Formal reasoning about dialogue properties with automatic support

Fabio Paterno*
CNUCE-C.N.R., Via S.Maria 36-56126, Pisa, Italy

Abstract

One of the advantages of using formal methods in the design of human–computer interfaces is the possibility to reason about user interface properties. Model checking techniques provide a useful support to this end. This paper discusses the possibilities of verifying the properties of user interfaces and related problems, such as when the dialogue specification has an infinite number of states. We provide an example of a set of general user interfaces properties, and we show how these properties can be tailored for specific cases and thus be used as a framework to evaluate the design of the interactive application considered. © 1997 Elsevier Science B.V.

Keywords: User Interface properties; Formal methods for HCI; Dialogue model checking

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 specification 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 an interactive application supporting flexible dialogues with the user 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. In the former case the specification represents the model against which properties can be

* Corresponding author.
checked, the latter allows designers 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.

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. Their main distinctive feature is that one component is a human. User behaviour is driven by cognitive systems which have specific mechanisms and requirements which differ from those of software architectures. Its specification cannot be prescriptive. However, the knowledge of the possible tasks can give useful information to manage the description of the possible user behaviour.

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 [1]. 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 too. We mainly consider environments for process-based notations such as LOTOS [2] and CCS [3].

The main problems in applying model checking techniques to the design of user interfaces are: the identification of relevant user interface properties to check and the development of a model of the User Interface System which is meaningful and, at the same time, avoids the introduction of many low level details which would increase the complexity of the model without adding important information for the design of the user interface.

Our approach to user interface design starts by considering the task specification which can be used as a framework for modelling the specification of the system. The task specification is hierarchically structured. We use it in such a way that abstract tasks correspond to abstract interactors which inherit the temporal relations of the corresponding tasks. An interactor [4] is a model for objects which interact with users. This approach is useful for obtaining a structured design of the system by using a systematic, recursive method where abstract interactors are transformed into one or more concrete interactors. Finally, all the interactors and their compositions are specified in LOTOS (a description of the main features of this notation is in ref. [5]), which was developed for another application area, distributed systems, but its features make it suitable for specifying any kind of system where temporal ordering among actions is an important feature. In the end, we obtain a large LOTOS specification which completely describes the behaviour of the system apart from some minor details. Our idea is to use formal verification for an early usability evaluation on the formal specification of the system. We use Action-Based Temporal Logic (ACTL) [6] formulae to verify properties of a system interacting with the user.

In this paper we discuss how and when model checking techniques can be used to support the user interface design process. Next we introduce the specific notations and
tools which we apply for this purpose. Then we describe a case study, a set of general user interface properties and we show how to tailor these properties on the case study. Finally, we introduce the problem of verification of properties of specifications describing behaviours with an infinite number of states and we discuss the possibilities offered by recent tools in these cases.

2. Related work

The first results of the application of tool-supported, formal verification to check properties of basic interaction objects and user interfaces are described in ref. [7]. Since then other works have been carried out: Palanque and Bastide [8] use Petri Nets to specify Interactive Systems and to reason about them. Petri Nets is a powerful notation supporting parallelism and time, however for large specifications it can be difficult to interpret and to modify. Some research work [9] is being developed to investigate whether it is possible to integrate this approach with our TLIM (Tasks, LOTOS, Interactors Modelling) method. Abowd et al. [10] have used tools similar to those used by us to verify properties expressed in CTL, which has the same power as the temporal logic that we use 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 properties by model checking in various case studies. For example, we considered the user interface of an air traffic controller [11]. In that case, we verified the possibility of undesirable events which, especially for that type of application, can have very serious consequences.

Recent work [12] 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 [13], but in that case so far there is no automatic support for reasoning. Johnson and Harrison [14] used temporal logic as a means to specify and prototype user interfaces rather than for reasoning about a more detailed specification developed by a different notation.

Other work in this area [15] proposed to derive a formal model of an interactive system from its UIL description in order to verify and test its behaviour.

3. Our approach for user interfaces design with model checking support

Our method aims to provide a concrete step towards the introduction of formal methods (methods which use notations with precise semantics) in the development process of better quality user interfaces. In fact, the first generation of studies to apply formal methods to Interactive Systems (see for example refs [16,17]) used very abstract models with no automatic support. In that case the use of formal notations was mainly dedicated to clarifying concepts and to giving precise definitions of relevant properties. Although much has been done on using formal methods to describe precisely user interface-related
concepts and gain insight from disciplined description, the reasoning power of formal
techniques still has to be fully demonstrated.

We believe that now it is time for a second generation of work on the application of
formal methods to Interactive Systems. The main goal is to provide an engineering
approach which can address medium-size systems and is useful even in industry as
happens in other application areas.

Note that we are not claiming that formal methods are always useful for everything. We
propose using them in various phases of the development process, combined with informal
techniques, and to analyse some specific aspects. Moreover, the application of formal
methods is not limited to the choice of a formal notation. The method is crucial: how
the model underlying the notation chosen is used to describe the main features of the
application domain considered, when the method requires to specify formally the func-
tionality considered, and how the method supports the possibility to reason about the
specification.

The basic idea is that after an informal task analysis, designers should perform a task
specification which forces them to clarify many aspects related to possible tasks and their
relationships. The task specification is obtained by first structuring the tasks in a hier-
archical way so that abstract tasks are described in terms of more refined tasks. Next,
temporal relationships among tasks are described using ConcurTaskTrees [18] which is a
graphical notation extending the precise semantics of LOTOS operators. Once the task
specification is obtained it provides the information which is useful to derive an interactor-
based architectural specification. This is obtained in a top-down way by applying specific
rules which have been defined [19]. The modelling work can be stopped at various
abstraction levels depending on the purposes of the designer. The architectural specifi-
cation indicates what interactors are part of the system and how they are composed in order
to support the communication of both data and control events. We can transform the
architectural description into a LOTOS specification where interactors are modelled as
LOTOS processes; at the end of this phase we may need to add control processes to avoid

Fig. 1. Verification in the design and development process.
undesired sequences of actions. The specification of the Interactive System considered is automatically transformed from Full LOTOS to Basic LOTOS and next to a corresponding finite labelled transition system. In this case we use automatic general purpose tools for LOTOS. Properties of the user interface are expressed as temporal logic formulae by the designer and model-checked against the model describing the Interactive System software derived in previous steps. The verification is performed by a general purpose automatic tool for formal verification. Once the formal specification has proved to satisfy the relevant properties for the possible dialogues it is easy to derive a software prototype from the architectural specification because this specification identifies the basic software components and how they have to communicate. The prototype can be used for empirical testing which can give useful information for improving both task and architectural model.

As is shown in Fig. 1, verification can be applied to both the task specification and the architectural specification. In both cases it can give useful feedback for the design of the model and of the interactions that it describes. In this paper we focus on the verification of properties of the formal specification at the architectural level in such a way to provide the possibility to consider task-related aspects at this level too.

4. The application of formal verification to evaluate user interface properties

We now show how techniques for formal verification can be applied to demonstrate automatically user interface properties. Since we follow a task-driven approach to modelling the software architecture, we can identify the actions in the architectural specification associated with the performance of a task. This allows us to express and verify task-related properties which answer questions such as “are these the right user actions to perform the desired task?”.

It is possible to apply the formal verification techniques on an architectural specification without using task models, but in that case we would miss the link between actions at the architectural level and tasks and thus we could reason only about low level properties.

4.1. ACTL: motivations and functionalities

Once we have obtained a specification of the considered system we can verify its properties, using ACTL. This notation has successfully been applied to other application areas such as hardware verification. ACTL is a kind of branching time temporal logic which allows specifiers to reason about the actions which a system can perform. This is very important in Interactive Systems, which are mainly characterised by the actions performed both by the user and the system where designers are interested in evaluating properties such as:

- Are these the right actions to perform this task?
- If users perform these interactions will the desired result be presented on the user interface?
- Is it true that whenever users perform this action then they will obtain this result?

An action-based temporal logic thus seems more suitable for evaluating Interactive Systems with respect to notations such as CTL [20], where properties are expressed in terms of state modifications, or the Temporal Logic of Action [21] which is a linear time
temporal logic and therefore cannot easily express possibility properties, which may be important elements for evaluating systems that should interact with users. By “possibility” we mean properties like “at some time the user can either perform task A or task B”.

ACTL is thus a branching time temporal logic which means that the temporal evolution of the system can be represented by a tree-like structure (see Fig. 2) where each branch represents one possible temporal evolution. Each transition is associated with a specific action that the system can perform to get to the next state.

The sort of properties that we are interested in dealing 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 [22];
- 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.

![Fig. 2. An example of tree-like description of the temporal evolution of a system.](image)

![Fig. 3. The verification environment.](image)
User interface properties can be verified by using general-purpose automatic tools for formal verification (Fig. 3). The logic checker which we apply needs the specification to be described in terms of labelled transition systems (systems where the actions are associated with state transitions). We thus transform our specification of the Interactive System from Full LOTOS into basic LOTOS. Full LOTOS is mainly the fusion of two notations: one is a process algebra which describes a system as a set of processes and their possible actions, the other is ACT ONE which allows us to specify algebraic data types. These are used in Full LOTOS to describe the state of each process and its possible modifications along with the types of information which can be transmitted on each communication gate. Transforming from Full LOTOS to basic LOTOS entails removing information on the data types. LITE [23] is a tool which supports two possible transformations from Full LOTOS into Basic LOTOS. In one case for each data type transmitted over one gate (gates support communication among processes in LOTOS), a new gate is associated in the basic LOTOS specification. In the other case there is a one-to-one association of gates performed by the transformation. Since in our case the data types transmitted on a gate are always the same type, we can apply the second type of transformation.

This type of transformation from Full LOTOS to basic LOTOS implies following a specific style of specification. If we want to prevent the resulting basic LOTOS specification from describing a different behaviour we need to:

- use carefully Boolean guards to constrain the behaviour by using synchronizations among processes because Boolean guards are automatically removed by this transformation;
- if some values have to be communicated by a specific gate, in order to discriminate them in the verification there are two possibilities: either associate specific data types with each meaningful subset of values (thus the transformation from Full LOTOS to basic LOTOS will generate different gates for each data type) or associate different gates for communications of different values (obviously this can be done when only a small set of values is communicated).

Once we have obtained the basic LOTOS specification we used automatic tools [24] to build the corresponding finite state machine (F.S.M.). Another advantage of the tool used is that if the specification fails to verify the given property then it is possible to have a counterexample: an indication of one trace of actions associated with the specification which does not satisfy the property. This is very useful to give indications to designers about where the considered specification fails in verifying the given property.

5. User interface properties verification in the MATIS case study

One important aspect in the application of formal methods to the design of interactive applications is the identification of relevant properties to verify for the evaluation of user interface design. We consider action-oriented properties because we think that they are the most interesting in the evaluation of interactive applications. In this section we show how
to express important properties in the ACTL notation. They represent useful templates that can be filled in for any given system with the specific actions of the system considered and we show examples for the case study considered. In Appendix A there is a short introduction to ACTL for readers who are not familiar with it.

The proposed approach can be applied to both the development of a new system and the analysis of existing systems. In this case we consider the second option but the approach is equally applicable to the first one. The only difference is that with the existing systems the perceivable objects are already identified, while with new systems they can, at least partially, be identified by designers after the dialogue verification phase.

MATIS [25] is a multimodal system which allows users to provide requests to a flights database by using graphics and voice devices. In its user interface (Fig. 4) for each request there is a form which shows the values provided by the user. There is a Request tool which allows users to visualise the available values for each type of information (city, airline company, light number, etc.). The result of a query is visualised in a window. Associated with the voice device is an icon and a recognition window which visualises what the natural language parser has interpreted from the vocal and keyboard input.

In order to submit a request at least departure and arrival towns have to be indicated.
5.1. Tasks-to-interactors association in MATIS

We considered a substantial subset of the possible tasks. When building a request various subtasks are possible (in our case study we consider only the three main subtasks): specifying slots of the request (SpecR), selecting a request of interest, clearing the request (Clr), and submitting a request (SubmitR).

Partial values can be provided either by filling in one slot of a form (SpecRFormFilling), or by a natural language recogniser (SpecRN), or by providing deictic requests (SpecRN & Mouse). The submission of the request is performed by sending the request (SendR) and showing the corresponding result (ShowResultR). If we consider a more refined description of the tasks we can see that filling a slot in the request form is possible either by using the keyboard (SpecRKeySlot) or the mouse (SpecRMouseSlot). Requests by natural voice recogniser can be generated either by voice (SpecRSpeech) or by keyboard (SpecRKeyNL). These natural language requests entail first producing non-deictic sentences and then receiving feedback from the natural language parser. Deictic requests can be made either by combining voice (ProduceDeicticSentence(voc)) and mouse input (SelectValue), or by combining keyboard (ProduceDeicticSentence(key)) and mouse input (SelectValue).

It is not the purpose of this paper to show how we obtain from a task specification an interactor-based architectural specification maintaining the same temporal and semantic constraints. The interested reader can refer to ref. [19].

![Diagram of the architecture corresponding to the abstract tasks considered.](image)
In Fig. 5 we have a description of the resulting architectures at a certain abstraction level with boxes which represent interactors, arrows which describe the communication among them and the names of actions that the system can perform. Both external (associated with user-input and user interface presentation modifications) and internal (associated with communication among interactors and between interactors and functional core) behaviours are described.

More specifically, we have one interactor (SpecformFilling) which allows the user to specify directly in a form the parameters for a request to the database, it can receive input form the Clr interactor which is associated with the button which allows the user to clear the current request. Then there is one interactor for natural language requests (SpecRNL) and one for deictic requests (SpecRNL & Mouse) which combines input from voice device and mouse. Finally there is an interactor (Show Result) which maintains the state of the current request and when it receives an input trigger from a specific button (SendR interactor) then it sends a request to the database.

We can further refine this architectural description. However, this level of abstraction provides already useful insight to understand the possible dialogues with the user and their properties.

5.2. The LOTOS specification of MATIS

Our approach in building the LOTOS specification of the Interactive System considered consists in firstly identifying the interactors needed (which can be graphically represented by boxes) and then associating each of them with one LOTOS process structured following the general interactor model or one of its small variations by providing the actual gates. These LOTOS processes are then composed by indicating on which gates they synchronise. The composition processes operator of LOTOS behaves in such a way that apart from the gates on which processes have to synchronise, the available actions of the processes which are active at the same time can occur in a non deterministic way. We have introduced a control process too, which is composed with the expression describing the behaviour of the system and which introduces additional constraints on the dynamic behaviour of the specification considered to get the complete description of the implemented system.

LOTOS allows specifiers to distinguish among external and internal actions. In this case we make internal all the actions which are related to communication among interactors and external those related to user input or modification of the presentation of the user interface.

The resulting specification, which is reported in part in Fig. 6, is obtained by the composition of processes associated with the boxes in Fig. 5. The gates through which they can communicate (arrows in Fig. 5) are explicitly indicated in the corresponding LOTOS expression by the synchronization operator ([...] a1, a2, a3...]). We do not include, for space problems, both the remaining part of the specification in Fig. 6 and a more detailed LOTOS specification of the system considered which we have developed too.

In the first part of the LOTOS expression there are two processes (SpecR and SubmitR) which are two high level expressions for the part of the system dedicated to the specification of the request (SpecR) and the part dedicated to the performance of the user request.
5.3. Example of verification of MATIS properties

In order to reason about user interfaces properties it is important to identify in the formal
specification the set of actions associated with user-generated events and those associated
with modifications of the perceivable presentation of the user interface.

With reference to the MATIS specification graphically represented in Fig. 5, the two
sets of events are:

User action = {show_me_string, show_me_word, filling_of_slot_key, select_value,
word, move_in_book_icon, deictic_sentence, clear_request}
In this set of user-generated input actions there are actions associated with different tasks: sending the current request (show_me_string, show_me_word, move_in_book_icon), filling the fields of the request form by keyboard (filling_of_slot_key), providing deictic input (select_value, deictic_sentence), and natural language input (word) and clearing the content of the current request (clear_request).

**Modifications of the presentation** = {visualize_alert_window, highlight_icon_clear, visualize_word_received, visualize_request, highlight_send_command, visualize_window}

Most of these actions are associated with local feedback of the user input in each interactor. Then we have the visualize_window action which shows the result of the user query and visualize_alert_window which is associated with the presentation of an error message indicating that an incorrect request has been provided.

In next paragraphs we show an example of a set of user interface properties expressed by ACTL. We express each of them in abstract terms and then we give examples in terms of the specific instance of the MATIS system being considered.

### 5.3.1. Reachability

This property allows us to verify that a user interaction can generate a specific effect. The designer can be interested in reachability of two types of effects: semantic effects ad presentation effects. In the former the purpose is to achieve the access to some internal functionality (for example, the designer is interested to know whether from a certain state it is possible to activate the printing of a file), in the latter (which can be considered the usability property) the effect desired is a modification of the presentation for communicating some information included in the functional core (for example, the designer wants to know whether from a certain state it is possible to activate the presentation of the list of trains able to satisfy the constraints provided by the user). ACTL does not differentiate user and system actions. However, the designer who developed the specification knows what they are and so know how to specify the property of interest.

The next ACTL formula means that for all the possible futures (A operator) and for all the possible states (G operator), if a user action is performed then there exists at least one temporal evolution (E operator) during which there is a set of transitions on which we do not provide any constraint (true {true} operator) until (U operator) the Reachable_effect action has been performed ({Reachable_effect} true operator).

\[
AG([\text{user action}_x]E[\text{true} \{\text{true}\} U \{\text{Reachable_effect}\} \text{true}])
\]

Another example is checking that the system specified supports that once the user has started to edit an email message then it will be possible that this message will be delivered to another user. The reachability property expressed by the formula does not guarantee that the effect will be reached in all the possible temporal evolutions. However, this option would be too strong since in an Interactive System many things can happen as the user may start several interactions whose effects might conflict (in the example above the user can stop the editing of the email message and decide discarding it). One way to define stronger reachability properties is to say that:
This means that once the user has performed the considered action then we have a
temporal evolution during which the possible conflicting actions do not occur and finally
we reach a state by performing a certain action ($act_y$ in the property) from where for all
the possible temporal evolution the desired effect will eventually be reached. This means
that before the $act_y$ action some conflicting actions can occur (for this reason we use only
the E operator before it) but after that action we are sure that the desired effect will be
reached for all the possible traces of actions.

We can check, for example, that the specification supports that after the user has pressed
the print button then it is possible that there are temporal evolution during which no
conflicting actions are performed (in this case the cancel printing command is not selected)
until a specific action occurs (the send print command in our case) which guarantees that,
after this action, for all the possible temporal evolution, the desired effect will be reached
(the printing of the file in our example).

In Fig. 7 the user interface of the tool is shown when it visualises the answers about
verification of properties. In the first case the property asks whether it is possible that for
all the voice input ($word$ action) followed by keyboard input ($filling_of_slot_key$ action)
then for all the possible futures the request can be sent to the database (which is repre-
sented by the $request_for_database$ action): the answer is negative. In the second case,
after the same two user inputs we consider one temporal evolution where the user does not

clear the request and selects the book icon, then we can check that for all the temporal evolutions the request can be sent to the database. In this case we narrow the space of the considered temporal evolutions by adding the constraint that after the two user actions to provide requests for the database we consider the temporal evolutions during which the previous input is not cleared and then the action which triggers the request (move_in_book_icon) occurs.

In the case of failure in the verification of the property it is possible to ask for a counterexample by selecting the why button: a trace of actions not satisfying the given formula will interactively appear.

5.3.2. Feedback

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 the user performs an action then there is at least one temporal evolution during which the event associated with the specific user interface modification indicating that the input has been received by the system will occur. In terms of the abstract model this is expressed as:

$$\boxforall (\text{user action}x \rightarrow E \text{true} \cup \text{true} \cup \text{User interface appearance}x \text{true})$$

This property does not guarantee that the action considered will occur for all the possible temporal evolutions. In this case, this limitation is acceptable because, for example, the user may close the session before that the system presents the feedback of the previous interaction. The existence of at least one temporal evolution during which the feedback occurs is useful for showing that the system has been correctly designed to support this effect.

In our case study, we have verified that each user action is associated with a corresponding modification to the presentation of the user interface, itself corresponding to the feedback generated from the system to the user. Some user actions needed to build a request (for example input by voice which is represented by the word action in the specification) have two kinds of feedback, syntactical and lexical. This property thus has to be verified in both cases. For example, if one value is provided by voice (word action) the lexical feedback is provided by the natural language parser while the syntactical feedback is provided by the request form which is updated in order to include the value provided (see the table below):

$$\boxforall (\text{word} \rightarrow E \text{true} \cup \text{true} \cup \text{visualise_word_received} \text{true})$$

&

$$\boxforall (\text{word} \rightarrow E \text{true} \cup \text{true} \cup \text{visualise_request} \text{true})$$

All the possible user-generated actions in Matis have been evaluated against this property with positive answers. The & symbol means the logical and between the two parts of the property.

5.3.3. Continuous feedback

This property is stronger than simple feedback since besides requiring a feedback associated with all possible user actions, it states that this has to occur before a new
user action is performed for all the possible temporal evolution. Here we only consider lexical and syntactical feedback. However, for semantical feedback (for example, when the application produces a result of the query provided by the user) this is not often satisfied as it is possible, in many applications, to start a new interaction before receiving the system’s answer to the previous request in order to improve the time performance of a user session.

Using ACTL in abstract terms this property can be expressed as follows, where user_action_x is the user-generated input considered and ~user_action is a contraction for the logical AND among the negation of all the possible user actions. It means that when the user performs the action user_action_x, then for all the possible temporal evolution s/he cannot provide another input before that the system feedback for the first actions has been provided.

\[ AG([user\_action\_x]A[true\{\neg user\_action\}U\{User\ interface\ appearance\}true]) \]

The previous example of MATIS property describing feedback becomes as shown in next table, in the case of continuous feedback. In this case the property becomes more complex because we list all the possible user inputs which cannot occur before the feedback of the word action.

\[ AG([word]A[true\{\neg word&filling\_of\_slot\_key&\neg show\_me\_string&\neg show\_ deictic\_sentence&\neg select\_value&\neg move\_in\_book\_icon} U[visualise\_request]true]); \]
\[ \& AG([word]A[true\{\neg word&filling\_of\_slot\_key&\neg show\_me\_string&\neg show\_ me\_word&\neg deictic\_sentence&\neg select\_value&\neg move\_in\_book\_icon} U[visualise\_word\_received]true]); \]

5.3.4. Reversibility

This property is a type of implicit undo. It means that users can perform part of the actions needed to fulfil a task and then perform them again if necessary before the task has been completed in order to modify its result.

\[ AG([user\ action\_x] E[true\{\neg task\_performance\} U [action\_cancelling\_previous\_effects] E[true\{\neg task\_performance\}U [user\ action\_x] EF < task\_performance > true]) \]

The property in the table above means that whenever a user action occurs (user_action_x event) we have the possibility of a temporal evolution during which the task is not performed, the user cancels previous effects, and then user actions are still possible for providing new input and finally performing the task.

With reference to the MATIS specification this property can be applied to the case when the user deletes the effects of a set of possible user actions previously performed. This means that after the actions allowing users to fill in the slots of a Request Window, the values contained in the slots can be cancelled by the clear_request command before
sending a request with these values to the database. The clear command gives the user the possibility, if errors have been made or the current task has been changed, to cancel the values inserted beforehand in the slots and go back to a situation where all actions are still possible.

Next table shows the property in the specific MATIS case where the user can start the interaction by selecting a value among a set of predefined values (select_value action), and then s/he can clear the request, compose new values, and finally send the request to the database.

\[
AG\{select_value\}E[true \{\neg request\_for\_database\}]U[clear\_request
E[true \{\neg request\_for\_database\}]U[filling\_of\_slot\_key|select_value|word
deictic\_sentence]EF < request\_for\_database > true])
\]

5.3.5. Possibility of performing a task at any state

This property states that starting from all the possible states which the system behaviour can reach (AG operator) there is at least one temporal evolution (E operator) during which, at some time (F operator), the action associated with the task performance occurs.

\[
AGEF(<task\_performance > true)
\]

This is a very compact and powerful way to check all the space of the possible states which can be reached by the system considered. In our case we can verify the property on the action request_for_database. This property is an indication of good architectural design as it means there are no one-way trapdoors: no action which allows the system to reach one state from where it is no longer possible to send requests to the database.

\[
AGEF(<request\_for\_database > true)
\]

6. Verification of properties of human–computer dialogues with an infinite number of states

We have shown examples of user interface properties and their verification on the specification of a case study which allow us to understand the features of the possible dialogues. However, designers should be aware that it is not always be possible to build a finite model describing an interactive software application.

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 a immediate way the features of concurrent systems.

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, in ref. [26] a comparison of the power of different subsets of LOTOS operators is provided. It shows 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.

6.1. 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 Fig. 8. We describe the objects illustrated in Fig. 8, 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 expressions which can be transformed into a finite state automaton.
Thus the behaviour of this expression can be verified without particular problems considering also that the related number of states is limited.

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.

In Fig. 9 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 Fig. 9 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 (..), in interleaving composition (||| operator) with a new interactor that manages a new text window.

```
Application[... ] ||| Text[int_text, text, up_pres_text] endproc
```

This LOTOS expression cannot be described by a finite state machine. 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.

6.2. A discussion on user interface properties that can be verified on models with an infinite number of states

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 [24] 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 [27] 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.

This approach provides a method to define the validity of a property on a non-finite system. It considers the classical subdivision of temporal properties into subclasses: liveness properties (something good eventually happens) and safety properties (nothing bad can happen). 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 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.

Properties can be divided into two classes: universal (for all computations, A operator) and existential (for one computation, E operator). 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, AFEGf, 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 are considering 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).

Now we want to analyse 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. Examples of properties useful in the user interface field which can be considered liveness properties are: reachability, possibility of performing a task at any state, visibility (which expresses the possibility that for each user action there will be a modification of the presentation of the user interface to present the result of the state modification generated by the user input), recoverable error (after a recoverable error, several actions are needed to return to the previous state whereas an error is unrecoverable if it does not allow the user to perform the desired task in the current session).

In the case of liveness properties, 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.

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 (AG{user actions}EX {User interface appears}true) or Minimal
error which means that immediately after an erroneous action, the user can perform one action which is useful for task performance (\(AG[recoverable\_error]EX[useful\_task\_action]\true\)).

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. Unfortunately, there are some HCI properties on which approximations of the infinite automaton cannot give any answers. This is the case of all formulae that contain negative formulae. This problem affects the evaluation on some CARE properties [28] too. We can express them in ACTL, and if we consider Complementarity (the possibility to achieve a desired effect using two modalities which are both indispensable, in an independent order) and Assignment (the possibility to reach a desired effect using only one modality) we still need to use negations to express them thus making them unsuitable for verification by this approach for non-finite models.

7. Conclusions

The importance of being able to reason rigorously about user interface properties is becoming increasingly recognised. Model checking is an interesting approach to support this kind of evaluation. Beside, the availability of general purpose automatic tools for the support of model checking makes it possible to address specifications of realistic case study with medium dimensions. However in these cases 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 this paper we first consider the application of model checking techniques when a finite model of the Interactive System considered can be built. We indicate examples of general properties and show their application to a specific case study. Then 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 creating an environment more suitable for user interface designers which allows them to use model checking techniques also when they are not expert in formal methods as it often happens. Furthermore we will consider the development of more powerful model checking techniques for our application area.

Acknowledgements

I wish to thank the anonymous reviewers for their comments which were useful for improving the presentation of the final version of the paper. Amodeus II was an interesting European project during which many ideas discussed in this paper were stimulated. Mefisto is an interesting European project during which we are extending these ideas for applying them to the application domain of air traffic control.
Appendix A. An Introduction to ACTL

We now outline some concepts of ACTL in order to understand better the examples provided in the sections below. It is possible to verify automatically properties expressed by ACTL when the complete system behaviour is described by a Labelled Transition System (LTS). In this kind of system transitions are labelled to describe the actions which cause state changes.

An LTS is a 4-tuple \((Q, Ac \cup \{\tau\}, \rightarrow, Q_0)\) where:
- \(Q\) is a set of states;
- \(Ac\) is a finite, non-empty set of visible actions;
- \(\tau\), which represents the internal, not visible actions, is not in \(Ac\);
- \(\rightarrow \subseteq Q \times (Ac \cup \{\tau\}) \times Q\), is the transition relation; an element \((r, a, q)\) is called a transition, and is written as \(r-a->q\);
- \(Q_0\) is the initial state.

In ACTL it is possible to express action formulae which are obtained just indicating actions or their composition with negation, logical and, logical or operators, in these formulae we can use \(true\) to indicate all actions and \(false\) to indicate no action.

The operator \(A\) means that the related properties are verified by observing all the possible futures, while \(E\) indicates that the related property will only be verified in some observed future.

We can summarise the main ACTL operators with the associated meaning in the following list:
- \(AX\)—in all possible next states
- \(AG\)—in all states of all possible futures
- \(AF\)—in some state of all possible futures
- \(EX\)—in the next state of some possible future
- \(EG\)—in all states of some possible future
- \(EF\)—in some state of some possible future

We will denote with \(a, b, a_1, \ldots\) actions, with \(af, af_1, \ldots\) action-formulae and with \(form, form_1, \ldots\) ACTL formulae which have to be satisfied by the considered state.

\[\text{[select menu]} \text{ form}\]

which means that considering the system from the initial state, for all the states of all computations that we reach if the select menu action occurs then form has to be true.

\[<\text{select menu}> \text{ form}\]

which means that from the initial state there is at least one computation in which no action is observed until the action select menu has occurred and then form has to be true.

\[AX \{af\} \text{ form}\]

this means that for all the possible futures the next action will satisfy \(af\) and then the LTS will be in a state satisfying form. For example:

\[AX \{\text{window\_appear}\} \text{ true}\]
which means ‘for all the possible futures in order to get to the next state the action associated with the visualization of the window has to occur’.

\[ A \{ \text{form} \ U \{ \text{af1} \} \ \text{form1} \] 

This means that for all the possible futures we will have traces of actions satisfying \( \text{af} \) and states satisfying \( \text{form} \), until \(( \text{U} \) operator\) there will be an action satisfying \( \text{af1} \) and a state satisfying \( \text{form1} \). When \( \text{form} \) and \( \text{form1} \) are true it means that no restriction is indicated on the state. For example:

\[ \text{AG} \{ \text{button} \_ \text{press} \} \ A[ \text{true} \{ \neg \text{button} \_ \text{press} \} \ U \{ \text{selected} \_ \text{position} \} \text{true} \} \]

which means ‘whenever the button press occurs, then for all the possible temporal evolutions it cannot occur again \( \neg \text{button} \_ \text{press} \) means that the button_press event does not occur) until the selected position has been delivered to the application’.

If in the previous expressions we substitute \( A \) with \( E \) we have to substitute in the explanation ‘for every path or transition’ with ‘there exists a path or transition’.

For example:

\[ \text{AG} \{ \text{city} \_ \text{selection} \} \ E \{ \text{request} \_ \text{for} \_ \text{database} > \text{true} \} \]

which means ‘whenever a city has been selected then one temporal evolution exists during which, at some time, the request for the database has been delivered’.

References


