Design of User-Driven Interfaces Using Petri Nets and Objects

Philippe A. Palanque[1], Rémi Bastide[1], Louis Dourte[2] and Christophe Sibertin-Blanc[3]

1: L.I.S., Université Toulouse I
Place Anatole France, 31042 Toulouse Cedex, France

2: D.I.R.O., Université de Montréal
C.P. 6128 succursale A, Montréal (Québec) H4A 3L4, Canada

3: U.F.R. Informatique, Université Toulouse I
Place Anatole France, 31042 Toulouse Cedex, France

Abstract. This paper presents a survey of three formalisms that are used for modelling the dialogue of user-driven interfaces: state diagrams, events and Petri nets. Petri nets are found to be the best suited formalism in this area, even if they lack structure. In order to address this problem, the usefulness of the object-oriented approach is discussed, and we present a formalism, called Petri Nets with Objects (PNO), that integrates both object-oriented and Petri nets approaches. A three-step method for building such models, consisting in defining the object classes, definiting the presentation and modelling the application's dialogue, is presented, and a detailed example illustrates the application of this method. Finally, we present an overview of the benefits that can be expected from the use of the PNO formalism in dialogue modelling.

1 Introduction

The state of the art in human-computer interaction is nowadays what is commonly known as event-driven, direct manipulation interfaces. The event-driven nature of that kind of interface puts the control of dialogue in the hands of the user, and makes its specification, validation and implementation very error-prone. Using such an interface, the user can hold multi-threaded dialogues including several control flows, what may raise subtle synchronisation or resource sharing problems. These characteristics of diaslogue in user-driven interfaces make them resemble real-time or reactive systems.
In classical interfaces, the control is always held by the application, which requests input from the user. The set of possible inputs at a given time is very small and the dialogue structure, which corresponds to the hierarchy of menus, is quite simple to manage. Designing such applications has been studied for a long time and both formalisms and design techniques have been involved in design methods such as USE [1] and AXIAL [2]. In USE the dialogue structure of the application is modelled by state diagrams while in AXIAL only the sequence of screens and the hierarchy of menus are modelled.
User-driven interfaces are reactive systems (by opposition to transformational ones [3]): they are passive regarding to their environment, and react to stimuli they receive by triggering internal operations. In a general way, the application's control is external, i.e. the application does not define its own sequence of procedures, but only replies to the invocations it receives. Moreover, the application never asks the user for input (this would imply a blocking dialogue) but for an imperative confirmation or when a supplementary parameter is absolutely needed. Of course, the designer must minimise that kind of interaction in order to respect the user-driven nature of the interface.

The difference between the control structure of an application featuring a conventional interface and a user-driven one is depicted in Figures 1 and 2.

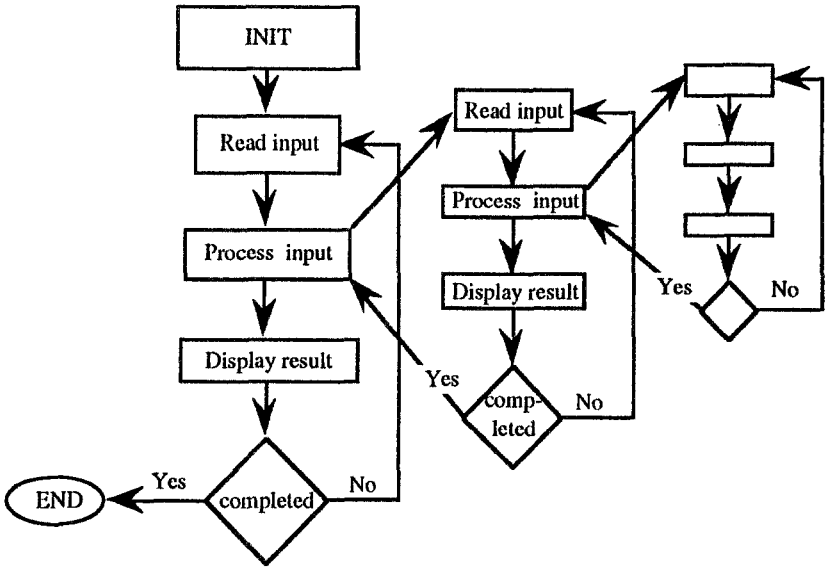

Fig. 1. Control structure of a conventional interactive application

Conventional applications feature a recursive transformational structure (obtain data, process data and display result) where the *process* part can invoke recursively the same structure (see Figure 1). On the opposite, a user-driven application features a flat structure built on a set of event-handlers procedures which do not invoke each other. A dedicated module, called the event manager (conceptually external to the application), manages an event queue, and ensures the interpretation of events by dispatching them to the event-handler which is able to process them. In most user interface management systems (UIMSes), the event loop and the event queue are transparent to the designer, who is only concerned by the design of the event-handlers and by the association of these procedures with the different widgets (see Figure 2).

Conventional applications feature a *centralised control* and *distributed inputs*. The control is located in the stack of functions call, and it is possible, by examining this stack, to determine the history of calls which lead to the current state of the application.

User-driven applications feature a *distributed control* and *centralised inputs*. Only the event loop catches inputs and all the event-handlers synchronise themselves by accessing in read and write on a set of variables defining the current state of the dialogue.

We can summarise the difference between conventional applications and user-driven ones by stating that for the former the control is based on the history of inputs, while for the latter the control is based on the state of the dialogue [4].

The hardest problem to be solved by the dialogue designer is precisely the modelling of the state of the application, which determines the evolution of the dialogue between the user and the application. Particularly, the designer must address the lack of control structure by using a formalism that allows to model both the states and the reactive nature of the application.

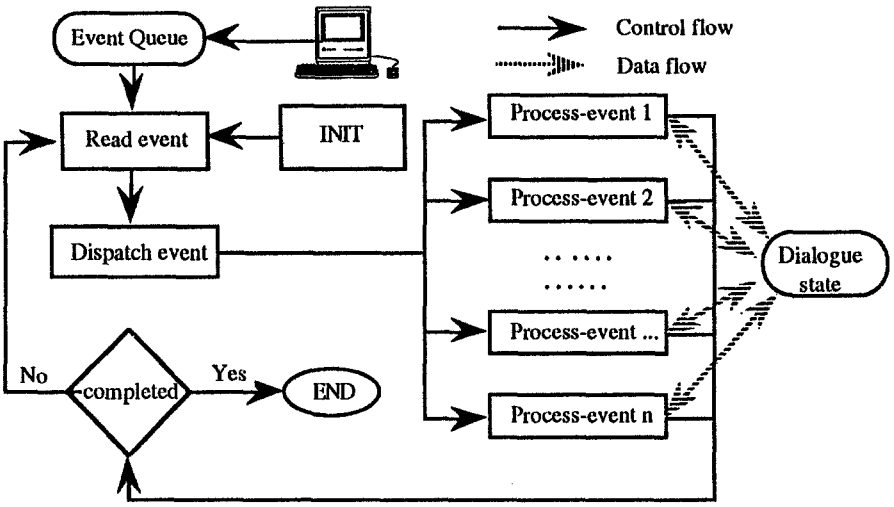

Fig. 2. Control structure of a user-driven application

In spite of the widespread use of such interfaces, the design methods or tools available today are surprisingly weak at handling their more important aspect: the design of the dialogue structure. According to the Seeheim architecture [5], this part of the interface is responsible for the syntactical analysis of the interaction language. In addition, [6 pp. 141] the dialogue structure must manage "the set of possible states and their relations". Various formalism such as context-free grammars [7, 8], state-transition diagrams and related formalisms [1, 9, 10] have already been investigated for modelling the dialogue structure in human-computer interaction. In the field of event-driven interfaces, these formalisms do not help much in modelling concurrent activities (particularly necessary in multi-threaded dialogues), do not provide structuring constructs, and do not allow the handling of data structures. Several extensions are available that tackle some of these drawbacks, but most often at the sacrifice of their formal definition. We quote Van Biljon [11] and say that "some investigation into extensions of finite state machines to enable them to provide this power led to the realisation that Petri nets already do this".
Petri nets have also been used for a long time in the area of user interface design [12, 13, 14]. Although they handle well concurrency and data structure (at least for the high-level models), Petri nets also used to lack structuring constructs. One of the aims of the Petri Nets with Objects (PNO) formalism, that we use for dialogue modelling, is precisely to allow the application of the object-oriented structuring constructs to high-level Petri nets.

Section 2 presents a survey of three formalisms that are used for modelling the dialogue of user-driven interfaces: state diagrams, events and Petri nets. Petri nets are found to be the best suited formalism in this area, even if they lack structure. In order to address this problem, the usefulness of the object-oriented approach is discussed. In section 3 we describe a complete example of a simple application featuring a user-driven interface. Section 4 details a three-step method based on the PNO formalism for building such an application. The last section (section 5) presents an overview of the benefits that can be expected from the use of this method.

2 Why do we Need Petri Nets and Objects

This part justifies the use of Petri nets for the modelling of reactive systems by comparing this approach with the state-based and event-based approaches, which are the most commonly used in user interface design. Then, the need for object-based modelling is considered.

2.1 State Versus Events in Modelling : the Benefits of Using Petri Nets

A reactive system is characterised by three components:

- the set of its possible states, denoted by $\mathbb{S}$, from which the current state is denoted by s_c and the initial state by s_0,
- the set of events to which it reacts, denoted by $\mathbb{E}$, from which the incoming event is denoted by e_i,
- the set of actions it can perform, denoted by $\mathbb{A}$.

The system's response to an event is to perform one of its actions, which may result in a change of state. The action performed depends on both the incoming event and the current state ($a = f(s_c,e_i)$), and the state reached (s_r) after the occurrence of the action depends on both the previous state and the incoming event ($s_r = g(s_c, e_i)$). f is called the *reaction* function while g is the *side-effect* function.

All the formalisms for the description of reactive systems aim at defining $\mathbb{S}$, $\mathbb{E}$, $\mathbb{A}$, f and g in a more or less explicit way, often providing a graphical notation designed to enhance the readability of models and generally putting the emphasis on one of the definition's components.

In the rest of this section, we use a toy example to illustrate the difference between event-based and state-based modelling, and show how Petri nets have the advantages of both.

The system considered as example is defined as follows:

- $\mathbb{S}$ = {S1, S2, S3, S4}, s_0 = S1
- $\mathbb{E}$ = {Ev1, Ev2},
- $\mathbb{A}$ = {A1, A2, A3, A4},
- f : $\mathbb{S}$ x $\mathbb{E}$ -> $\mathbb{A}$, such that
  f(S1,Ev1) = A1,
  f(S2,Ev1) = f(S3,Ev1) = A2,
  f(S4,Ev2) = f(S3,Ev2) = A3,
  f(S2,Ev2) = A4,
- g : $\mathbb{S}$ x $\mathbb{E}$ -> $\mathbb{S}$, such that
  g(S1,Ev1) = g(S3,Ev2) = S2,
  g(S2,Ev1) = g(S4,Ev2) = S3,
  g(S3,Ev1) = S4,
  g(S2,Ev2) = S1.

a. State-Based Modelling

In state-based modelling, the emphasis is put on the system's states, which are explicitly enumerated. The best suited formalisms for this approach are state diagrams and their extensions (statecharts [15], augmented transition networks [16], ...).

In state-based modelling, the system is represented by a quadruplet <S, O, T, s_0> where:

- S and s_0 are as stated above,
- O is the set of state changing operators,
- T is the transition function such that T : S x O --> S.

In finite state automata (FSA), O = $\mathbb{E}$ and T is equivalent to the side-effect function, which means that the reaction function is not modelled. Models derived from FSA take into account the reaction function by defining O such that O $\subseteq \mathbb{E} \times \mathbb{A}$.
The system considered as example may be modelled by a derived FSA as shown in fig. 3.

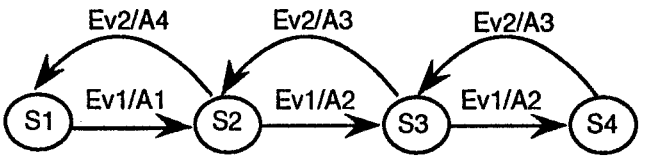

Fig. 3. State-based modelling of the toy example

The emphasis put on the system's states is highlighted by the graphical notation, since each state has a dedicated graphical representation. On the opposite, the elements of the sets $\mathbb{A}$ and $\mathbb{E}$ are duplicated in the model (e.g. the association Ev1/A2 is duplicated between states S2-S3 and S3-S4; association Ev2/A3 is also duplicated), and those sets may only be built by sorting out the inscriptions on the arcs. Moreover, if one wants to know from which state a given operation may occur, every state of the model must be studied. Likewise, the set of actions possibly triggered by a given event is not explicited, and must be built in the same way.
When there is much concurrency in the system, this leads to an automaton with a very large number of states, and the replication of events and actions hinders the readability and the conciseness of the model.

b. Event-Based Modelling

Most current UIMSes rely on an event-based approach where the focus is on the set of possible events to which the system has to respond. In practice, events and event-handlers are embedded in general purpose programming languages [17].
In event-based modelling, the system is represented by a quintuplet <V, $\mathbb{E}$, $\mathbb{A}$, O, v_0> where:

- $\mathbb{E}$ and $\mathbb{A}$ are the Event and Action sets as stated above,
- V is a set of state variables,
- v_0 is the initial value of the state variables,
- O is a set of operators such that O $\subseteq \mathbb{E} \times C \times \mathbb{A} \times SI$ where C is a set of conditions (boolean expressions on the state variables) and SI is a set of instructions consisting solely of affectations to state variables.

The system considered as example may be modelled with an event formalism, as shown in Figure 4.

V = {v}, v_0 = 1 and v : integer.
O = { <Ev1, v=1, A1, v:=2>,
<Ev1, v=2, A2, v:=3>,
<Ev1, v=3, A2, v:=4>,
<Ev2, v=4, A3, v:=3>,
<Ev2, v=3, A3, v:=2>,
<Ev2, v=2, A4, v:=1>}.

Fig. 4. The toy example modelled by an event formalism

The event-handlers as defined in the introduction may be deduced trivially from this description. There is one event-handler for each event Ev_i in E, built by selecting from O all the quadruplets in which Ev_i appears. The event-handler for Ev1 is shown in Figure 5.

```
Handler Ev1 is begin
        Case v of
                1 : A1; v:=2;
                2 : A2; v:=3;
                3 : A2; v:=4;
        Endcase
EndHandler;
```

Fig. 5. An example of an event handler

With such a formalism, the events appear very clearly. However, it is difficult to know which events may trigger a given action. This can only be achieved by searching for that action in all the event-handlers. Moreover, it is almost impossible to ensure that the action may actually be triggered, because the triggering depends on a state whose reachability is unknown. The set of all the possible states of the system is not explicit; we can only know that this set of states is a subset of the cartesian product of the state variables' domains. Of course, this problem does not appear in this toy example since there is only one state variable.
Finally, as in the state-based approach, actions are duplicated in the models.

c. Petri Nets-Based Modelling

Petri nets are often ranked amongst the state-based formalisms, which may be due to a hasty assimilation with finite state automata.
A Petri net is defined by a quintuplet <P, T, Pre, Post, M> where:

- P is the set of places,
- T is the set of transitions,
- Pre is the forward incidence function representing the input arcs of the transitions,
- Post is the backward incidence function representing the output arcs of the transitions,
- M is the distribution function (such that M : P -> $\mathbb{N}$) of tokens in the places, stating the number of tokens in each place of the net.

Due to space reasons a more complete definition of Petri nets is not given, but the interested reader may refer to [18, 19].

When modelling a reactive system with Petri nets, there is a need to represent the interface between the system being modelled and its environment. Two approaches may be followed to this end:

- consider a subset of T as interface transitions, which are triggered by the environment,
- consider a subset of P as interface places, in which the environment may deposit tokens.

The former is not suited to the modelling of reactive systems because it considers that the environment directly triggers actions in the system, whereas the latter allows to represent the fact that an incoming event may trigger different actions in the system. Moreover, in this second approach each event is directly modelled in the system by the deposit of a token in an interface place.
The system taken as example may be modelled by a Petri net as shown in Figure 6.

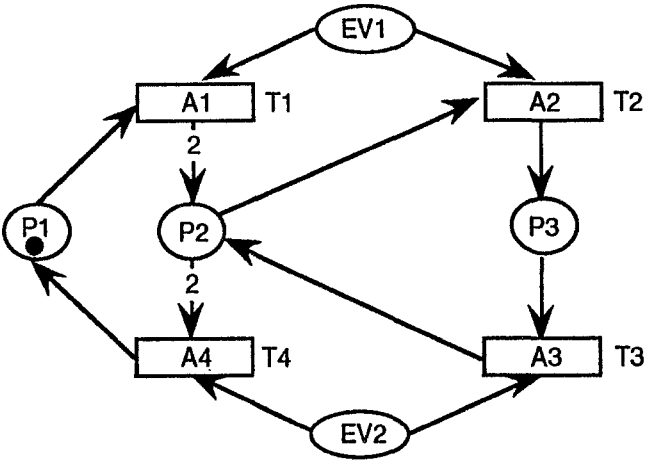

Fig. 6. Modelling by a Petri net of the toy example

Events are explicited in the model by places without incoming arcs called event-places. Contrarily to the state-based approach, events are not duplicated in the model. Actions are represented by transitions; they also appear explicitly and are not duplicated, contrarily to both state-based and event-based modelling.
The set of states is not directly shown, preventing the model from combinatory explosion. Instead, the structure of the set of states is modelled by the state-places (the ones which are not event-places). In the toy example there are three state-places - P1, P2 and P3 - but actually four different states: (1, 0, 0), (0, 2, 0), (0, 1, 1) and (0, 0, 2). However, Petri net theory allows for the easy calculation of the set of states, which is provided by the net's reachability graph.
The reaction function (associating events to actions) is clearly stated by the arcs from event-places to transitions. This function is explicit neither in the state-based nor in the event-based approach. Finally, the side-effect function is described by the arcs between the transitions and the state-places. Let's remark that Petri nets allow to model a reaction function which is not deterministic (and also provide mathematical tools to check if it is or not), but as expected, any side-effect function will be deterministic.

2.2 Benefits of the Object-Oriented Approach in the Area of HCI Modelling

Object-oriented modelling is particularly well suited to the area of reactive systems because it allows to consider such a system "from the outside", as a black box offering to its environment a set of operators and encapsulating a private state.
In reactive systems' modelling, the operators (often called methods) are mapped to the events, and the encapsulated state as well as the state changes may be modelled either by an automaton, by the set of variables of an event system or by a Petri net.
Moreover, the fact that most current UIMSes are object-based highlights the interest of following an object-oriented approach throughout the design of user-driven interfaces.

Building on the conclusions of this analysis, we have built a model, called Petri Nets with Objects (PNO), which combines the benefits of both objects and Petri nets [20]. The

example given in the next section illustrates how this formalism may be used in the modelling of the dialogue of user-driven interfaces.

3 An Example of PNO for the Modelling of Dialogue

First, we present the informal specification of the application's user interface. Then we define the window displayed to the user.

3.1 Informal Specifications

The example chosen to illustrate the use of the formalism is a fairly common one: an editor for tuples in a relational database table whose attributes are (Identifier, X, Y). This editor allows adding new tuples into the table, deleting tuples, selecting tuples from those already stored and changing their value. Of course, our goal is to provide a fully user-driven dialogue, as opposed to menu-driven interactions.
The overall look of the interface is shown in Figure 7. Three different areas can be distinguished in that window:

- The editing area in which the attributes of a selected tuple may be edited through the use of standard interface components (radio buttons, check box, simple-line entry field).
- A scrollable list (list box) shows the tuples of the table, presenting them by their distinctive attribute Identifier (a primary key). Items in this list may be selected by clicking on them with the mouse.
- A command zone in which database operations (addition, deletion, ...) may be launched by clicking on command pushbuttons.

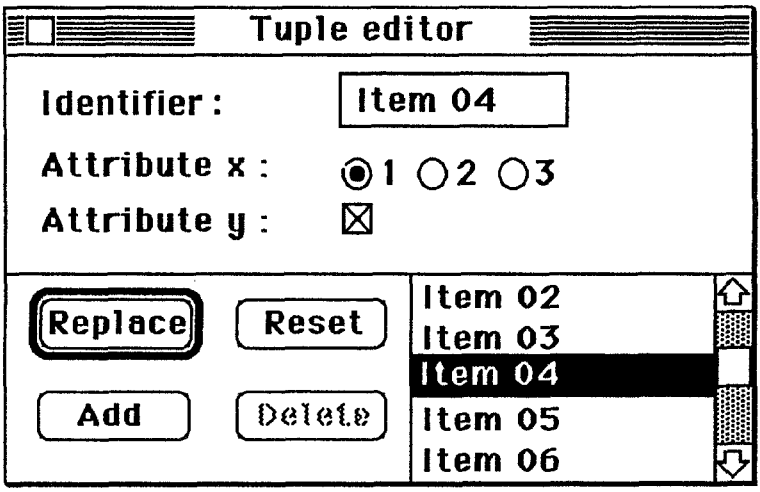

Fig. 7. Overall look of the interface window

The actions available to the user change through time and depend on the state of the dialogue. Those dialogue rules are expressed here informally. One of the goals of

modelling by PNOs is to make formal and non ambiguous such informal requirements expressed in natural language:

- It is forbidden to Select a tuple from the table when another one is being edited.
- It is forbidden to Quit the application while the user is editing a tuple. In any other case it must be possible to quit.
- It is forbidden to Delete a tuple whose value has been modified by the user.
- After modification of the current tuple, only the actions Add, Replace and Reset are available.
- The user must be able to act on the items of the edit area at any time.
- Only tuples that satisfy the integrity constraints may be added to the table.

3.2 Modelling of Dialogue Using PNOs

The tuple editor's dialogue is presented in Figure 8. It is modelled by a PNO: this provides a concise, yet formal and complete specification of the control structure of the application. The modelling power of the formalism allows to describe a lot of constraints, otherwise hard to describe in natural language. This PNO must be read in the following way:

•*nitialisation*

The initial marking of the net depends on the actual contents of the table at the time the window is opened. Figure 8 shows an initial marking: the places *list, selected* and *edited* are empty, and the place *default* contains the template for the first item to be edited. If the table was not empty, one tuple would be automatically selected while all the others would be in the place *list.*

Processing

In this initial state, only the two services **edit** and **add** (or transitions T1 and T2) may occur.

The occurrence of the **edit** service removes the template token from the place *default,* modifies its value and puts it back in the same place.

The occurrence of the **add** service depends on the precondition o.correct, which checks integrity constraints on the object, eventually producing a modal error dialogue. If the precondition is verified, the token is moved from the place *default* to the place *selected* and the tuple is stored in the table.

From then on, the table has one tuple. As the place *selected* is the only one holding a token, only the **delete** (transition T3) and **edit** (transition T4) services may occur. The occurrence of the **delete** service puts the PNO back in its initial state. The occurrence of the **edit** service results in the creation of a local copy of the tuple and the deposit of the original (o) and the copy (dup) in the place *edited.*

While the place *edited* holds a token, several services may occur:

- Modify the value of the copy by the occurrence of the service **edit** (T8).
- Replace the original by the copy through the service **replace** (T5).
- Cancel all changes by the occurrence of the service **reset** (T7); the copy is then deleted.
- **Add** the edited tuple to the table (T6); the added tuple becomes selected, while the original one becomes unselected.

If this cycle (**edit / add**) is performed a number of times, we will reach the state corresponding to Figure 7, where the place *selected* is empty, the place *edited* holds one token - a tuple with ident = "Item 04" -, and the place *list* contains at least tokens corresponding to tuples items 02, 03, 05, 06. This picture shows three activated pushbuttons, which correspond to the operations currently allowed on the table. The

active or inactive state of the pushbuttons is fully determined by the possible occurrence of the transitions they relate to in the PNO. For example, the delete button is not activated since place *selected* holds no token.

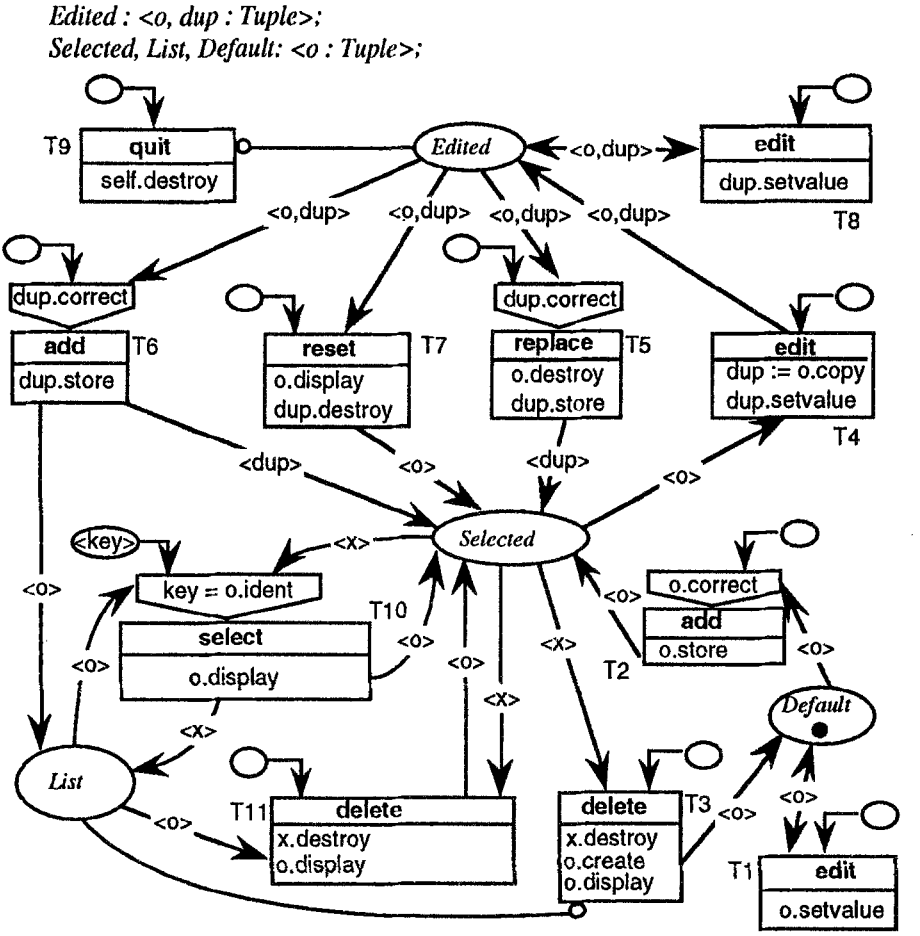

Fig. 8. The tuple editor's dialogue

The small places linked to transitions by squared arrows represent the event-places related to the environment. The occurrence of events are deposited as tokens in those places when the user acts on a widget of the presentation. The name of the event is not inscribed in the place in order to alleviate the diagram; instead it is inscribed in the transition(s) handling this event, since there is a one-to-one correspondence between events and the name of services. Several transitions bear the same name when an event is carried out in a different manner, according to the circumstance of its occurrence.

Figure 9 shows both the list of widgets (and events they may generate) associated to the tuple editor, and how the user can request a user service. All the widgets of the editing area

trigger the **edit** service since their purpose is only to inform the dialogue that an editing action has been performed.
All the transitions of the PNO presented in Figure 8 are related to event-places, which means that the action can only occur when the environment (in this case the user) triggers them. This is a characteristic of reactive systems which are fully controlled by their environment.

Widget	User's action	Event / Triggered service
PushButton Add	Click	Add
PushButton Delete	Click	Delete
PushButton Replace	Click	Replace
PushButton Reset	Click	Reset
PushButton Close_Box	Click	Quit
RadioButton 1	Click	Edit
RadioButton 2	Click	Edit
RadioButton 3	Click	Edit
EntryText	Any (Click, Keyboard, ...)	Edit
CheckBox	Click	Edit
ListBox	Click	Select

Fig. 9. Activation function

4 A Design Method of User Interface Dialogue

In this section we will present a way for building the PNO modelling the dialogue of a user-driven interface.
The highly reactive nature of a user-driven application has a great influence on the way of designing it. For that reason, the method presented here differs greatly from usual Petri nets design methods, which generally aim at modelling transformational applications.
The methodology for the design of user-driven interfaces, using the PNO formalism, is divided in three steps:

4.1 Define the Object Classes

- identify the objects handled by the application: in the example, there is only one class of objects, namely the tuples;
- define the operations that the objects must perform (or that may be applied to the object, according to its active or passive nature); in the example the operations which may be applied to a tuple are: create, copy, destroy, display, correct, setvalue, and store;
- define the objects' life cycles: the life cycle of an object states what are the operation sequences that the object may perform while keeping a consistent state and value; thus it gives the dynamic aspects of the object: how it behaves and how it may be used. A FSA is well suited for modelling an object's life cycle; the FSA's states are the states the object may be in. A transition from s_1 to s_2 labelled by an operation OP means that OP may be performed when the object is in state s_1, and that performing OP results in state s_2. A transition not labelled by any operation

corresponds to a change of state which may occur while no object's operation is performed (only passive objects which are fully controlled may have such transitions). The life cycle of a tuple shown in Figure 10 shows five states for an object: *not existent, created, in the list, selected* and *substitutable*. A tuple arrives in the *created* state by the create operation (cf. transition T3 of Figure 8) or by the copy operation (transition T4) and then it may be stored in the table. Once it is in the table, a tuple may become either *in the list* or *selected* or *substitutable* (a copy of the tuple is made whenever it goes from the *selected* to the *substitutable* state).

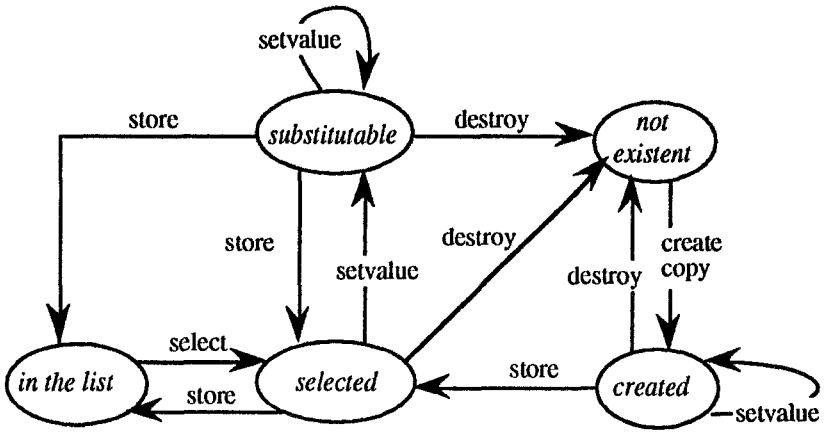

Fig. 10. Life cycle of a tuple modelled by a FSA

4.2 Define the Presentation

The design of the presentation is generally achieved using an UIMS and consists in drawing the layout of the different windows of the application. This step defines both the characteristics (type, title, size, etc.) of the windows and the list of widgets they contain. The virtual dialogue space of the application may then be automatically computed from the presentation. This virtual dialogue space is the set of all possible dialogues that could be expressed by a user provided with such a presentation if the availability of the window's widgets was not constrained by the state of the application. It consists of a service-transition having an event-place for each widget of each window featured in the presentation. Figure 11 presents the dialogue space of the tuple editor example, where each transition corresponds to a widget defined in the presentation shown in Figure 7.

4.3 Model the Application's Dialogue

The design of the application's dialogue consists of associating the virtual dialogue space together with the sets of states of the objects handled by the application (that is: what can be performed) while taking into account additional constraints expressed by the application's specifications (such constraints may be of various concerns: organisation, ergonomics, data integrity, etc.).

The set of states of the application is the synchronisation of the life cycles of the objects it handles, since the application's objects are processed in such a way that the constraints relating their respective value and state are fulfilled.

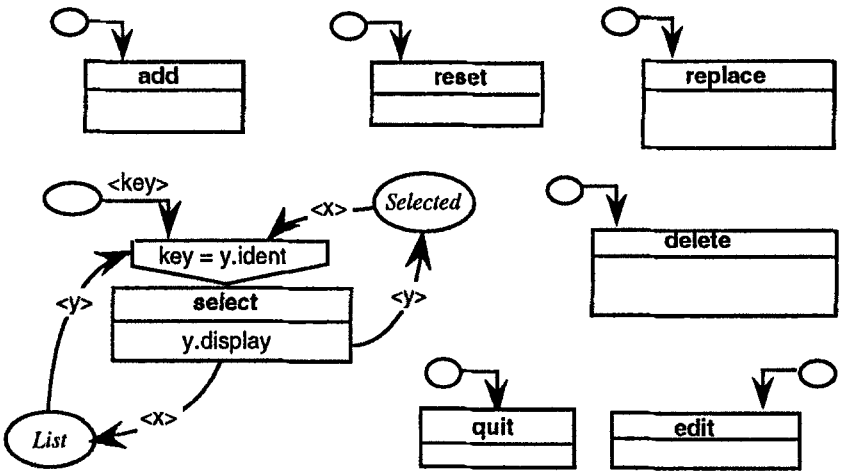

Fig. 11. Dialogue space of the Tuple-editor example

The tuple editor example features a single object class, Tuple. Any number of tuple occurrences may be simultaneously managed, with synchronisation constraints such as the following: when a *selected* tuple becomes *substitutable*, another tuple passes from *not existent* to *created* by the copy operation; if the *substitutable* tuple is destroyed or goes to the *in the list* state then the *created* one becomes *selected* by the store operation, and if the *substitutable* tuple comes back to the *selected* state then the *created* tuple is destroyed. Another synchronisation constraint is that whenever an *in the list* tuple becomes *selected* then a *selected* tuple becomes *in the list*. The synchronisation of the life cycles results in a PNO for which each place corresponds to one state of an object or to the gathering of several states of several objects (e.g. the *Edited* place of Figure 8 results from the merging of the *substitutable* and *created* states). The transitions of this PNO are related to these places according to state changes, and their actions are the operations labelling the transitions of the life cycles.

Now, the PNO modelling the dialogue results from the merging of the dialogue space and the life cycle FSAs of the different object classes. In addition to the services at the user's disposal, some transitions of the FSA are related to an event-place and become service-transitions. Taking into account the additional constraints of the application's specification, some transitions corresponding to unauthorised state changes are eliminated (although these state changes match the consistence of objects), some places may be added in order to restrict the transition enabling, and some transitions may be guarded by a precondition.

5 Expected Benefits of the Method

PNOs are a powerful formalism offering precious features such as validation, graphical representation, simulation and static analysis.

5.1 Design Validation

Using PNOs allows the designer to use mathematical properties of Petri nets for validation of the models. A lot of results are available in that area ([21, 17] among others) and we give some examples of results which may be used for event-driven interfaces.

- **Service availability:** Deadlocks are easy to avoid in a conventional, application-driven style of dialogue. However, in an event-driven dialogue, where the flow of control cannot be predicted because the user carries on concurrent dialogue flows with several parts of the application, the designer should have a way to ensure that whatever state the application is in, any command of the application will have a way to become available again through a given sequence of commands. In our design, this problem is directly connected to *liveness* in Petri net theory: for a given operation to be accessible, at least one of the transitions it relates to should be live in the PNO. Liveness of several transitions can be verified through static analysis of the models. For example, it can be proved that the **edit** service is always available to the user during the use of the application.
- **Limiting the number of occurrences:** It is frequently meaningful to specify limits on the number of occurrences in a model in order to represent physical limits of the real system. That kind of restriction is related to *boundedness* in Petri net theory. A place in a Petri net is k-bounded if there exists an integer k such that the number of tokens in this place cannot exceed k. For example, given the initial marking stated in Figure 8, place *Edited* is 1-bounded, thus establishing the desirable property that only one tuple can be acted upon at a time. This shows how bounds checking can help enforcing that the models have the right semantical properties.
- **Initial state reachability:** While modelling a system, it may be important to ensure that the initial state is always reachable. This is related to *analysis of transition sequences* in Petri net theory. Given an initial marking, a net is reinitialisable if there is a finite sequence of transition occurrences that can bring it back to its initial state. In our design, this means for the user the ability to reproduce his initial working environment, which might be desirable in most cases. In the example, this initial marking is a token in the *Default* place.
- **Verification of specific properties:** The techniques for the calculation of invariants [20, 22] allow to check that the model fulfils some properties expected of the real system. For instance, mutual exclusion may be proved in the PNO pictured in Figure 8: there is an S-invariant on places *Default*, *Selected* and *Edited*, ensuring that the total number of tuples held by these places cannot exceed their initial marking (in this case, one tuple, initially in place *Default*).

The design properties detailed above may be verified by static analysis on the PNO. That kind of validation allows evaluation of the quality of the model before implementation. The global formal validation of this example may be found in [23].

5.2 Prototyping an Application

Available UIMSes mainly focus on the definition of the external "look and feel" of the interface and provide only minimal definition of the dynamics of the dialogue. The prototyping possibilities are thus rather poor because they restrict the definition of dynamics to a mere sequencing of screens, like in former character-based applications. Actually, the most complicated part of the dynamics occurs inside each application window, as the sequence of windows is usually user-driven.

What UIMSes lack is a formal definition of the application's dialogue structure. The PNO formalism is formal and yet fully executable. Many tools, called "Petri nets interpreters" or "token-game players" and allowing the execution of Petri nets, have already been built [24].
Such an interpreter must be tightly integrated into an UIMS supporting the PNO formalism. Then both the internal dialogue state (provided by the PNO) and the external representation (provided by the presentation part) would be automatically supported.

5.3 Modelling Power of the PNO Formalism

Problems usually encountered in modelling are particularly due to the lack of modelling power of traditional formalisms. PNOs, with the numerous extensions (emission rules, emptying arcs, macro-places, macro-transitions) and the increase of modelling power they feature (preconditions, inhibitor arcs, object structuring), allow to model systems that would otherwise be hard to specify. Modelling the behaviour of an event-driven application with PNOs offers several benefits:

- A full description of the interface control structure, including causal dependencies between the application's services, i.e. dealing fully with the concurrency inside the application and between several applications.
- A single formalism may be used for different purposes: for drawing the application specifications, for validating the models, for prototyping them, and even for executing them with a PNO interpreter.
- The graphic (but still formal) modelling allows an easy communication between designers, programmers and users. Moreover, the descriptions are explicit as all the possible states and state-changing operations are clearly shown to the reader.
- Modelling the behaviour of a system may be done from the point of view of states (what are the reachable states and the transitions between them) or from the point of view of operations (what are the operations and their pre/post-conditions). Both of these points of view are useful for different purposes, and the PNO formalism enables to adopt one as well as the other.

6 Conclusion

This paper has presented a survey of three dialogue formalisms, state diagrams, events and Petri nets. By opposition to Green's survey [16] comparing grammars, transition networks and events, and affirming that events is the most powerful formalism, we have highlighted in our study the fact that Petri nets is the most concise, powerful and explicit formalism. Moreover, it is well known that Petri nets prevent the combinatorial explosion that is typical of state diagrams.
The advantages of using an object-oriented approach has also been discussed and the use of a formalism, called Petri Nets with Objects (featuring both object-oriented and Petri nets approaches), is then introduced.
An example of dialogue design using the PNO formalism has been fully presented and a three step method (define the object classes, define the presentation, and model the application's dialogue) for building such models has been introduced.
Finally, we have presented an overview of benefits which can be expected from a specification of dialogue using the PNO formalism.
The example given in this paper describes an application featuring only one window. Generally, user-driven applications will have many windows communicating with each

others. A methodology based on the principles presented here is fully described in [25]. This methodology encapsulates each window in an objet called Interactive Cooperative Object (ICO). An ICO is composed of attributes, methods, behaviour and presentation; the behaviour is modelled by a PNO such as described in the example of this paper, while the presentation consists of both the layout of the window and the activition function. Automatic implementation of ICO models by compilation has already been studied but has not been presented here for space reasons [23].
Actually our efforts are on the building of a development environment supporting our method. This environment will integrate a graphical presentation editor, a syntactic editor allowing the edition of PNOs, several analysis modules allowing to prove design properties of the models and a PNO interpreter acting as a run-time kernel.

References

1 A.I. Wasserman: Extending state/transition diagrams for the specification of human-computer interaction. IEEE Transactions on Software Engineering 11, 8 (August 1985), 699-713

2 P. Pellaumail: Guide d'utilisation d'AXIAL. Tomes 1 et 2. Editions d'organisation, 1986

3 A. Pnueli: Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. Lecture Notes in Computer Science 224. Springer-Verlag, Berlin, 1986, pp. 510-584

4 W. Cowan, M. Wein: State versus history in user interfaces. In: D. Diaper et al. (eds.): Human-Computer Interaction - INTERACT'90. North-Holland, 1990

5 G.E. Pfaff (ed.): Proceedings of IFIP/EG Workshop on User Interface Management Systems (November 1983, Seeheim, FRG). Springer-Verlag, Berlin,.1985

6 J. Coutaz: Interfaces homme-ordinateur : Conception et réalisation. Dunod Informatique, Paris, 1990

7 A.V. Aho, R. Sethi, J.D. Ullman: Compilers: principles, techniques and tools. Addison-Wesley, Reading, Mass., 1986

8 D.R. Olsen: Syngraph: A graphical user interface generator. Computer Graphics 17, 3 (July 1983), 43-50

9 D. Kieras, G. Polson: A generalized transition network representation for interactive systems. In: Proceedings of CHI'83, Human Factors in Computing Systems. 1983, pp. 103-106

10 R.J.K. Jacob: A specification language for direct-manipulation user interfaces. ACM Transactions on Graphics 5, 4 (October 1986), 283-317

11 W.R. Van Biljon: Extending Petri nets for specifying man-machine dialogues. International Journal of Man-Machine Studies 28 (1988), 437-455

12 M. Zizman: A System for Computerisation of Office Procedures. Ph.D. thesis, Warton School of Management, 1977

13 H. Oberquelle: Human-machine intercation and role/function/action-nets. In: W. Brauer, W. Reisig, G. Rosenberg (eds.): Petri nets: applications and relationships to other models of concurrency. Lecture Notes in Computer Science 254 & 255. Springer-Verlag, Berlin, 1986, pp. 171-190

14 B. Roudaud , V. Lavigne, O. Lagneau, E. Minor: SCENARIOO: A new generation UIMS. In: D. Diaper et al. (eds.): Human-Computer Interaction - INTERACT'90. North-Holland, 1990, pp. 607-612

15 D. Harel: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8, 231-274 (1987)
16 W.A. Wood: Transition network grammars for natural language analysis. Communications of the ACM 13, 10 (October 1970), 591-606
17 M. Green: A survey of three dialogue models. ACM Transactions on Graphics 5, 3 (July 1986), 244-275
18 J.L. Peterson: Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs, N.J., 1981
19 G.W. Brams: Réseaux de Petri : Théorie et pratique. Tome 1 : théorie et analyse ; Tome 2 : modélisation et applications. Masson, Paris, 1983
20 C. Sibertin-Blanc: High level Petri nets with data structure. In: 6th European Workshop on Petri Nets and Applications (June 1985, Espoo, Finland)
21 K. Lautenbauch: Linear algebraic techniques for place/transition nets. In: W. Brauer, W. Reisig, G. Rosenberg (eds.): Petri nets: applications and relationships to other models of concurrency. Lecture Notes in Computer Science 254 & 255. Springer-Verlag, Berlin, 1986, pp. 142-167
22 K. Jensen: Coloured Petri nets and the invariant method. In: A. Pagnoni and G. Rozenberg (eds.): Applications and Theory of Petri Nets, Informatik-Fachberichte 66. Springer-Verlag, Berlin, 1983
23 P. Palanque, C. Sibertin-Blanc and R. Bastide: Validation du dialogue par analyse d'une spécification fondée sur les réseaux de Petri. In Actes IHM'92 Quatrièmes journées sur l'ingénierie des interfaces homme-machine (30 nov., 1 et 2 déc. 1992, Paris). Telecom Paris, 1992, pp. 121-127
24 F. Feldbrugge, K. Jensen: Petri net tools overview. In: W. Brauer, W. Reisig, G. Rosenberg (eds.): Petri nets: applications and relationships to other models of concurrency. Lecture Notes in Computer Science 254 & 255. Springer-Verlag, Berlin, 1986, pp. 20-61
25 P. Palanque, Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur. Thèse de doctorat de l'Université Toulouse I (France), 1992
26 R. Bastide and P. Palanque: Petri nets with objects for the design, validation and prototyping of user-driven interfaces. In: D. Diaper et al. (eds.): Human-Computer Interaction - INTERACT'90. North-Holland, 1990, pp. 625-631