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

Context Sensitive User Interfaces

J. Creissac Campos and F.M. Martins

Published in Collaboration with the British Computer Society

BCS FACS

Springer

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76105-5

Context Sensitive User Interfaces

José Creissac Campos
jfc@di.uminho.pt
Departamento de Informática, University of Minho
Largo do Paço, 4719 Braga Codex, Portugal

Fernando Mário Martins
fmm@di.uminho.pt
Departamento de Informática, University of Minho
Largo do Paço, 4719 Braga Codex, Portugal

Abstract

This paper presents a conceptual design model for user interfaces (MASS[1]) and a general formalism for dialogue specification (Interaction Scripts) which are the most important components of an approach to the methodological, iterative design of Interactive Systems from formal, model-based specification of both the application and the User Interface (UI).

This approach allows the integration of both dialogue and application semantics from the beginning of the design process, by using prototypes derived from both specifications.

Assuming that all the application semantics is available at early design stages, the MASS model defines a set of guidelines that will enforce the designer to create user interfaces that will present a *prophylactic* instead of the usual *therapeutic* behaviour. By a *prophylactic* behaviour it is meant, metaphorically, that the UI will exhibit a behaviour that prevents and avoids both syntactic and semantic user errors, in contrast with the most usual *therapeutic*, or error recovery, behaviour.

The dialogue specification formalism (Interaction Scripts) despite being general, in the sense that it may be applied to the specification of any kind of dialogue, is specially suited to the specification of UIs with the behaviour prescribed by the MASS design model. In addition, it is independent from concrete environment details, therefore allowing for different implementations of the same specification, that is, different *looks* and *feels*.

The operational semantics of the Interaction Script notation is also presented in terms of Petri-Nets that are automatically generated from the Interaction Script specification of the dialogue controller.

Keywords: Assisted Human-Computer Interaction, Formal Specification, Rapid Prototyping, Semantic ”feedforward”, Methodology of Interactive Systems

1 Introduction

In this paper we put forward some of the ideas and tools developed under a methodological work to apply formal methods to the systematic development of User Interfaces (UIs).

Usually, methodologies for the specification of human-computer interaction specify the dialogue in a loose relation with the specification and design of the functional layer of the system (the application) (cf. [1, 2, 11]).

This arises from an inadequate interpretation of the Separation Principle [10]. By completely separating the dialogue layer from the semantics of the application, the quality of the interactive system and the flexibility of its design is compromised. Because *semantic feedback* is not easily specified, interfaces specified (and possibly prototyped) this way tend to be static, not being able to reflect in its behaviour (*feel*) the state of the application.

We argue that, although the Principle of Separation should be taken into account (for modularity reasons), a strong link between both specifications must be established at design level, for the sake of the quality of the final interface and also for promoting its iterative design in the context of the application semantics.

The level of abstraction at which the UI specification is made is also important. Most proposed notations specify at a level too close to key strokes and mouse clicks [1, 11, 3]. We argue that the dialogue specification must only be concerned with the flow of information between the user and the application layer. The exact way in which information

[1] Syntactic and Semantic Assisted Mode

is presented to and manipulated by the user, should be left to the responsibility of the toolkit being used, in the same way as the exact implementation of the data being manipulated by the application layer is its responsibility.

Finally, a dialogue specification should be constructive, allowing for the generation of dynamic prototypes, for iterative, validated design. Based on this assumptions we propose a conceptual design model for user interfaces (MASS) and a specification formalism (Interaction Scripts). The MASS model [12] is a conceptual model for user interface design. The model aims at defining the properties that the interaction should obey.

Interaction Scripts [12, 9, 7] were developed as a formalism to specify dialogues which may satisfy the MASS model. Usually an Interaction Script will specify the dialogue leading to an application function call. The dialogue is specified by an expression defining its valid traces. A set of context validity conditions specify the conditions in which the Interaction Script may be activated and the values read are validated, based on the state of the application. The operational semantics of Interaction Scripts is defined by Petri Nets, and a dynamic prototype[2] of the user interface may then be generated from the specification.

2 The MASS Model

The MASS model is a conceptual model of the UI. Taking into account the taxonomical works of Rohr and Tauber [19] and Nielsen [15], the MASS model may be characterized as a conceptual design model of the UI of type RUi or DUi. This means that being a conceptual design model, MASS defines a set of properties (or guidelines) that the Designer (D) of the UI should obey in order to design a UI with the intended characteristics. The model may also be of interest to the human factors Researcher (R). Following the above referred classification it is also both an internal (existing inside the person intended to use it) and external model (may be formally or informally communicated). It is also a structural model because it follows some main principle (in this case fully assisted interaction in order to prevent user errors) and it is generic (may be applied to different kinds of objects and problems). The key properties prescribed by MASS are: syntactic assistance, semantic assistance, flexibility, and mode commutation. Each one of the properties is explained below.

- **Syntactic Assistance**: the interaction language of the UI hides from the user the concrete syntax and details of the language used to communicate with the application; by using structured editing, either explicitly (cf. Interaction Archetypes[3] [14]) or implicitly (cf. the Interaction Script notation presented in this paper), the user always writes syntactically correct commands.

This property relates to the fact that users think in an informal way, but have to interact in a formal interaction language. This implies a great probability of syntactic errors. The Syntactic Assistance suggested by MASS and implemented through a scheme of structured editing allows for the elimination of such errors. The interaction style that best fits to structured editing is the menu based style (navigation through options).

- **Semantic Assistance**: the model must ensure that the values/actions the user can choose are correct at any state; therefore the user will not cause error situations.

Semantic assistance together with syntactic assistance, allows for user interfaces with a *prophylactic* perspective of interaction. Syntactic errors, as well as meaningless sentences are prevented, freeing the application layer from error treatment and recovery. Error messages and user actions to deal with them are also minimized, with evident benefits to the interaction process.

The idea of *prophylactic* interaction (a metaphor) is used to stress the fact that we are able to design user interfaces which are error-sensitive, by providing both syntactical and semantical assistance to the user. Semantical aid may only be correctly provided to the user if the interactive system is able to “calculate” the actual context. So, the UI will present a look (presentation) and a feel (behaviour) which are sensitive to (that is always aware of, informed of) the application state. The key idea is to *feedforward* application semantics to the UI without disregarding the usual need for semantic *feedback* in the output. However one must understand that the first is concerned with correct input while the other is concerned with clear, understandable output.

[2] Unlike a static prototype, that only shows the *look* of the user interface, a dynamic prototype shows both the *look* and *feel* of the user interface.
[3] Archetypes are terms with variables that represent the possible navigations needed to interactively synthesize a correct application command.

- **Flexibility in the Arguments Input Order**: the order by which the arguments are read may also be specified.

Because the order of the arguments to be read may be flexible (possibly concurrent), that depending on the semantics of the application, we need to use a more powerful structure (Nets) than the usual trees or graphs used in conventional structural editing.

- **Mode Commutation**: in implementations where it makes sense, the model must allow the syntactic-semantic assistance to be disconnected by user's choice.

So, a user can choose between an assisted or a command mode. However, if an error occurs during the input of a command (let us say interactive expression) the assisted mode is automatically offered as an aid to the user.

In the next session we present the formalism for UI specification.

3 The Interaction Script Formalism

The Interaction Script formalism despite being general [13], is particularly well suited for the specification of dialogue controllers that follow the MASS model. In fact, each Interaction Script can be seen as a mini-MASS, specifying an assisted interface for an API-based application.

The Interaction Script formalism is used to describe, at a high level of abstraction, the control of the flow of interaction that the Dialogue Controller must execute. Each Interaction Script describes all the steps of the dialogue that will lead to the correct synthesis of an API-call[4]. This correctness must be both syntactic and semantic.

The synthesis of a command, also implies the synthesis of its arguments. So, Interaction Scripts will provide for argument validation, as well as for the specification of the argument input order, which may be random in certain cases.

Methodologically, in a first stage, an Interaction Script is written for each command of the application. In a second stage, Interaction Scripts are written to specify the correct dialogue structure (which command can be called and when). Although more natural, this bottom-up strategy is not mandatory and a top-down approach can also be used.

The operational semantics of Interaction Scripts is represented by Petri Nets. These Petri Nets are automatically generated allowing for fast prototyping [18]. This approach is, then, positioned at a higher level than some systems that specify UI directly using Petri Nets (cf. [5])

3.1 Properties

The development of the Interaction Script formalism was made bearing in mind the need for a high degree of expressive power. Let us see the main properties of the formalism:

compositionality , that is, the possibility of building complex specifications by combining simple Interaction Scripts;

local reuse , that is, the possibility of an Interaction Script being imported by, or included in, another Interaction Script, in this latter case the included Interaction Script will share the variables of the one which includes it;

sub-dialogue concurrency , that is, the possibility of specifying that events may occur concurrently;

specification of external behavior , by specifying the acceptable traces of interaction, that is, the correct event sequences;

declarative specification of internal behavior , by means of event-condition-action triples, representing the internal state transitions of the dialogue controller;

asynchronous events treatment , that is, events that are originated by non deterministic user actions and that imply immediate action from the dialogue controller. Example of such events are `Cancel`, `Ok`, `Reset` and `Apply` that will be described later;

[4] From now on will call it generically "command".

data synthesis interaction , that is, the possibility of specifying restrictions on the order of argument reading, and also the use of pre-defined interactors for specific data types;

error treatment , that is, specification of handling code for exception situations, which may arise whenever the MASS model is not fully obeyed;

localization of interaction effects , through the explicit declaration of the effects of each event in the internal state of the application, of the presentation, and of the dialogue controller;

presentation abstraction , as non commitment with the low level semantics of actual presentation technology, even though allowing for the existence of different views of the same interactive object - for each Interaction Script there will be at least a presentation descriptor (DA [5]) that maps the Interaction Script to a concrete presentation, we may have several DA attached to each Interaction Script, and the actual DA in use may be changed at run time;

prototyping , that is, from a specification written with Interaction Scripts a running prototype of the dialogue controller can be automatically generated.

3.2 Syntax and Static Semantics

We will now present the syntax and static semantics of the Interaction Script formalism. As we said before, an Interaction Script usually describes the dialogue needed to synthesize an application command. We need to specify the dialogue structure that leads to the choice of a command and the synthesis of their arguments. We considered three types of Interaction Scripts:

- DECISION - this type of Interaction Script is used to structure the dialogue helping the user to select the operation that he/she wants to execute.
- SYNTH - this type of Interaction Script describes the dialogue steps that will lead to an operation call.
- VALSYNTH - this type is similar to SYNTH but returns the value obtained from the execution of the operation.

We have also defined some Interaction Scripts for low-level input of some pre-defined data types (finite functions, sets, etc. - cf. data synthesis interaction)

An Interaction Script starts with the keyword `DefGI`[6] and ends with `EndGI`. Each Interaction Script is divided into two main components (see **IS 1** and **IS 2**): the Declarations component, and the Behaviour describing component. All identifiers of variables and of other Interaction Scripts that are used in an Interaction Script are defined in its Declarations component. In the Behaviour component the interaction flow is defined. Each component may have several clauses. We will now list the clauses that may be used in all Interaction Script types.

The Declarations component may include declarations of:

- the type of the Interaction Script (clause TYPE);
- the imported Interaction Scripts (clause EXTERNAL);
- the locally defined Interaction Scripts (clause SUBGI);
- the local variables (clause STATE-CTRL);
- the application variables that are to be accessed by the Interaction Script (clause STATE-APL).

For all Interaction Script types the Behaviour component may include:

- a context condition that defines when the Interaction Script may be activated (clause CONTEXT);

[5] From the portuguese *Descritores de Apresentação*.
[6] `GI` comes from the portuguese for Interaction Script: `G`*uiões de* `I`*nteracção*.

- the description of all the event sequences that the Interaction Script accepts (clause EVSEQ).

The combinators for writing EVSEQ are:

- exp1 + exp2 - **choice**, only one expression can be selected;
- exp* - **repetition**, the dialogue in `exp` may be repeated;
- exp1.exp2 - **sequence**, `exp1` followed by `exp2`;
- exp1|exp2 - **synchronous parallelism**, both sub-dialogues may proceed (events are accepted) in parallel, and the main dialogue will finish whenever one of the sub-dialogues finishes.
- exp1||exp2 - **asynchronous parallelism**, both sub-dialogues may proceed in parallel, and only when both sub-dialogues are finished will the main dialogue finish too.
- $exp1[exp2]$ - **interruptible sequence**, the dialogue specified by `exp1` can be interrupted by `exp2`; if that happens `exp2` must end before `exp1` can continue.

The Interaction Script **Menu** shown in **IS 1** is an example of a DECISION Interaction Script adapted from [8], and it is part of a simple dictionary interface specification. This Interaction Script defines a dialogue in which the user must choose one of five options, the dialogue being terminated when he/she chooses **End**.

In this case we have a locally defined Interaction Script: **End**. **End** is the empty Interaction Script and does nothing, being used by **Menu** just to end the dialogue.

IS 1 *DECISION type Interaction Script*

```
DefGI Menu
  Declarations
    TYPE DECISON
    EXTERNAL GInit, GManage, GDefWord
    SUBGI End
  Behaviour
    EVSEQ (GInit + GManage + GDefWord)* + End
SubGI
    DefGI End
    EndGI
EndGI
```

In this example we show how the specification of the event sequences that an Interaction Script accepts (cf. specification of external behaviour) can be defined by composition of other Interaction Scripts (cf. compositionality and local reuse). In this case at most one Interaction Script can be active at each moment. However, if we had used the | or || combinators we could have more than one Interaction Script simultaneously active (cf. sub-dialogue concurrency).

The Declarations component of an Interaction Script of type SYNTH may include the clauses listed before and also the declarations of:

- the arguments of the operation to call (clause ARGS);
- the variables of the Presentation that are to be accessed by the Interaction Script (clause STATE-UI).

The Behaviour component includes the clauses listed before and also the declaration of:

- conditions to be verified and actions to be executed after each event (TRANS);
- actions to execute upon Interaction Script start (clause INIT);
- the operation to call (clause EXEC).

Their are some predefined events that can be included in TRANS:

- OK - when this event is present in TRANS, the Interaction Script ends only when the event happens, this event is only available when EVSEQ is finished;
- CANCEL - this event is always available (when present in TRANS) and ends the Interaction Script without invoking EXEC;
- APPLY - similar to OK but the dialogue is restarted;
- RESET - similar to CANCEL but the dialogue is restarted.

In **IS 2** the Interaction Script **GDelWord** specifies the synthesis of the **DELWORD** operation call[7]. We can see that this synthesis is valid only if the condition "not(EMPTYDIC())" holds (cf. CONTEXT clause). After reading the word (input(w)), if the word exists ("EXISTWORD(w)" is true) then "m = DEFWORD(w)" is executed[8] otherwise an error message is shown (cf. TRANS clause for input(w)).

IS 2 *SYNTH type Interaction Script*

```
DefGI GDelWord
  Declarations
    TYPE SYNTH
    ARGS w: Word
    VAR-UI m: Meaning
  Behaviour
    CONTEXT not(EMPTYDIC())
    INIT w = "";
         m = ""
    EVSEQ input(w)
    TRANS input(w): EXISTWORD(w) => m = DEFWORD(w)
                           EXCEP out("Word does not Exist!")
          OK:
          CANCEL:
    EXEC DELWORD(w)
EndGI
```

This example shows how the TRANS clause allows us to have declarative specification of internal behaviour, asynchronous events treatment (cf. OK and CANCEL), and error treatment.

The type VALSYNTH is identical to the type SYNTH. The only difference is that it declares the type of the result that it will produce.

3.3 Operational Semantics

The operational semantics of the Interaction Scripts is based on an extension of Labelled Petri Nets (LPN)[17] described in [6] and called Labelled-Guarded-Interruptible Petri Nets. Basically these are Labelled Petri Nets extended with two features [12]:

- the association of a condition to each transition, so that only if this condition is true the transition will fire;
- the possibility of a net being interrupted by another net, the first being resumed only when the second net ends.

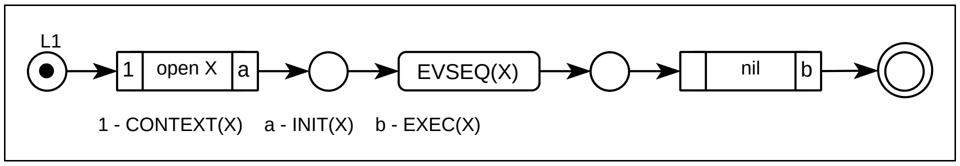

Figure 1: Net for a generic Interaction Script

3.3.1 Generic Interaction Script Semantics

Let us now show the structure of a net that represents a generic Interaction Script.

The following conventions will be used:

- The tokens will show the initial marking.
- The places of the final marking will be represented by a double circle.
- Transitions will be represented by rectangles with margins. In the ”entrance” margin we write the conditions associated with the transitions, and in the ”exit” margin we write the action associated with the transition.
- Rounded squares represent abstractions of subnets.

Figure 1 represents the execution of a generic Interaction Script *X*. We can see that the Interaction Script can only start if the condition in the CONTEXT clause of *X* is true (or non existing), and that when that happens the actions in INIT are executed. Upon start the Interaction Script will behave according to a net representing its EVSEQ clause. When that net ends, the Interaction Script will end automatically and the expression in EXEC will be executed. In this example we are not considering asynchronous events, which will be introduced later.

3.3.2 The EVSEQ Clause

The simplest EVSEQ clause we can have is the one which includes only one event. For example:

EVSEQ gi(var)

where *gi* is some Interaction Script identifier and *var* a variable. In figure 2 we show the net corresponding to this EVSEQ, and we call *X* the Interaction Script that contains this EVSEQ clause.

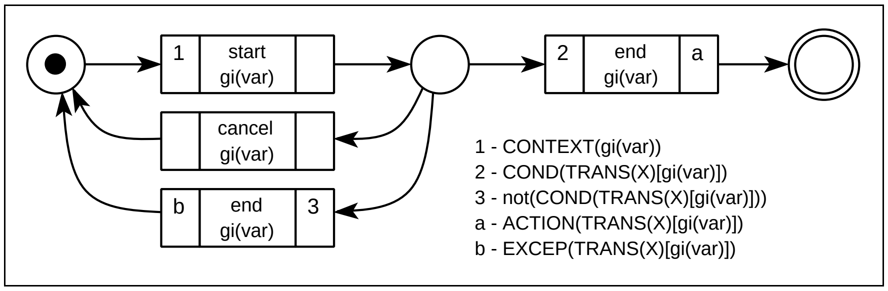
1
start gi(var)
2
end gi(var)
a
cancel gi(var)
b
end gi(var)
3
1 - CONTEXT(gi(var))
2 - COND(TRANS(X)[gi(var)])
3 - not(COND(TRANS(X)[gi(var)]))
a - ACTION(TRANS(X)[gi(var)])
b - EXCEP(TRANS(X)[gi(var)])

Figure 2: Net for gi(var)

We can see that the first transition can only fire (with the start event) if the context condition of *gi* is true. After that, three transitions can happen. One corresponds to the cancel event and can be fired as soon as the event occurs. The

[7] `DELWORD` deletes a word from the dictionary.

[8] Note that `m` is a User Interface variable. So, the meaning of the word is passed to the interface (cf. localization of interaction effects).

other two correspond to the end event: one for when the condition associated with *gi(v)* (in the TRANS clause) is true and one for when that condition is false. In the first case, the actions associated with *gi(v)* in the TRANS clause are executed.

With this example we can also see that each event in EVSEQ generates three events at the petri net level: start, cancel, and end of the event.

Each combinator has an associated Petri Net that combines the nets of its arguments to build the Petri Net of the global EVSEQ expression.

3.3.3 Asynchronous events

In figure 3 we show the same net of figure 1, now completed with the asynchronous events CANCEL and OK. The CANCEL event can happen at any time after the start of the Interaction Script and ends the dialogue. The OK event is placed after the net of EVSEQ so that the dialogue does not end automatically after EVSEQ. The RESET and APPLY events are equivalent to CANCEL and OK but the token goes to an initial place.

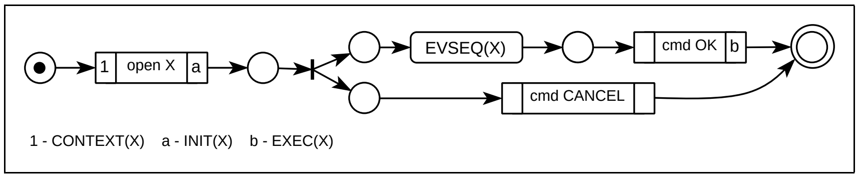

Figure 3: Net for OK and CANCEL events

4 Fast Prototyping

Building on the fact that Interaction Scripts can be translated to Petri Nets, a prototyping tool was developed. At the moment an Interaction Script compiler and the runtime system[8] have been implemented. The compiler translates Interaction Scripts into Petri Nets and the runtime system animates these nets.

The architecture of the runtime system is shown in figure 4. It has three components: the Dialogue Controller (specified with Interaction Scripts), an application module (that links the dialogue controller to the application) and a presentation module (that implements the *look* of the dialogue with the user).

While the Interaction Script specification of the dialogue controller defines the *feel* of the UI, its *look* depends upon the Presentation Model. A presentation description language (DA) was developed for the description of the *look* of the UI. This language has the ability to specify more than one *look* for the presentation, allowing the user to choose, in runtime, the *look* he/she prefers.

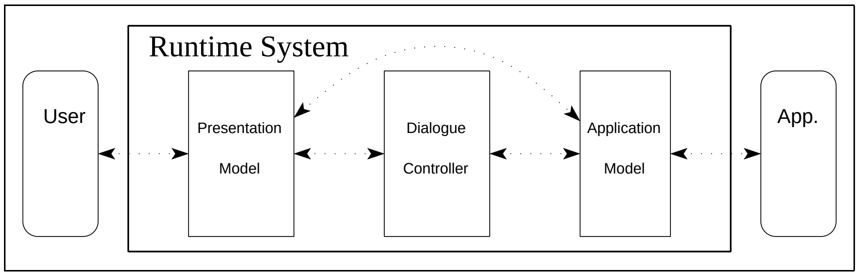

Figure 4: Runtime System Architecture

Each component of the runtime system is a separate process communicating through UNIX channels. So, we can "plug" and "unplug" different application and presentation modules to the dialogue controller. This has two main advantages. On the side of the application, it allows us to use the prototype of the UI either with the prototype or the final implementation of the application. This way the UI prototype can be used throughout the reification process of the specification of the application. On the side of the presentation, we may have different kinds of presentation modules depending on the technology available. At present there are two presentation modules available, one for vt100 terminals and one for X11.

The dialogue controller and the vt100 presentation module were written in a functional language called CAMILA [4], the X11 presentation module was written in Tcl/Tk [16] and CAMILA. The presentation module implementation language will depend on the final language used in the implementation of the application.

In figures 5 and 6 we show examples of dialogue portions generated by the runtime system using the X11 presentation. The first is the dialogue generated for the Interaction Script GDelWord (an appropriate DA was supplied), the second was taken from [12].

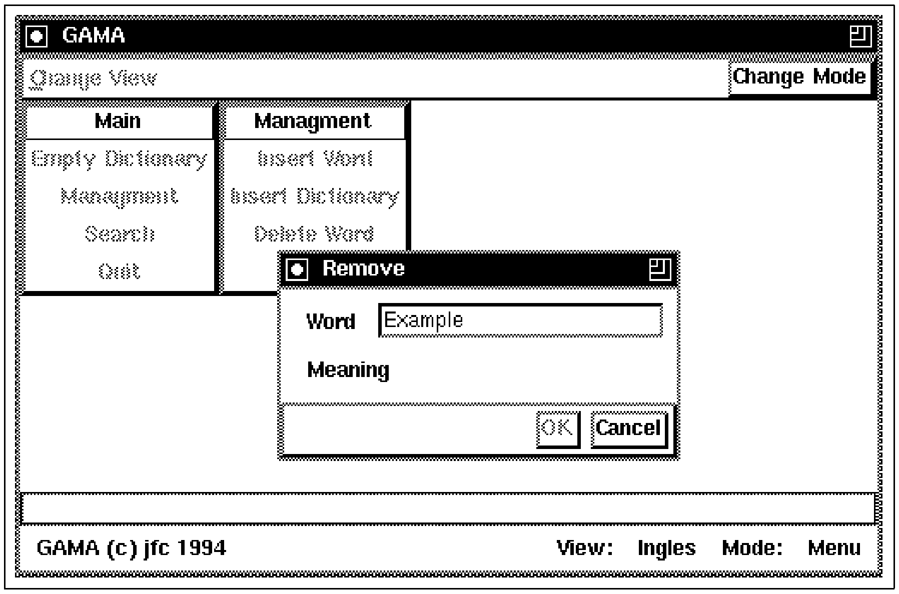

Figure 5: Dialogue generated for the Interaction Script GDelWord

5 Conclusions

In this paper we argue that the user interface of an interactive system, although kept separate from the application layer, must include enough semantic information to be able to reflect the internal state of the application layer. We also argue that the dialogue specification must be done at a high abstraction level, making it independent from details of the actual interaction technology being used in a given context. This way, the specification becomes generic and can be implemented in different types of environments. Therefore, our approach supports separate implementation but integrated and iterative design.

We propose a conceptual model of interaction (the MASS model). This model comprises a set of properties that a user interface should obey, and incorporates an implicit user model. This user model takes into account the erroneous and unpredictable behaviour of users, as well as his/her vague knowledge of the interaction language. The two most important properties defined by the MASS model (Syntactic Assistance and Semantic Assistance) define a *prophylactic approach* to user interface design and implementation, useful in many API-based applications, by *feedforwarding* application semantic information at early design stages.

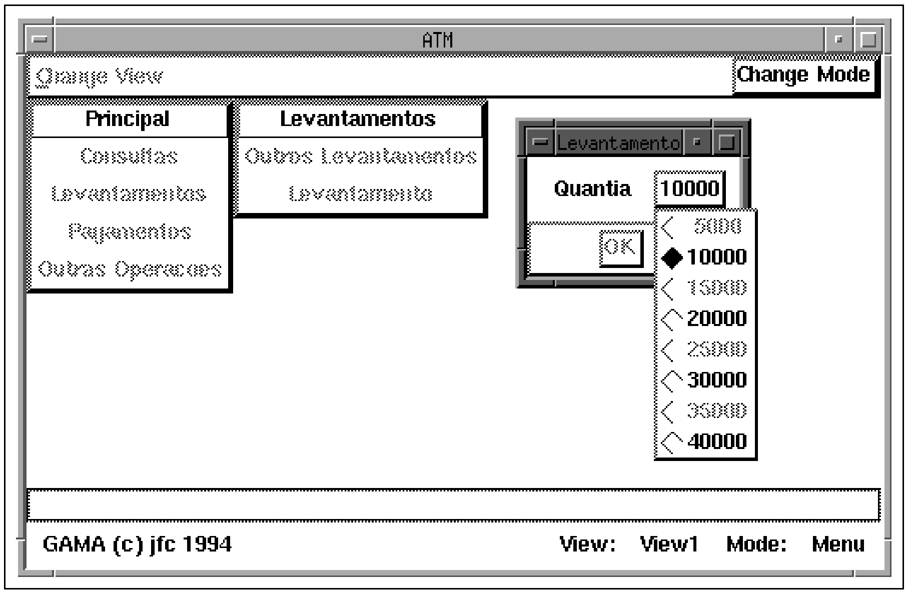

Figure 6: Screen Shot of the Runtime System

We then propose a specification formalism (the Interaction Script formalism) that allows for the specification of user interfaces obeying the MASS model. Specifications done with the Interaction Script formalism describe, at a high level of abstraction, the control of the flow of dialogue between the user and the application layer. Among other properties the Interaction Script formalism allows for assisted interface specification, reuse, dialogue concurrency, fast prototyping and iterative design.

The prototype of a specification done with Interaction Scripts is obtained by an automatic translation into Petri Nets. This UI prototype can be used throughout the reification process of the application layer (from its own prototype to its final implementation), and it can be used to generate interfaces with different kinds of interaction technology.

All these tools are working together in a system still being developed called GAMA-X [7, 8].

Acknowledgments

The authors wish to thank the referees for their comments and remarks, which we believe helped to improve the contents of the final version of this paper.

References

[1] Abowd, G. D. Agents: Communicating interactive processes. In Diaper, D. et al. (Eds), Human-Computer Interaction - INTERACT'90, pages 143–148. Elsevier Science Publishers, 1990.

[2] Alexander, H. Formally-based tools and techniques for human-computer dialogues. Ellis Horwood Series in Computers and their Applications, 1987.

[3] Alexander, H. Structuring dialogues using csp. In Harrison, M. and Thimbleby, H. (Eds), Formal Methods in Human-Computer Interaction, chapter 9, pages 273–295. Cambridge University Press, 1990.

[4] Barbosa, L. and Almeida, J. J. CAMILA by example. Internal report, DI/INESC, Universidade do Minho, 1991.

[5] Bastide, R. and Palanque, P. Petri nets with objects for the design, validation and prototyping of user-driven interfaces. In Diaper, D. et al. (Eds), Human-Computer Interaction - INTERACT'90, pages 625–631. Elsevier Science Publishers, 1990.

[6] Biljon, W. Extending petri nets for specifying man-machine dialogues. Int. J. Man-Machine Studies, 28:437–455, 1988.

[7] Campos, J. C. GAMA-X Geração Semi-Automática de Interfaces Sensíveis ao Contexto. MSc thesis, Departamento de Informática, Universidade do Minho, 1993.

[8] Campos, J. C. Gama-X - programmer's manual. Technical report, Departamento de Informática, Universidade do Minho, 1995.

[9] Campos, J. C. and Martins, F. M. Automatic generation of user interfaces at prototype level. Technical report, Project EUREKA-SOUR - Olivetti Ricerca/INESC-Braga, 1994.

[10] Casey, B. and Dasarathy, B. Modelling and validating the man-machine interface. Software Practice and Experience, 12(6):557–569, June 1984.

[11] Cockton, G. Designing abstractions for communication control. In Harrison, M. and Thimbleby, H. (Eds), Formal Methods in Human-Computer Interaction, chapter 8, pages 233–271. Cambridge University Press, 1990.

[12] Martins, F. M. Métodos Formais na Concepção e Desenvolvimento de Sistemas Interactivos. PhD thesis, Departamento de Informática, Universidade do Minho, 1995.

[13] Martins, F. M. Specifying interaction with interaction scripts: A comparative study. Technical report, DI/INESC, Universidade do Minho, January 1996.

[14] Martins, F. M. and Oliveira, J. N. Archetype oriented user interfaces. Computer & Graphics, 14(1):17–28, 1990.

[15] Nielsen, J. A meta-model for interacting with computers. Interacting with Computers, 2(2):147–174, August 1990.

[16] Ousterhout, J. K. Tcl and the Tk Toolkit. Addison-Wesley Publishing Company, Inc., 1.st edition, 1994.

[17] Peterson, J. Petri nets. Computing Serveys, 9(3):223–252, September 1977.

[18] Reps, T. and Teitelbaum, T. The synthesizer generator: A system for constructing language-based editors. In Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[19] Rohr, G. and Tauber, M. Representational framework and models for human-computer interfaces. In van der Veer et. al. (ed), Readings on Cognitive Ergonomics - Mind and Computer. Springer-Verlag, 1984.