Implementation Techniques for Petri Net Based Specifications of Human-Computer Dialogues

Rémi Bastide and Philippe Palanque

Laboratory for Information Science, Université Toulouse I,
Place Anatole France, F-31042 Toulouse Cedex, France
Phone: +33-61.63.35.88 – Fax: +33-61.63.37.98
E-mail: {bastide,palanque}@cict.fr
WWW: http://lis.univ-tlse1.fr/~bastide -
http://www.cenatls.cena.dgac.fr/~palanque/

Abstract

Modern window-based user interfaces are actually a special kind of reactive system, and Petri nets may be fruitfully used to design such user-computer dialogues. This paper describes two techniques allowing to produce an executable system from a Petri net based specification of dialogue, namely interpretation and compilation. We first describe the compiled solution, where the Petri net structure is translated into conventional algorithms and data structures that can be implemented into any conventional event-driven UIMS. We then detail the object-oriented software architecture of an environment based on the interpreted approach, where the net structure is preserved at run-time, and present an original algorithm for interpreting high-level Petri nets in an event-driven environment.

Keywords

User interface design, computer tools for nets, high-level Petri nets.

Introduction

State of the art user interfaces are developed nowadays in graphical, window-based and mouse-driven environments. Once a very tedious and error-prone task, the development of such user interfaces is now greatly aided by interactive interface construction tools. Although the software marketplace abounds in such commercial products, the aim of such UIMS is usually somewhat limited : most available products only deal with the external appearance of the interface (its presentation).

Usually, the software designer is able to choose the interaction components from a large palette (buttons, menus, checkboxes, etc. which we will from now on call interactors), to partition the user interface into several windows, to define the layout of the interactors in the windows and to set various cosmetic properties.

However, currently available tools usually provide no help in the design of the dynamic behaviour of the interface. That behaviour consists in specifying the various reactions of the system to user-triggered events, in stating in some way the sequence of user commands that the application is able to accept, and in designing the visual response performed by the application in answer to user actions. This kind of specifications is actually made rather difficult by the event-driven nature of those event-driven dialogues. In current tools, this specification is postponed until the actual implementation of the system, since the dynamic behaviour is only defined by associating event-handling procedures, written in some algorithmic programming language such as C, to the various events that the user is able to trigger.

Our research team has been advocating for the past few years the use of Petri nets for the design of the dynamics of event-driven interfaces. We have proposed such an approach at the specification and design level [Palanque94b, Palanque93c], have investigated the use of Petri net theory to provide formal correctness proofs on the behaviour of interactive systems [Palanque95], and have also applied Petri net analysis techniques for providing contextual help systems [Palanque93c].

The executable nature of Petri nets make them a good candidate for an actual development language for that kind of system. The present paper describes our current work in providing automatic generation of executable systems from our interface specification approach. We first describe the compiled solution, where the Petri net structure is translated into conventional algorithms and data structures that can be implemented into any conventional event-driven UIMS. We then detail the object-oriented software architecture of an environment based on the interpreted approach, where the net structure is preserved at run-time, and present an original algorithm for interpreting high-level Petri nets in an event-driven environment.

1 Event-Driven Programming

The vast majority of interactive applications are nowadays developed with the aid of so-called UIMS tools. Despite the great diversity of graphical systems, all of those tools rely on a common programming paradigm, called event-driven programming.

In that kind of user interface, any command may be triggered through the use of some graphical interactor (icon, button, menu), accessible to the user by direct manipulation. This type of interaction is characterized both by a great freedom of action and an good level of guidance for the user (any forbidden action is presented as a greyed out or otherwise inactivated interactor).

Such interactive applications may be ranked among reactive systems [Pnueli86] : They do not act as transformational black boxes providing a result according to a given input, but maintain an ongoing interaction with their environment (in that case, the user). W. Reisig [Reisig92] states that most reactive systems should be better termed as « interactive systems ». The reverse is also true : modern interactive

software do function like reactive systems, and thus deserve the same methodological treatment.

However, interactive applications differ from real-time, industrial reactive systems by two important points :

- Interactive applications are most often programmed in a non-preemptive way, where a given event-handler, while activated, retains control over the application without being interrupted. This gives rise to cooperative multitasking environments, where several dialogues may proceed at once, provided that each event handler relinquishes control to the event manager, which may then dispatch a pending event. Interactive applications are in that respect easier to program than « hard » real-time systems, since the programmer does not have to deal with interrupts, critical sections, semaphores and the like. Each event-handler may be considered like a critical section in itself.
- Ergonomic rules state that, in such applications, the inner state of the system must always be perceptible to the user, and that each user action must always provide a visible feedback. In that respect, event-driven user interfaces bring a new and difficult task to their designers : they must ensure that the external presentation always faithfully reflect the internal state, by accurately displaying information, or by activating/deactivating several interactors. Such a process is known as rendering.

2 Designing Event-driven Interfaces with Petri Nets

Petri nets very naturally come into play for the design of the Dialogue component of the Seeheim model. They allow for an easy description of complex, concurrent control structures, they offer several structuring constructs, and, for the high-level models, they cleanly integrate the data structure aspects by allowing tokens to hold structured data.

In our approach, we will consider that (as it is often the case with current development methods) the presentation component is handled by specialised tools of the UIMS category. Moreover, we will consider that the non-interactive application kernel is designed in an object-oriented approach. If this is not the case (for example, if the application kernel is a relational database) the Application interface component will provide the necessary object-oriented layer.

We have proposed a Petri-net based, object-oriented formalism called Interactive Cooperative Objects (ICO) dedicated to the design of interactive systems [Palanque93c]. The formal definition of the ICO formalism is rather lengthy, since it needs to cope with concepts borrowed both from the object oriented approach (classification, inheritance, polymorphism, dynamic instanciation and use relationship) and from the Petri nets theory. Therefore the presentation in this paper is informal and only limited to the Petri net related aspects, but the interested reader may refer to [Palanque93a] for more details.

ICOs use a high-level dialect of Petri nets, where tokens are objects in the sense of object-oriented languages. In this paper, we will use the C++ notation for the description of classes, since the current implementation is in C++, and that C++ syntax is used for the annotations of the nets.

The places in the nets are typed, stating the type of tokens they may receive. Any C++ type (built-in, class type or pointer) may be used, and the C++ type system may be used to provide polymorphism for the tokens. The arcs hold variables that allow to state the flow of objects in the net. The variables on the arcs act as formal parameters for the adjacent transition. The type of those variables is deduced from the type of the places they are connected to. The transitions feature an action part, which may create or delete objects or call methods on the objects denoted by the arc variables. Transitions also feature a precondition, a boolean expression of the input variables acting as a guard.

Such a Petri net, called the Object Control Structure (ObCS), is associated with each window in the interactive application.

The ObCS plays the role of the *Dialogue* component in the Seeheim model. The *Application interface* and *Application kernel* are modelled by the classes of the tokens flowing in the net. The *Presentation* component is made of a set of interactors (widgets) that may display and edit data (for example text entry fields or radio buttons), or trigger events of interest to the application (for example, menu items or buttons).

The communication between the *Dialogue* component and the *Application kernel* is thus described both by the flow of tokens in the net and by the calling of tokens methods in the transitions' actions.

The communication between the *Dialogue* component and the *Presentation* component is more complex to describe, since several aspects are to be taken into consideration :

- The *Presentation* component influences the dialogue through the occurrence of events. This occurrence is modelled in the ObCS by special places called event places. The *Presentation* component is able to deposit tokens in those event places after the occurrence of an event. A transition in the ObCS net may have at most one input event place. A transition with an input event place is called an event transition. The very notion of interface place is made necessary by the fact that a given incoming event may trigger different actions in the system, according to the system's inner state. This is modelled by two or more event transitions in the ObCS sharing a common event place. Those transitions are therefore in structural conflict, and this indeterminism has to be relieved by the structure of the ObCS.

- Conversely, the state of the *Dialogue* component (i.e., the marking of the ObCS net) influences the *Presentation* component : according to this state, several events may be disabled, and their associated interactor greyed out. This is de-

scribed by associating event transitions to one or several interactors in the presentation : when a transition is not fireable, all of its associated interactors are greyed out or disabled.

- Lastly, the state of the ObCS net must be displayed by the presentation. This is done by associating a rendering action to each place of the ObCS. Such actions may call methods of the tokens held in the place in order to display whatever information is appropriate.

The example chosen to illustrate the use of the formalism is a fairly common one: an editor for information about customers stored in a relational database table. This editor allows adding new customers into the database, deleting customers, selecting customers from those already stored and changing their values. Of course, our goal is to provide a fully user-driven dialogue, as opposed to a menu-driven one.

Figure 1. Presentation of the customer edition window

The overall look of the interface is shown in figure 1. Three different areas can be distinguished in that window:

1. The editing area, in which the attributes of a selected customer may be edited through the use of standard interface components (radio buttons, simple-line entry field).
2. A command zone in which database operations (creation, deletion, ...) may be launched by clicking on command push-buttons.
3. A scrollable list (list box) shows the customers in the table. Items in this list may be selected by clicking on them with the mouse.

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 the modelling is to make formal and non ambiguous such natural language informal requirements:

- It is forbidden to select a customer from the table when another one is being edited.
- It is forbidden to quit the application while the user is editing a customer. In any other case it must be possible to quit.
- It is forbidden to delete a customer whose value has been modified by the user.
- After a modification of the current customer, only the actions Add, Replace and Reset are available.
- The user must be able to act on the items of the editing area at any time.

The application kernel is modelled by a single class : class *Customer*. The declarations for that class (figure 2) feature a constructor, used to generate new instances. The code for this constructor should query the various interactors in the edit zone to gather the values for the new Customer's attributes. This code is not shown here for it is highly dependent on the graphical system providing the user interface.

```
class Customer {
public:
    Customer();                     // Constructor for the class
    ~Customer();                    // Destructor for the class
    void Render() const;// Display attributes in the window
protected:                          // Data structure of the object
    String ID;
    String Name;
    enum { Card, Check, Cash } Payment;
};
```

Figure 2. Excerpt from the C++ class Customer

The constructor should also take care of inserting the new instance in some kind of persistent storage, for example a database table. Conversely the destructor, called on object deletion, should take care of removing the instance from the persistent storage. Lastly, the class features a method called Render, whose purpose is to display the values of the instance's attributes in the window. The ObCS for the dialogue is shown in figure 3. The event places are greyed out, and all of the transitions are event transitions. The interactor associated to each transition is apparent from the transition's label (e.g., the push-button Add is associated to the transition labelled Add) except for a few cases :

- The transition labelled Select is associated with the selection of a new element in the list box. This action is considered to deposit a pointer to the selected customer in the transition's input event place.
- The three transitions labelled Edit are associated with any of the interactors in the editing area. Any modification in those interactors will deposit a token in the input event place of those transitions.

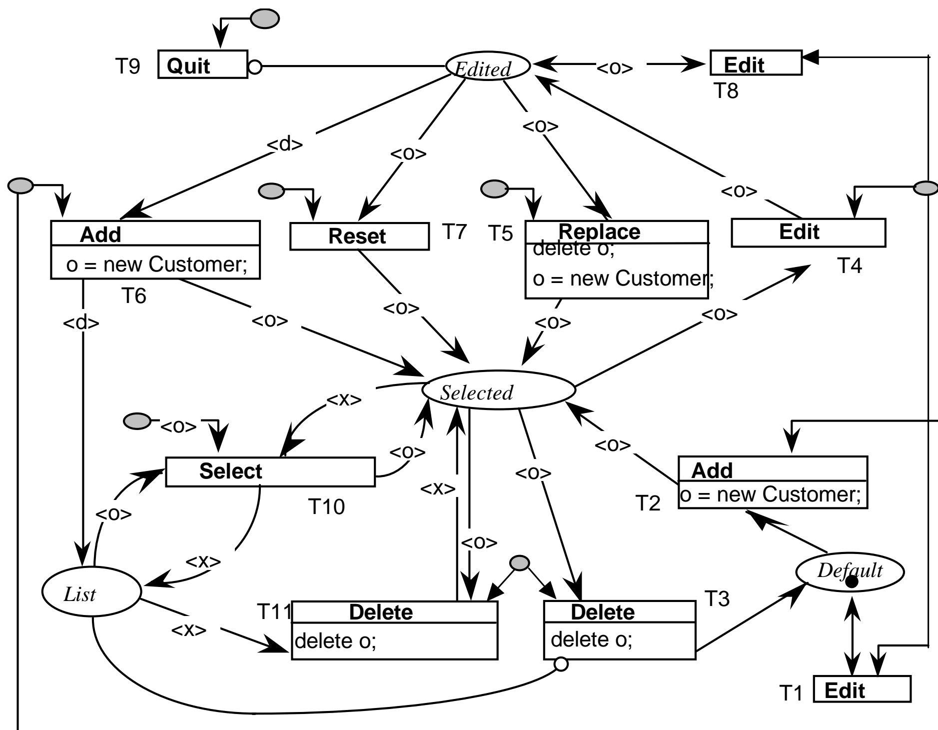

Figure 3. ObCS of the example dialogue

The places *List*, *Selected* and *Edited* are of type `<Customer *>`, i.e., they may hold pointers to instances of the class *Customer*. Place Default holds simple (untyped) tokens. Only the place *Selected* has a rendering action : it only calls the `Render` method on the *Customer* objects that enter that place.

From the initial marking pictured in figure 3, only the two events **Edit** and **Add** (or transitions T1 and T2) may occur.

The occurrence of the **Add** event creates a new Customer object from the values held by the interactors of the edition area. The newly created object is set in place *Selected*.

From now on, the table holds one customer. As the place *Selected* is the only one holding a token, only the **Edit** and **Delete** events may occur. The occurrence of the **Delete** event puts the net back in its initial state. The inhibitor arc between the place *List* and the transition T3 means that this transition may only occur if the place is empty, i.e. if the customer to be deleted is the last in the list. The occurrence of the **Edit** event transfers the token from place *Selected* into the place *Edited*.

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

- Modify the values of the attributes in the editing area by the occurrence of the event **Edit**.

- Replace the original by the new values through the event **Replace**.
- Cancel all changes by the occurrence of the service **Reset** (the original values of the Customer token are redisplayed, through the rendering function of place *Selected*).
- **Add** the edited customer to the table; the added customer becomes selected, while the original one becomes unselected.

If this **edit** / **add** cycle is performed a number of times, we might reach the state where the place *Edited* is empty, the place *Selected* holds one token - a customer whose identifier value is "CS_001" -, and the place *List* contains at least tokens corresponding to the customers CS_001, CS_002, and CS_003 (figure 1). This picture shows three inactivated push-buttons, which correspond to the currently forbidden user operations on the database. The active or inactive state of the push-buttons is fully determined by the possible occurrence of the transitions they relate to in the ObCS. For example, the Add button is not activated, since place *Edited* holds no token.

3 The Compiled Solution

The process is divided in two main stages: The first one aims at transforming the ObCS into several intermediate representations, while the second aims at producing the code of the application.

The first stage of the automatic code generation process is the transformation of the ObCS into an augmented transition network. The second stage processes the state-transition matrix, which is an equivalent description of the ATN. This matrix is correlated with the activation function, which relates the widgets to the actions to be performed. From these two components, the generation of the event-handlers for the widgets is quite simple, and essentially follows the process described in [Green86].

3.1 Transformation of the ObCS into an ATN

The techniques to calculate an ATN from a Petri net based description have been extensively studied [Wood70, Peterson81].

3.1.1 Calculation of the Marking Tree

The **marking tree** of a Petri net provided with an initial marking explicitly details the set of reachable states from this initial marking, as well as the sequences of transitions needed to reach those states. Each node in this tree represents a reachable marking of the net, and each arc is labelled with the name of the transition which causes the corresponding change to the marking. In many cases, the set of reachable markings is infinite, and the marking tree is thus also infinite. This infinite tree may be reduced to a finite structure called the **covering tree** of the net.

3.1.2 Calculation of the Marking Graph

The marking graph of a Petri net is a state transition diagram whose behaviour is strictly equivalent to that of the marked Petri net. The marking graph is easily deduced from the marking tree. The nodes of the marking tree which are associated to an identical marking are collapsed into a single node. Each node of the marking graph corresponds to a state of the dialogue. The marking graph is usually used to prove initial marking dependent properties of the net.

3.1.3 Calculation of the ATN

The marking graph automatically produced from the ObCS of an ICO cannot be represented by a finite state automaton, but it can be by an Augmented Transition Network (ATN) [Wood70].

In the graphic representation of an ATN, states are depicted by ellipses (initial state being thick lined) and transitions by arcs. The arc of a transition is labelled by: the service / the assignments, if any/ the preconditions, if any, as shown in figure 4.

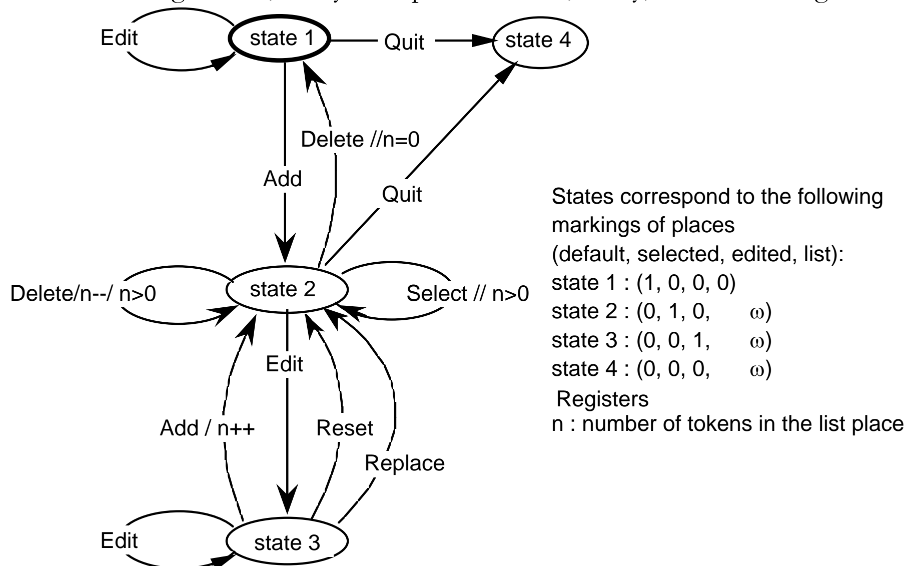

Figure 4. The ATN of the Editor

An ATN is essentially a finite state automaton provided with a set of registers which may be checked and modified when a changing of state occur. Thus, an ATN whose set of register is empty is a Finite State Automaton.

This ATN is built from the covering graph of the ObCS. Only the states from which a transition associated to a service may occur are kept, and there is one register for each unbounded place of the ObCS (a place for which the number of token has no upper limit) being an input place of such a transition.

Figure 4 shows the ATN of the Editor, and thus the command language at the user's disposal.

Although the ATN in figure 4 may appear simpler than the original ObCS, we are convinced that the ObCS is actually simpler to design than the ATN. In effect, for a complex dialogue, the most intricate parts to manage in the ATN construction are the definition and the handling of the registers.

These complex tasks may be dispensed of in the construction of the ObCS. Moreover, the Petri net description allows for an easy description of parallel dialogue and of synchronisation that are needed in multi-threaded application and are especially difficult to model in a sequential formalism such as ATN.

3.1.4 Construction of the State-Transition Matrix

An ATN may be described by a matrix, a representation which makes it easier to process by computer programs. This matrix is constructed in the following way:

- Each transition in the ATN is associated with a line in the matrix.
- Each state in the ATN is associated with a column in the matrix.
- Each cell in the matrix is divided into three components.

The first one represents the conditions imposed on the triggering of the transition. These conditions may come from preconditions in the original ObCS transition, or may concern the value of one of the ATN registers.

The second component of a cell represents the action to be performed when the transition occurs. This action is deduced both on the action in the original ObCS transition and on the modifications to be applied to the value of the ATN registers.

The third component describes the state reached after the occurrence of the transition.

3.1.5 Construction of a State-Service Matrix

In the state-transition matrix each line concerns one transition. As it is possible for a service to be related to several transitions it is possible for the matrix to contain several lines related to the same service. For example, the service Add is associated to the transitions T2 and T4 (see figure 3). The state-service matrix is constructed by merging all the lines related to a same service into one single line.

3.2 Code Generation

The steps we have described so far are independent of any given UIMS. Of course the details of the final step, which is the actual code generation, depend heavily on the UIMS at hand and on the Application Programming Interface (API) it supports.

The activation function is used to generate the part of the application code that is aimed to dispatch the incoming events to the right event handlers.

In some UIMSs (such as the C language interface to the MS-Windows toolkit), this is done by explicitly generating a complex switch statement, where the first dispatching is done according to the identifier of the widget which has received the event, and the second dispatching is done according to the type of event received.

With higher level APIs, this dispatching is often hidden to the programmer, and implemented with more powerful language constructs.

This may be done for example by associating a widget identifier to a virtual member function in a class representing the window (such as in the Borland C++ ObjectWindows API), or the dispatching process may be at the very basis of the programming environment (such as in Microsoft Visual Basic), and thus totally transparent to the programmer.

In any case, the activation function holds sufficient information to automatically generate the dispatching code.

From the components that have been produced so far, its possible to generate the code of the application.

3.2.1 Production of the Procedures Associated to the Services

A call-back procedure is automatically generated for each service. All the procedures to be generated have the same framework : a procedure is basically a switch structure according to the set of possible values for the state variable (corresponding to the columns of the state-service matrix).

Each switch is filled in with the contents of corresponding cell of the state-service matrix. Each branch of the switch will consist in four parts, the first three of which are directly extracted from the sub-cells of the corresponding cell in the matrix.

The first one is a pre-condition test, the second part holds the semantic action, and the third one sets the state reached after the occurrence of the service.

The fourth part of the branch corresponds to the visual feedback of the newly reached state. This part results in visually showing which user actions are enabled in the newly reached state. The necessary enabling and disabling actions are calculated from the state-service matrix and the activation function.

The services for which the cell corresponding to the new state is empty have all their associated widget disabled.

As an example, the call-back procedure associated to the add service is described in figure 5 and clearly shows the four parts.

```
Call-back procedure ADD;
Switch (CurrentState) { // test of the state variable
    case state1 :
            // no pre-condition to test
            // semantic action
            o.add  // add the tuple o to the table
            // state changing
            CurrentSate = State2 // change the current state
            // feedback of the commands available in the new state
            disable(PushButtonAdd)
            disable(PushButtonReset)
            disable(PushButtonReplace)
            enable(PushButtonClose_Box)
            enable(PushButtonDelete)
            enable(PushButtonListBox)
    case state2 :
            // no action
    case state3 :
            o.add  // add the tuple o to the table
            CurrentSate = State2 // change the current state
            n++  // increment the number of tuples in the table
            // show the commands available in new state
            disable(PushButtonAdd)
            disable(PushButtonReset)
            disable(PushButtonReplace)
            enable(PushButtonClose_Box)
            enable(PushButtonDelete)
            enable(PushButtonListBox)
}
```

Figure. 5. Callback procedure automatically generated for the service Add

3.2.2 Set-Up of the Event Handlers

The final step to produce a executable application is to associate an automatically generated procedure to a couple (widget, user-action). the details of this process depend completely on the API of development environment, and thus are not detailed here, but the process is usually straightforward. When the dispatching is done by the system (such as in Visual Basic), a empty procedure has only to be filled in with a call to the corresponding call-back procedure.

4 The Interpreted Solution

We have constructed a software environment to support the design of user interfaces where the dialogues are described by Petri nets in the approach described

above. This tool is integrated with a commercial UIMS which allows to generate the presentation part of the application in the Motif environment. The graphical representation of Petri nets make them a powerful debugging tool in the domain of user-interface design : the net may be displayed in a window along with the window which dialogue is being debugged, and the designer may then spot design flaws more easily by inspecting the marking of the ObCS net. At present, however, there is no possibility to interactively change and test some parts of the net, since its execution involves some C++ compilation and linking.

The kernel of the tool is a high-level Petri net interpreter developed in the C++ language. The architecture of this interpreter is original, and makes use of the powerful object-oriented features of C++ to achieve a high level of genericity.

A Petri net interpreter maintains a data structure isomorphous to the structure of the net it is playing : it has data structures for places, for transitions and for the incidence matrixes Pre and Post. The interpreter does its job by actually moving data structure representing tokens between data structure representing places.

In our case, the nets to be played differ from one another only by the nature of tokens that can be moved around (described by C++ classes), and also by the actions to be performed when firing a transition or when setting a token into a place (described by fragments of C++ code). This characteristic is very important for us, since we wish to be able to provide a user interface to any application written in C++.

We have therefore developed a generic C++ Petri net interpreter, made up of several interrelated classes : the generic Place, describing the basic data structure of a place, which allows it to store tokens ; The generic Transition, containing the code to determine if the transition is fireable and to fire it, etc.

To achieve the interpretation of an actual ObCS, several new C++ classes have to be generated from the structure of the net and of its component. For example, a transition with a special action will give rise to a subclass of the generic transition, with the overloading of one or several methods. The process necessary to generate a complete interactive application from the ObCS net is illustrated in figure 6.

Actually, no algorithmic code is generated by the translator, since all of that code is already contained in the generic classes. The code for the derived classes is mainly devoted to setting up data structures (such as the Pre and Post incidence matrix), and to insert cleanly the various elements of code given by the designer in the preconditions and actions of the transitions, and in the rendering actions of the places.

Most simple Petri nets interpreters are based on the basic structure given in figure 7. The most time-consuming step in that algorithm is step 4, and sophisticated data structure may be used to enhance that step, by avoiding unnecessary recomputations at each cycle. However, such an interpretation algorithm is not convenient in our approach, since this algorithm is preemptive. It does not fit with the basic structure of event-driven applications, for there is no place in this structure for

such a « never ending » control flow, which would prevent user events to be processed and dispatched.

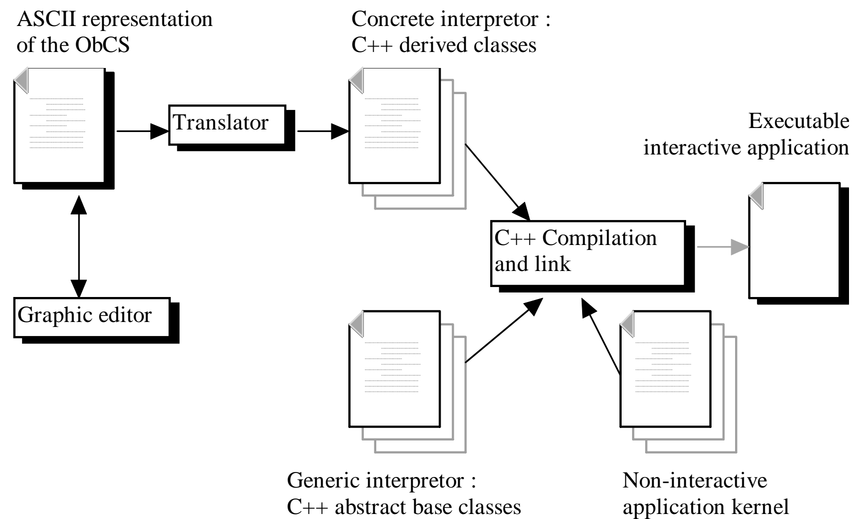

Figure 6. Architecture of the environment

Of course, one could think of implementing this algorithm as a separate process or thread, and of implementing the communication with the event-handlers as some kind of interrupt service.

We have chosen another approach, which avoids such complex constructs that might not be available or portable in every operating system. The solution chosen fits cleanly into the event-driven approach, and only uses event-driven constructs to achieve the same result.

set up the net structure
set up the initial marking
repeat
    search for *t*, a transition enabled by the current marking
    if *t* can be found **then**
        fire *t*, modifying the current marking
    end if
until *t* cannot be found

Figure 7. Basic algorithm of a Petri net interpreter

The basic idea is to associate with a dedicated event type the code necessary to process only one cycle of the loop described in figure 7. The program must then ensure that this event is triggered each time a transition might be fireable in the ObCS net.

The implementation of a Petri net interpreter in a purely event-driven system thus requires some primitives from the supporting environment :

- The ability to register new, « application defined » types of events, beyond those initially supported by the system. We will call this primitive `Register Event`.
- The ability to trigger the occurrence of a given event under the program's control. The event is inserted in the event queue, and later processed by its event-handler as though it had been triggered by an external action. We will call this primitive `PostEvent`.

Those primitives are actually quite common, and are present in one form or another in any UIMS we have had access to.

The basic algorithms for implementing a Petri net interpreter in an event-driven fashion is divided in three procedures : an initialization part, to be called in the main procedure of the program, a event-handler procedure whose role is to execute a single cycle of the interpretation loop, and a framework of code to be associated with any user-triggered event.

> set up the net structure
> set up the initial marking of the **interaction** net
> RegisterEvent(*one_more_try*)
> associate the event-handler *one_step* to the event *one_more_try*
> provide the rendering of the initial state
> PostEvent(*one_more_try*)
> activate the main event loop

Figure 8. Initialization procedure of the event-driven interpreter

The initialization procedure (figure 8) has to set up the various data structures necessary to represent the ObCS net. In the following algorithms, we will distinguish between what we call the **interaction net**, (i.e., the complete ObCS including its event places) and the **internal net** (The ObCS where all event places and their outgoing arcs are removed).

> search for *t*, a transition enabled by the marking of the **interaction** net
> **if** *t* can be found **then**
>   fire *t*, modifying the current marking
>   provide rendering according to the new marking
>   post_event(*one_more_try*)
> **end if**

Figure 9. One_step event-handler procedure

The initialization procedure registers a new event type (called *one_more_try*). This event is to be triggered when one loop through the interpretation code has to be performed.

The interpretation process is started in the initialization procedure by posting the event *one_more_try*. The event handler to be called on each occurrence of the *one_more_try* event is called *one_step*. This procedure is given in figure 9.

The procedure tries to find a fireable transition, and, if found, posts a new *one_more_try* event to make sure that any other fireable transition will be found when the event is processed.

parameters : *it* : Interactor, *ev* : Event,
create a new token *tok* according to *ev* attributes
set *tok* in the event place associated with *it*
post_event(*one_more_try*)

Figure 10. Framework for the event-handler associated to each interactor

The code framework to be associated to each interactor is given in figure 10. When an event triggered by an interactor occurs, the procedure computes a new token, and sets its into the event place associated with the interactor. A *one_more _try* event is now posted, since this new token may make some other transition fireable in the ObCS net.

As is apparent from the algorithm presented above (figures 8, 9 and 10), all the preemptive control structures in the interpreter have been replaced by a purely event-driven code.

The basic principle is that the *one_step* event-handler will be called once after the initialization phase of the program (this call is triggered by the post_event clause in line 5 of figure 8), and will be called again each time it detects an activated transition (call triggered by the post_event clause in line 5 of figure 9).

This ensures that any activated transition will fire. In most cases, the net will quickly reach an « dead » state, where no transitions are activated1. The only thing that may trigger an evolution is then an external action, via one of the interface's interactors.

Only the active interactors (i.e., those associated with a user transition fireable in the internal net) may be triggered, and thus the triggering of an interactor will deposit one token in the ObCS net, which will allow at least its associated transition to fire (and maybe some other internal transitions, not associated with any interactor).

---

1 This may not always be the case, e.g., if the dialog features a « background task », modeled by a sequence of transitions that remains constantly enabled during the processing.

Conclusion : Compilation vs. Interpretation

We have presented how Petri nets integrate in the process of designing modern interactive software. Petri nets might be used only for the specification phase, allowing to state in a concise manner complete and non ambiguous requirements for the control structure of interactive systems. With the help of the two implementation techniques described here, Petri nets can be retained throughout the development process, until the development phase.

The two techniques presented above (compilation and interpretation) both aim at executing an ICO specification of Human-Computer dialogue. This end is however achieved by very different means.

Obviously, the compiled solution will be much more efficient in terms of execution speed. The interpreted solution is time consuming, since the task that consists in checking which transitions are enabled in the ObCS net is computationally intensive. This drawback must be weighted, however, by the fact that this computation occurs in the interval of time between user-generated events, which is large with regards to machine efficiency. The ObCS nets are object-structured, and remain usually very simple, addressing the usual complaint about Petri nets being unstructured. The interpretation process can thus be made efficient enough to provide response times compatible with user expectations.

An advantage of the interpreted solutions is that the net structure is preserved at run-time, thus allowing for debugging facilities (e.g. animating the net representation during user activity). Moreover, the fact that the net structure is available at run-time allows for run-time reasoning about user interaction in terms of the dialogue model itself. We have explored ways to provide contextual help from this representation, for example [Palanque93b]. With the interpreted solution, the ICO formalism is amenable to a « model-based UIMS » environment, where the interface model is preserved until run-time.

References

[Green86] Green, M., *A Survey of Three Dialogue Models*, ACM Transactions on Graphics, Vol 5, No. 3, July 1986, pp. 244-275.

[Palanque93a] Palanque, P., Bastide, R., Sibertin, C., Dourte, L., *Design of User-Driven Interfaces using Petri nets and Objects*, in Proceedings of 5th Conference on Advanced Information Systems Engineering CAISE'93 (Paris, June 1993), F. Bodart, C. Rolland, C. Cauvet (Eds.), Lecture Notes in Computer Science No. 685, Springer-Verlag, Berlin, 1993. http://www.cenatls.cena.dgac.fr/~palanque/Ps/caise93.ps.gz

[Palanque93b] Palanque, P., Bastide, R., *Contextual Help for Free with Formal Dialogue Design*, in Proceedings of British Conference on Human-Computer Interaction HCI'92 « People and Computers VIII », J.L. Alty, D. Diaper, S. Guest (Eds.),

Cambridge University Press, Cambridge, 1993. http://www.cenatls.cena.dgac.fr/~palanque/Ps/hciinter 93. ps.gz

[Palanque94a] Palanque, P., Bastide, R., *Formal specification of HCI for increasing software's ergonomics*, in Proceedings of ERGONOMICS'94 (Warwick, 19-22 April 1994). http://www.cenatls.cena.dgac.fr/~palanque/Ps/ergono94.ps.gz

[Palanque94b] Palanque, P., Bastide, R., *Petri Net based Design of User-Driven Interfaces using the Interactive Cooperative Objects Formalism,* in Proceedings of 1st Eurographics Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'94 (Bocca di Magra, 8-10 June 1994), Focus on Computer Graphics Series, Springer-Verlag, Berlin, 1995, pp. 383-400. http://www.cenatls.cena.dgac.fr/~palanque/Ps/dsvis94.ps.gz

[Palanque95] Palanque, P., Bastide, R., *Verification of an Interactive Software by Analysis of its Formal Specification*, in Proceedings of the 5th IFIP TC13 Conference on Human-Computer Interaction INTERACT'95, Lillehammer, 25-29 June 1995, K. Nordbyn, P.H. Helmersen, D.J. Gilmore and S.A. Arnesen (Eds.), Chapman & Hall, London, 1995, pp. 191-196. http://www.cenatls.cena.dgac.fr/~palanque/Ps/interico95.ps.gz

[Peterson81] Peterson, J.L., *Petri net theory and modeling of systems*, Prentice-hall. Englewood Cliffs, 1981.

[Pnueli86] Pnueli, A., *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*, Lecture Notes in Computer Science Vol. 224, Springer-Verlag, Berlin, 1986, pp. 510-584.

[Reisig92] Reisig, W., *Combining Petri Nets and Other Formal Methods*, in Proceedings of ATPN'92 (Sheffield, June 1992), Lecture Notes in Computer Science Vol. 616, Springer-Verlag, Berlin, 1992, pp. 24-44.

[Wood70] Wood, W.A., *Transition network grammars for natural language analysis*, Communications of the ACM, Vol. 13, No. 10, October 1970, pp. 591-606.