terms to be checked. In this paper we discuss an approach to apply it to the verification of user interfaces properties. For some kinds of properties, this method gives a semidecisional procedure, which means that in some cases the verification may be performed, but not in others.

Temporal properties are usually subdivided into subclasses: *liveness* properties (something good eventually happens) and *safety* properties (nothing bad can happen). Properties can be divided into two classes: *universal* (for all computations, A operator) and *existential* (for one computation, E operator).

6.1 Classification of properties expressed in ACTL

If we use the ACTL notation to express these properties we can identify some operators which are more suitable for indicating the related concepts. We can say a property is a *liveness property* if it is expressed in the following manner: AFf , EFf , $AFAGf$, $EFAGf$, $AFEGf$, $EFEGf$, $AGAFf$, $EGAFf$, $AGEFf$, $EGEFf$ with f positive formula. This mainly means that the F (eventually) operator should be included in the property.

In the case of *safety properties*, they are expressed by using ACTL in the following way: AGf or EGf and f is a finite positive property. This means that the G operator (indicating all the states) should be part of them.

Note that these illustrations of properties do not cover all the liveness and safety properties that can be expressed using ACTL. In fact, all properties containing the negation are not considered. The reason is that the model checking environment which we will consider can only give useful answers with positive properties (properties which do not contain negations).

Finally, we can introduce *finite properties*. They are properties expressed in the format $X f$. These properties indicate aspects which should be verified in order to get to the next state (X operator).

6.2 Properties which can be verified over infinite states automaton

The approach presented in [7] provides a method to define the validity of a property on a non-finite system. This method gives a decisional procedure for some types of properties (finite properties), a semidecisional procedure for other properties (safety and liveness properties), and no answer for the remaining properties (such as negative properties).

The verification of a property is obtained by proving it on the elements of an approximation chain of finite labelled transition systems which are branching complete simulation equivalent among them and with the starting specification.

In this method the liveness properties are more easily preserved. The safety properties are more difficult to verify. In fact, this type of formula cannot be verified on all states of the specification (as in this case they are infinite). More precisely in [7] it is demonstrated that if a liveness property is verified on a finite approximation model than it is verified on the original infinite model as well. In the case of weak safety properties it is indicated that if they are not verified on a finite approximation model then they are not verified on the original infinite model.

7 A discussion on user interface properties that can be verified

We want to analyze what user interface properties can be inserted in the liveness or safety category whose syntax has been described previously and which user interface properties cannot be classified following that syntax.

7.1 User Interface Liveness Properties

We want to consider the HCI properties that we can insert in the liveness category formulae. The following four properties can be considered as examples of liveness properties in the user interface field.

Reachability: this property allows us to verify that a user interaction can generate an effect on a specific part of the user interface. This means that, given a user action, we wonder whether it is possible to reach a given effect.

The next ACTL formula means that for all the possible futures (A operator) and for all the possible states (G operator), if user action x is performed then there exists at least one temporal evolution until the *Reachable_effect* action has been performed ($\{ \text{Reachable_effect} \}$ true operator).

$$AG([user\ actionx]EF\langle Reachable_effect \rangle true)$$

Possibility of performing a task at any state: this property states that starting from all the states there is one temporal evolution during which, at some time, the action associated with the task performance occurs.

$$AGEF\langle task_performance \rangle true$$

Visibility: this property means that each user action is associated with a modification of the presentation of the user interface to give feedback on the user input. More precisely, in ACTL, we specify that whenever we have a given user action (*user_action_x*) we want to be sure that there is at least one temporal evolution where the event associated with the specific associated user interface modification (*User interface appearancex* action) will occur. This property can be considered as a specific subclass of reachability properties. It can be expressed as:

$$AG([user\ actionx]EF\ <User\ interface\ appearancex>true)$$

Recoverable Error: an error can be defined as an action which is not needed to perform the current user task. Errors can be divided into minimal, recoverable and unrecoverable. If a minimal error occurs, immediately afterwards the user can perform one action which is useful for task performance. After a recoverable error, several actions are needed to return to the previous state. An error is unrecoverable if it does not allow the user to perform a given task in the current session.

A recoverable error can be defined as an action *er* which verifies the following property. This means that whenever the *er* action occurs then it is possible at some time to perform a useful action for the desired task, and when this useful action occurs then the task can eventually be accomplished:

$$AG[er]\ E[true\ \{true\}\ U\ \{useful_task_action\}\ EF\ <task_performance\ >true]$$

These properties are liveness properties, so we can evaluate them by building some approximations of the infinite automaton. If the liveness properties are evaluated as true on a approximation of the LTS, then we can affirm that the properties are true on the infinite automaton too. On the other hand, if the properties are false on all the approximations, we cannot affirm anything about the falsity of these on the automaton. In fact, this could simply mean that we have not yet found the approximation on which the properties are true.

7.2 Safety User Interface properties

We now want to consider HCI properties that can be inserted into the class of safety formulae.

Here we can have a version of the visibility property which expresses the related concept in a stronger way: if we want the feedback of a user action to occur immediately after it. This is expressed in ACTL:

$$AG([user\ actionx]EX\ \{User\ interface\ appearancex\}true)$$

Minimal error: immediately after an erroneous action, the user can perform one action which is useful for task performance. This is expressed by the ACTL formula:

$$AG([recoverable_error]EX\ \{useful_task_action\}true)$$

The truth of safety properties is more difficult to prove than liveness properties on an infinite automaton approximation with the method which we have considered.

7.3 User Interface Properties which cannot be verified

Unfortunately, there are some HCI properties on which approximations of the infinite automaton cannot give any answers. This is the case of the unrecoverable errors and of all formulae that contain negative formulae.

By unrecoverable errors we mean that once the error has been performed, for all the possible temporal evolutions, it is no longer possible to perform the desired task, thus it is not true that once the unrecoverable error has been performed then there exists one possible evolution where eventually the task is performed:

$$AG[unrecoverable_error]\sim(EFEX\ \{task_performance\}true)$$

This formula contains a negation, so it cannot be evaluated using the method considered. This problem affects the evaluation on some CARE properties too.

The CARE (Complementarity, Assignment, Redundancy, Equivalence) properties [5] have been recognised as a simple but useful framework to evaluate the usability of Multi-Modal Interactive Systems. We can express them in

ACTL, assuming that $m1$ and $m2$ are two actions associated with the use of two different modalities (for example, voice input and graphical input). For example, we can consider Complementarity and Assignment.

Complementarity: the possibility to achieve a desired effect using two modalities (which are both indispensable) in an independent order. It is expressed by:

$$\begin{aligned} &AG[m1] EF\langle m2 \rangle EF\langle effect \rangle true \ \& \ AG[m1] \sim(E[true\{\sim m2\} U \{effect\} true]) \\ &\quad \& \\ &AG[m2] EF\langle m1 \rangle EF\langle effect \rangle true \ \& \ AG[m2] \sim(E[true\{\sim m1\} U \{effect\} true]) \end{aligned}$$

which means that whenever one modality is used then eventually it is possible to use the other modality and then to reach the desired effect, but it is not possible to reach it without performing the other modality as well. The first part of the property is valid if the user interacts first with $m1$ and then with $m2$. The second part is valid in the inverse case.

Assignment : the possibility to reach a desired effect using only one modality is expressed by:

$$AG[m1] EF\langle effect \rangle true \ \& \ \sim(EF([m2] E[true\{\sim m1\} U \{effect\} true])$$

which means once $m1$ has been performed then it is possible to reach the desired effect and there is no possibility to perform $m2$ and not $m1$ to reach it.

For these properties, the approximations do not say anything because they contain formulae with negations.

8 Conclusions

The importance of being able to reason about user interface properties is becoming increasingly recognised. Model checking is an interesting approach to support this kind of evaluation. When industrial size cases are considered it is important to understand the limitations of the notations used in order to drive their application and to recognise when results can be obtained. In previous works we considered the application of model checking techniques when a finite model of the Interactive System considered can be built. Here we have discussed how to use a new research approach for verifying properties of Interactive Systems corresponding to infinite state specifications in order to give more precise indications about when model checking can be used in Interactive System design.

Future work will be dedicated to create an environment more suitable for user interface designers which allows them to use model checking techniques even if they are not expert in formal methods.

References

- [1] Abowd G., Wang H., Monk A. A formal technique for automated dialogue development. Proceedings Designing Interactive Systems'95, August'95, ACM Press.
- [2] Bumbulis P., Alencar P., Cowan D., Lucena C. Combining Formal Techniques and Prototyping in User Interface Construction and Verification. Proceedings of Design, Specification, Verification of Interactive Systems'95, Springer Verlag, pp.17-192.
- [3] Bouali A., Gnesi S., Larosa S. The integration Project for the JACK Environment. Bulletin of the EACTS, N.54, October 1994, pp.207-223.
- [4] Burch J., Clarke E., Long D., McMillan K., Dill D., Wang L. Symbolic Model Checking: 1020 states and beyond. Information and Computation, 98(2), pp.142-170, June 1992.
- [5] Coutaz J., Nigay L., Salber D., Blandford A., May J., Young R. Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. Proceedings INTERACT'95, Norway, June'95.

- [6] DeNicola, R., Fantechi, A., Gnesi, S. and Ristori, G. An Action Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. *Computer Networks & ISDN Systems* 1993, 25, 761-778.
- [7] De Francesco N., Fantechi A., Gnesi S., Inverardi P. Model Checking of non-finite state processes by Finite Approximations. *Lecture Notes in Computer Science* N.1019, pp.195-217, Springer Verlag.
- [8] Fantechi A., Gnesi S. The Expressive Power of LOTOS Behaviour Expressions. *IEI Internal Report*, B4-43, October'92
- [9] Harrison M., Duke D. The Specification of User Requirements in Interactive Systems. to appear in *Handbook of Human-Computer Interaction*
- [10] ISO (1988) Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on temporal Ordering of Observational Behaviour. *ISO/IS 8807*, ISO Central Secretariat.
- [11] Johnson C., Harrison M. Using Temporal Logic to Support the Specification and Prototyping of Interactive Control Systems. *International Journal of Man-Machine Studies*, 37, pp.357-385, 1992.
- [12] Milner R.. *Communication and Concurrency*. Prentice Hall 1989.
- [13] Mezzanotte M., Paterno' F. Use of Task, User, and Formal Models to Support Development of Multimedia Interactive Systems. *Proceedings of IFIP Conference on Domain Knowledge for Interactive Systems Design*", Geneve, May 1996, Chapman & Hall.
- [14] Paterno' F. Definition of Properties of User Interfaces. *Proceedings of SEKE '93 Conference*, June'93, S.Francisco, pp.314-318.
- [15] Palanque P., Bastide R. Verification of an Interactive Software by Analysis of its Formal Specification. *Proceedings INTERACT'95*, Lillehammer, June'95.
- [16] Paterno' F., Mezzanotte M. A systematic approach to transform task specifications into architectural specifications to verify user interface properties. *CNUCE Internal Report*, September 95.
- [17] Paterno' F., Mezzanotte M. Formal Analysis of User and System Interactions in the CERD Case Study. *Proceedings EHCT'95 Ifip Conference*, Chapman&Hall Publisher, Wyoming, August 1995, pp.213-226.
- [18] Palanque P., Paterno' F., Bastide R., Mezzanotte M. Towards an Integrated Proposal for Interactive Systems design based on TLIM and MICO. *Proceedings DSV-IS'96*, Springer Verlag, Namur, June'96.