User Interface Prototyping based on UML Scenarios and High-level Petri Nets [1]

Mohammed Elkoutbi and Rudolf K. Keller

Département d'informatique et de recherche opérationnelle
Université de Montréal
C.P. 6128, succursale Centre-ville, Montréal, Québec H3C 3J7, Canada
voice: (514) 343-6782
fax: (514) 343-5834
e-mail: {elkoutbi, keller}@iro.umontreal.ca
http://www.iro.umontreal.ca/~{elkoutbi, keller}

Abstract: In this paper, we suggest a requirement engineering process that generates a user interface prototype from scenarios and yields a formal specification of the system in form of a high-level Petri net. Scenarios are acquired in the form of sequence diagrams as defined by the Unified Modeling Language (UML), and are enriched with user interface information. These diagrams are transformed into Petri net specifications and merged to obtain a global Petri net specification capturing the behavior of the entire system. From the global specification, a user interface prototype is generated and embedded in a user interface builder environment for further refinement. Based on end user feedback, the input scenarios and the user interface prototype may be iteratively refined. The result of the overall process is a specification consisting of a global Petri net, together with the generated and refined prototype of the user interface.

Keywords: User interface prototyping, scenario specification, high-level Petri net, Unified Modeling Language.

1 Introduction

Scenarios have been identified as an effective means for understanding requirements [16] and for analyzing human computer interaction [14]. A typical process for requirement engineering based on scenarios [7] has two main tasks. The first task consists of generating from scenarios specifications that describe system behavior. The second task concerns scenario validation with users by simulation and prototyping. These tasks remain tedious activities as long as they are not supported by automated tools.

[1] This work is supported by FCAR (Fonds pour la formation des chercheurs et l'aide à la recherche au Québec) and NSERC (National Sciences and Research Council of Canada).

For the purpose of validation in early development stages, rapid prototyping tools are commonly and widely used. Recently, many advances have been made in user interface (UI) prototyping tools like UI builders and UI management systems. Yet, the development of UIs is still time-consuming, since every UI object has to be created and laid out explicitly. Also, specifications of dialogue controls must be added by programming (for UI builders) or via a specialized language (for UI management systems).

This paper suggests an approach for requirements engineering that is based on the Unified Modeling Language (UML) and high-level Petri nets. The approach provides an iterative, four-step process with limited manual intervention for deriving a prototype of the UI from scenarios and for generating a formal specification of the system. As a first step in the process, the use case diagram of the system as defined by the UML is elaborated, and for each use case occurring in the diagram, scenarios are acquired in the form of UML sequence diagrams and enriched with UI information. In the second step, the use case diagram and all sequence diagrams are transformed into Colored Petri Nets (CPNs). In step three, the CPNs describing one particular use case are integrated into one single CPN, and the CPNs obtained in this way are linked with the CPN derived from the use case diagram to form a global CPN capturing the behavior of the entire system. Finally, in step four, a prototype of the UI of the system is generated from the global CPN and embedded in a UI builder environment for further refinement.

In our previous work, we have investigated and implemented the generation of UI prototypes from UML scenarios using exclusively the UML, most notably UML Statecharts [5, 13, 21, 22]. In this Statechart-based approach, Statecharts are used to integrate the UML scenarios and capture object and UI behavior. In the work presented in this paper, we decided to take a Petri-net based approach, with CPNs taking the role of UML Statecharts. We opted for Petri nets because of their strong support of concurrency, their ability to capture and simulate multiple copies of scenarios in the same specification, and for the wealth of available tools for analyzing, simulating, and verifying Petri nets. A comparison of the two approaches is provided in Section 5 of the paper.

In our approach, we aim to model separately the use case and the scenario levels. We also want to keep track of scenarios after their integration. Thus, we need a PN class that supports hierarchies as well as colors or objects to distinguish between scenarios in the resulting specification. We adopted Jensen’s definition of CPN [10] which is widely accepted and supported by the *designCPN* tool [3] for editing, simulating, and verifying CPNs. Object PNs could also have been used, but CPNs are largely sufficient for this work. In our current implementation, UI prototyping is embedded into the *Visual Café* environment [23] for further refinement.

Section 2 of this paper gives a brief overview of the UML diagrams relevant to our work and introduces a running example. In Section 3, the four activities leading from scenarios to executable UI prototypes are detailed. Section 4 reviews related work, and in Section 5, we discuss a number of issues related to the proposed approach. Section 6 concludes the paper and provides an outlook into future work.

2 Unified Modeling Language

The UML [19], which is emerging as a standard language for object-oriented modeling, provides a syntactic notation to describe all major views of a system using different kinds of diagrams. In this section, we discuss the three UML diagrams that are relevant for our work: Class diagram (ClassD), Use Case diagram (UsecaseD), and Sequence diagram (SequenceD). As a running example, we have chosen to study a part of an extended version of the Automatic Teller Machine (ATM) system described in [2].

2.1 Class diagram (ClassD)

The ClassD represents the static structure of the system. It identifies all the classes for a proposed system and specifies for each class its attributes, operations, and relationships to other objects. Relationships include inheritance, association, and aggregation. The ClassD is the central diagram of UML modeling. Figure 1 depicts the ClassD for the ATM system.

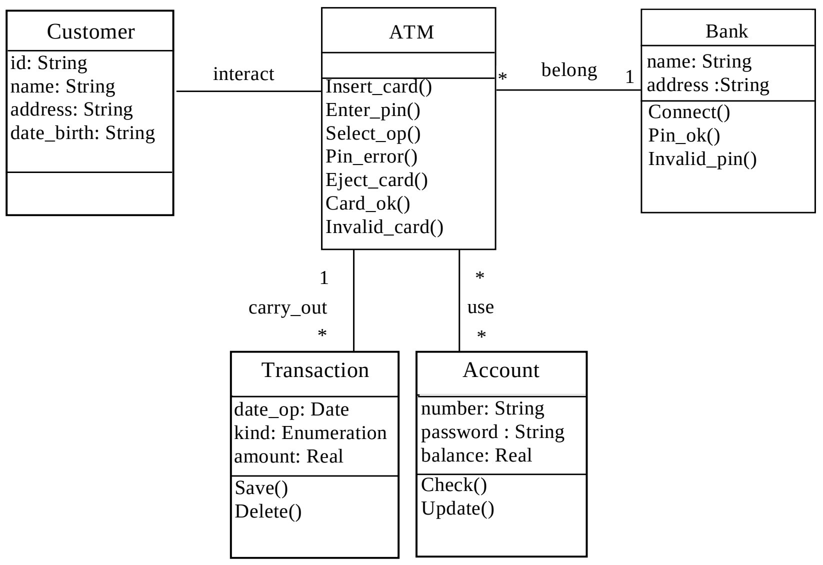


Figure 1: Class diagram of the ATM system.

2.2 Use Case Diagram (UsecaseD)

The UsecaseD is concerned with the interaction between the system and actors (objects outside the system that interact directly with it). It presents a collection of use cases and their corresponding external actors. A use case is a generic description of an entire transaction involving several objects of the system. Use cases are represented as ellipses, and actors are depicted as icons connected with solid lines to the use cases they interact with. One use case can call upon the services of another use case. Such a relation is called a *uses* relation and is represented by a directed solid line. Figure 2 shows as an example the UsecaseD corresponding to the ATM system. The *extends* relation, which is also defined in UsecaseDs, can be seen as a *uses* relation with an additional condition upon the call. A UsecaseD is helpful in visualizing the context of a system and the boundaries of the system's behavior. A given use case is typically characterized by multiple scenarios.

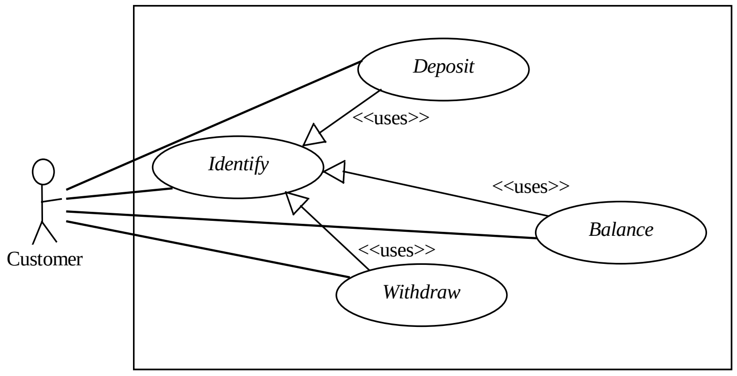


Figure 2: Use Case diagram of the ATM system.

2.3 Sequence Diagram (SequenceD)

A scenario shows a particular series of interactions among objects in a single execution (*instance)* of a use case. Scenarios can be viewed in two different ways: through SequenceDs or collaboration diagrams. Both types of diagrams rely on the same underlying semantics, and conversion from one to the other is possible. For our work, we chose to use SequenceDs for their simplicity and wide use.

A SequenceD shows the interactions among the objects participating in a scenario in temporal order. It depicts the objects by their lifelines and shows the messages they exchange in time sequence. However, it does not capture the associations among the objects. A SequenceD has two dimensions: the vertical dimension represents time, and the horizontal dimension represents the objects. Messages are shown as horizontal solid arrows from the lifeline of the object sender to the lifeline of the object receiver. A message may be guarded by a condition, annotated by iteration or concurrency information, and/or constrained by an expression. Constraints are used in our work to enrich messages with UI information.

Figures 3 and 4 depict two SequenceDs of the use case *Identify*. Figure 3 represents the scenario where the customer is correctly identified (*regularIdentify*), whereas Figure 4 shows the case where the customer entered an incorrect pin (*errorIdentify*).

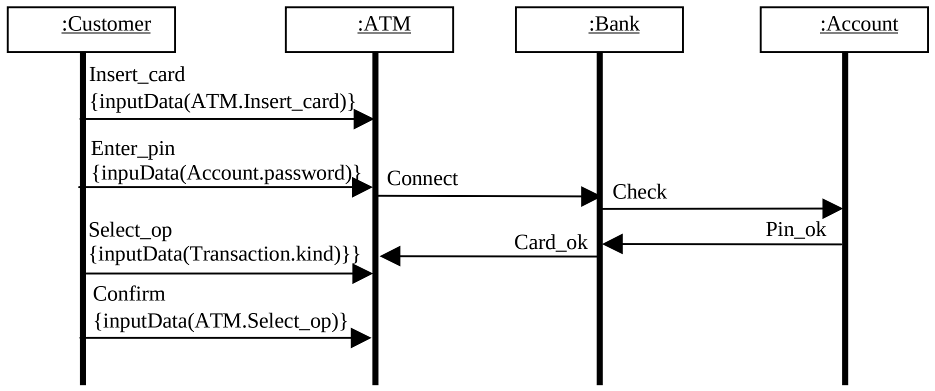

:Customer
:ATM
:Bank
:Account
Insert_card
{inputData(ATM.Insert_card)}
Enter_pin
{inpuData(Account.password)}
Connect
Check
Pin_ok
Card_ok
Select_op
{inputData(Transaction.kind)}}
Confirm
{inputData(ATM.Select_op)}

Figure 3: Scenario *regularIdentify* of the use case *Identify*.

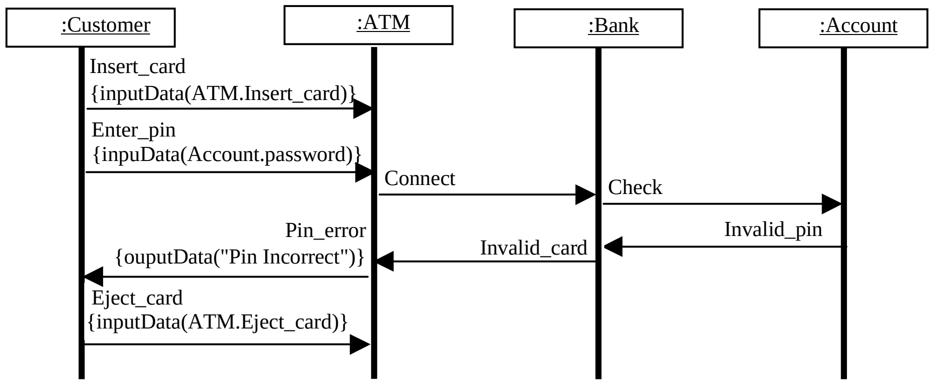

:Customer
:ATM
:Bank
:Account
Insert_card
{inputData(ATM.Insert_card)}
Enter_pin
{inpuData(Account.password)}
Connect
Check
Pin_error
{ouputData("Pin Incorrect")}
Invalid_card
Invalid_pin
Eject_card
{inputData(ATM.Eject_card)}

Figure 4: Scenario *errorIdentify* of the use case *Identify*.

Beyond the UML standard message constraints found in SequenceDs, we define the two additional constraints *inputData* and *outputData*. The *inputData* constraint indicates that the corresponding message holds input information from the user. The *outputData* constraint specifies that the corresponding message carries information for display. Both *inputData* and *outputData* constraints have a parameter that indicates the kind of user action. This parameter normally represents the dependency between the message and the elements of the underlying ClassD. It may be either a method name, one or several class attributes, or a string literal (see figures 3 and 4).

Once the analyst has specified the UI constraints of the messages in the SequenceD at hand, this information is used to determine the corresponding widgets that will appear in the UI prototype. Widget generation adheres to a list of rules, which is based on the terminology, heuristics and recommendations found in [8] and which includes the following eight items:

- A button widget is generated for an *inputData* constraint with a method as dependency, e.g., *Insert_card() {inputData(ATM.insert_card)}* in Figure 3.
- An enabled textfield widget is generated in case of an *inputData* constraint with a dependency to an attribute of type String, Real, or Integer, e.g., *Enter_pin() {inputData(Account.password)}* in Figure 3.

- A group of radio buttons widgets are generated in case of an *inputData* constraint with a dependency to an attribute of type Enumeration having a size less than or equal to 6, e.g., *Select_op() {inputData(Transaction.kind)}* in Figure 3.
- An enabled list widget is generated in case of an *inputData* constraint with a dependent attribute of type Enumeration having a size greater than 6 or with a dependent attribute of type collection.
- An enabled table widget is generated in case of an *inputData* constraint with multiple dependent attributes.
- A disabled textfield widget is generated for an *outputData* constraint with a dependency to an attribute of type String, Real, or Integer.
- A label widget is generated for an *outputData* constraint with no dependent attribute, e.g., *Pin_error() {outputData("Pin Incorrect")}* in Figure 4.
- A disabled list widget is generated in case of an *outputData* constraint with a dependent attribute of type Enumeration having a size greater than 6 or with a dependent attribute of type collection.
- A disabled table widget is generated in case of an *outputData* constraint with multiple dependent attributes.

3 Description of Approach

In this section, we detail the iterative process for deriving a system UI prototype from scenarios using the UML and CPNs. Figure 5 presents the sequence of activities involved in the proposed process.

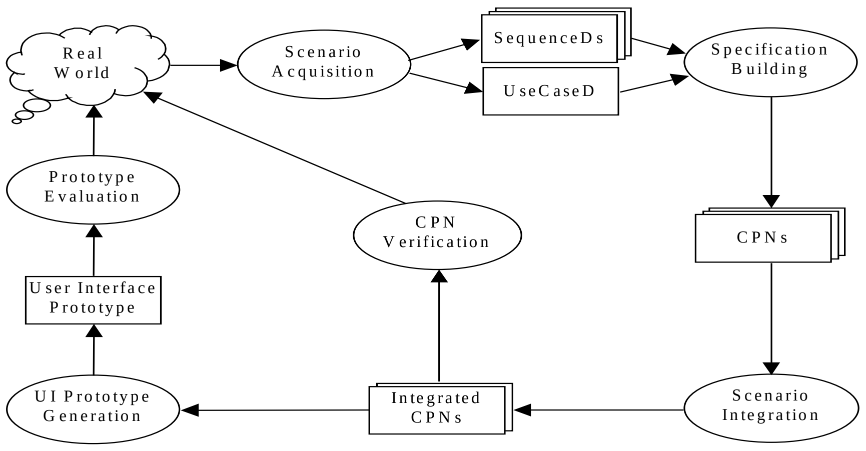


Figure 5: Activities of the proposed process.

In the *Scenario Acquisition* activity, the analyst elaborates the UsecaseD, and for each use case, he or she elaborates several SequenceDs corresponding to the scenarios of the use case at hand. The *Specification Building* activity consists of deriving CPNs from the acquired UsecaseD and SequenceDs. During *Scenario Integration,* CPNs corresponding to the same use case are iteratively merged to obtain an integrated CPN of the use case. Integrated CPNs serve as input to both the *CPN Verification* and the *UI Prototype Generation* activities. During *Prototype Evaluation*, the generated prototype is executed and evaluated by the end user. In the *CPN Verification* activity, existing algorithms can be used to check behavioral properties.

In the following subsections, we will discuss in detail the four activities of the UI prototyping process: scenario acquisition, specification building, scenario integration, and UI prototype generation. The CPN verification activity is discussed in [4].

3.1 Scenario acquisition

In this activity, the analyst elaborates the UsecaseD capturing the system functionalities, and for each use case, he or she acquires the corresponding scenarios in form of SequenceDs. For instance, the UsecaseD of the ATM system is shown in Figure 1, and two SequenceDs of the use case *Identify* are given in figures 3 and 4.

Scenarios of a given use case are classified by type and ordered by frequency of use. We have considered two types of scenarios: normal scenarios, which are executed in normal situations, and scenarios of exception executed in case of errors and abnormal situations. The frequency of use (or the frequency of execution) of a scenario is a number between 1 and 10 assigned by the analyst to indicate how often a given scenario is likely to occur. In our example, the use case *Identify* has one normal scenario (scenario *regularIdentify* with frequency 10) and a scenario of exception (scenario e*rrorIdentify* with frequency 5). This classification and ordering is used for the composition of UI blocks [5].

3.2 Specification building

This activity consists of deriving CPNs from both the acquired UsecaseD and all the SequenceDs. These derivations are explained below in the subsections *Use case specification* and *scenario specification*.

3.2.1 Use case specification

The CPN corresponding to the UsecaseD is derived by mapping use cases into places. The transition leading to one place (*Enter*) corresponds to the initiating action of the use case. A place *Begin* is always added to model the initial state of the system. After a use case execution, the system will return, via an *Exit* transition, back to its initial state for further use case executions. The place *Begin* may contain several tokens to model concurrent executions. Figure 6 depicts the CPN derived from the ATM system's UsecaseD (Figure 2).

In a UsecaseD, a use case can call upon the services of another one via the relation *uses*. This relation may have several meanings depending on the system being modeled. Consider a use case Uc_1 using a use case Uc_2. Figure 7(a) shows the general form of this relation. The use case Uc_1 is decomposed into three sub-use cases: Uc_{11} represents the part of Uc_1 executed before the call of Uc_2, Uc_{12} represents the part of Uc_1 that is concurrently executed with Uc_2, and Uc_{13} represents the part executed after the termination of *Uc*$_2$. Note that one or two of these three sub-use cases may be empty. The figures 7(a) through 7(g) depict the eight possible mappings.

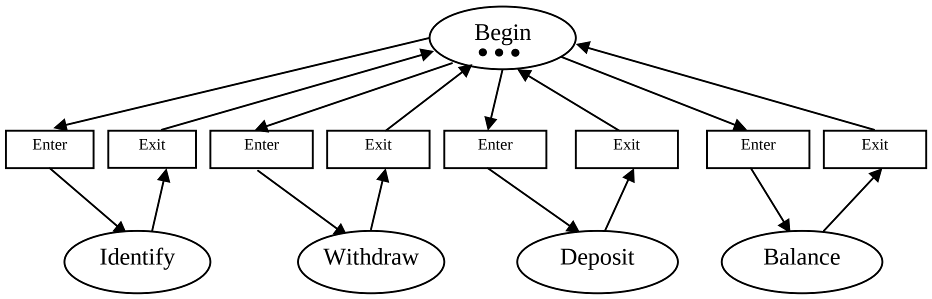


Figure 6: CPN derived from the UsecaseD of the ATM system.

A relation of type (g) between *Uc*$_1$ and *Uc*$_2$ means that *Uc*$_2$ precedes *Uc*$_1$. This implies that *Uc*$_1$ is not directly accessible from the place *Begin*. So the transitions from the place *Begin* to the place representing *Uc*$_1$ must be substituted for an *Enter* transition into *Uc*$_2$ and an *Enter* transition from *Uc*$_1$ into *Uc*$_2$. In the ATM system (Figure 1), all three *uses* relations are of type (g), and the initial CPN (Figure 7) must be updated accordingly (Figure 9(a)).

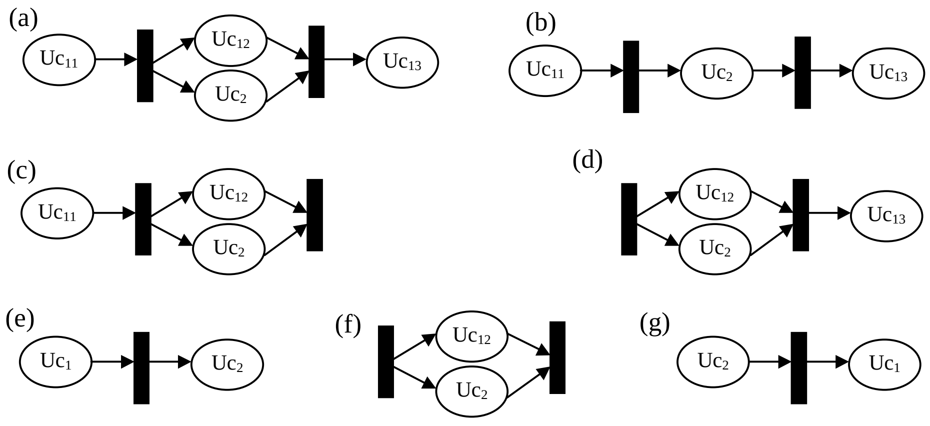
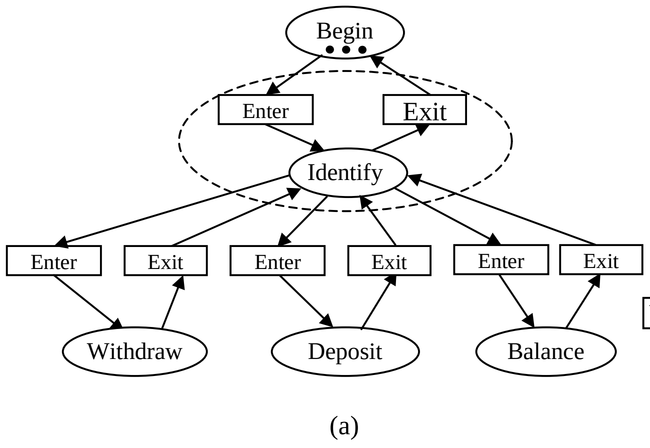


Figure 7: Possible mappings of the *uses* relation (*Uc*$_1$, *Uc*$_2$).

The designCPN tool, which we adopted in our work, allows for the refinement of transitions, but does not support the refinement of places. Therefore, in order to substitute the use cases, which are represented as places, for CPNs representing integrated scenarios (see Subsection 3.3), the CPN obtained after processing the *uses* relation (Figure 8(a)) requires adaptation: each subnet $Enter \rightarrow place_i \rightarrow Exit$ is substituted for a simple transition representing the use case underlying $place_i$ (cf. dashed ellipse in Figure 8(a)), and intermediary places such as *endIdentify* are inserted (see Figure 8(b)).



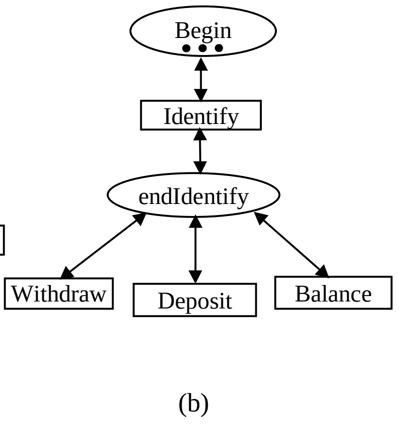


Figure 8: CPN of the UsecaseD of the ATM system after (a) processing the *uses* relation, and (b) adaptation to designCPN.

3.2.2 Scenario specification

For each scenario of a given use case, the analyst builds an associated table of object states. This table is directly obtained from the SequenceD of the scenario by following the exchange of messages from top to bottom and identifying the changes in object states caused by the messages. For example, tables 1 and 2 show the object state tables associated with the scenarios *regularIdentify* (Figure 3) and *errorIdentify* (Figure 4). In such tables, a scenario state is represented by the state vector of the objects participating in the scenario (column *Scenario state* in Table 1 and 2, resp.).

Objects / Messages	Customer	ATM	Bank	Account	Scenario state
Insert_card	Present	Card_in	void	void	S1={Present, Card_in, void, void}
Enter_pin	Present	Pin-entered	void	void	S2={Present, Pin_entered, void, void}
Connect	Present	Pin-entered	Connected	void	S3={Present, Pin_entered, Connected, void}
Check	Present	Pin-entered	Connected	Checked	S4={Present, Pin_entered, Connected, Checked}
Pin_ok	Present	Pin-entered	Valid_pin	Checked	S5={Present, Pin_entered, Valid_pin, Checked}
Card_ok	Present	Valid-card	Valid_pin	Checked	S6={Present, Vaild_card, Valid_pin, Checked}
Select_op	Present	Selection	Valid_pin	Checked	S7={Present, Selection, Valid_pin, Checked}
Confirm	Present	Confirmation	Valid_pin	Checked	S8={Present,Confirmation, Valid_pin, Checked}

Table 1: Object state table associated with the scenario *regularIdentify*.

Objects / Messages	Customer	ATM	Bank	Account	Scenario state
Insert_card	Present	Card_in	void	void	S1={Present, Card_in, void, void}
Enter_pin	Present	Pin-entered	void	void	S2={Present, Pin_entered, void, void}
Connect	Present	Pin-entered	Connected	void	S3={Present, Pin_entered, Connected}
Check	Present	Pin-entered	Connected	Checked	S4={Present, Pin_entered, Connected, Checked}
Invalid_pin	Present	Pin-entered	Invalid_pin	Checked	S9={Present, Pin_entered, Invalid_pin, Checked}
Invalid_card	Present	Invalid-card	Invalid_pin	Checked	S10={Present,Invalid_card, Invalid_pin, Checked}
Pin_error	Present	Invalid_pin	Invalid_pin	Checked	S11={Present, Invalid_pin, Invalid_pin, Checked}
Eject_card	Present	Idle	Invalid_pin	Checked	S12={Present, Idle, Invalid_pin, Checked}

Table 2: Object state table associated with the scenario *errorIdentify*.

From each object state table, a CPN is generated by transforming scenario states into places, and messages into transitions (see figures 9 and 10)[2]. Each scenario is assigned a distinct color, e.g., *rid* for the *regularIdentify* scenario, and *eid* for the *errorIdentify* scenario. All CPNs (scenarios) of the same use case will have the same initial place (state) which we call *B* in figures 9 and 10. This place will serve to link the integrated CPN (see below) with the CPN modeling the UsecaseD of the system (Figure 8(b)).

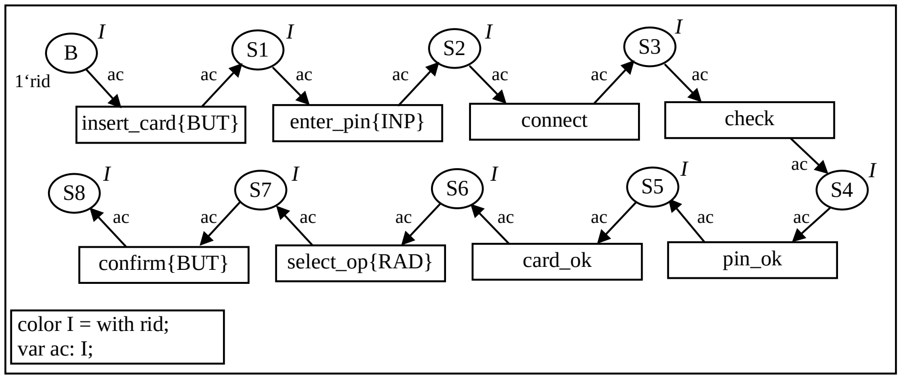


Figure 9: CPN corresponding to the *regularIdentify* scenario.

[2] For readability, we do not show screen dumps produced by designCPN, but replace them with CPNs redrawn by hand.

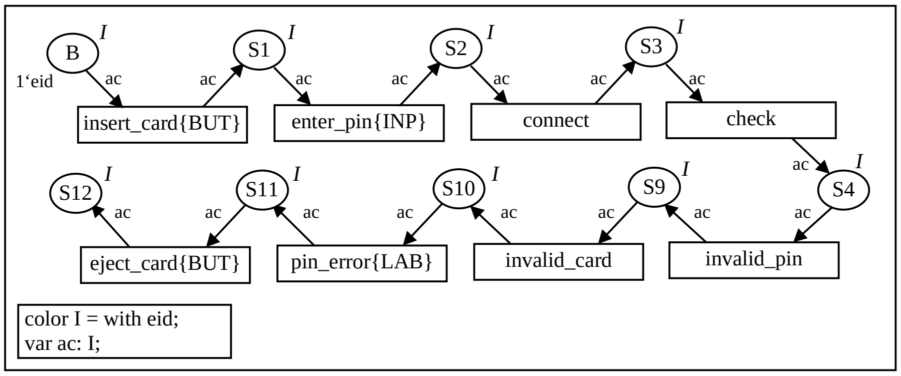


Figure 10: CPN corresponding to the *errorIdentify* scenario.

Note that in scenario specification, only the building of the object state tables is manual. The rest of the operation is fully automatic.

The next activity of the approach, scenario integration, requires as input serialized CPNs. Since designCPN uses SGML as interchange format and since conversion tools between SGML, XML, and Java are readily available, we decided to represent CPNs in XML. In our current implementation, we use the transformation scheme as depicted in Figure 11. Using the designCPN tool, the analyst will edit and then save the use case CPN and all scenario CPNs into the textual SGML format. Then, the *SX* tool [1] is used to do the conversion from SGML to rough XML (RXML), and the *XJParse* Java program [9] to convert RXML into XML.

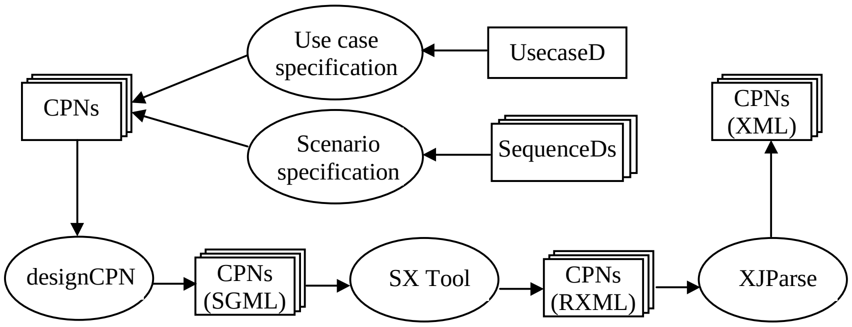


Figure 11: Generation of CPNs in XML format from UsecaseD and SequenceDs.

3.3 Scenario integration

In this activity, we aim to merge all CPNs corresponding to the scenarios of a use case Uc_i, in order to produce an integrated CPN modeling the behavior of the use case. Our algorithm is based on a preliminary version presented in [6]. It takes an incremental approach to integration. Given two scenarios with corresponding CPNs

CPN_1 and CPN_2, the algorithm merges all places in CPN_1 and CPN_2 having the same names. The merged places will have as color the union of the colors of the two scenarios. Then, the algorithm looks for transitions having the same input and output places in the two scenarios and merges them with an *OR* between their guard conditions. In the following, we describe the algorithm in pseudocode, using the “dot”-notation known from object-oriented languages.

```
Uc$_i$.IntergrateScenarios()
    scList = Uc$_i$.getListOfScenarios();
    // returns the list of scenarios of the use case Uc$_i$
    uc_cpn = getXML(scList[0]);
    // returns the XML file corresponding to the scenario scList[0].
    i=1;
    while (i < scList.size())
        sc_cpn = getXML(scList(i));
        sc_cpn = makeUniqueID( uc_cpn, sc_cpn);
        uc_cpn = merge(uc_cpn,sc_cpn);
        i = i + 1;
    end
end Uc$_i$.IntergrateScenarios
```

XML identifies each element in a CPN (place, transition, edge, etc.) by a distinct identifier (ID). Before integrating two scenarios, the method `makeUniqueID` checks if both the two input files `uc_cpn` and `sc_cpn` comprise distinct IDs. In case they share common IDs, `makeUniqueID` will modify the IDs of `sc_cpn` by adding the maximum ID of `uc_cpn` to all IDs of `sc_cpn`.

Merging (integrating) two scenarios whose CPNs have the colors *[sc_1]* and *[sc_2]*, respectively, will produce a CPN with the list *[sc_1,sc_2]* as color. The operation of merging follows the steps described below:

```
merge(uc_cpn,sc_cpn)
    uc_cpn.addPlaces(sc_cpn)
    // adds in uc_cpn places of sc_cpn that do not exist in uc_cpn
    for each t in sc_cpn.getListOfTransitions()
        t’ = uc_cpn.LookForTrans(t)
        // t’ is a transition of uc_cpn with •t=•t’ and t•=t’•
        if (t’ does not exist)
            uc_cpn.addtrans(t)
        endif
    end
    uc_cpn.addEdges(sc_cpn)
    // adds to uc_cpn edges of sc_cpn that do not exist in uc_cpn
    uc_cpn.mergeColors(sc_cpn)
    // calculates the new color of the integrated CPN (uc_cpn)
    uc_cpn.putColorsOnPlaces(sc_cpn)
    // all places of the net will have the merged color
    uc_cpn.putGuardOnTransitions(sc_cpn)
    // common transitions will be guarded by the merged color,
    // the others will be guarded by their original colors
    uc_cpn.putVariablesOnEdges(sc_cpn)
    // put on edges variables or token expressions
end merge
```

The result of applying this algorithm on the two scenarios of the use case *Identify* (figures 9 and 10) is shown in Figure 12.

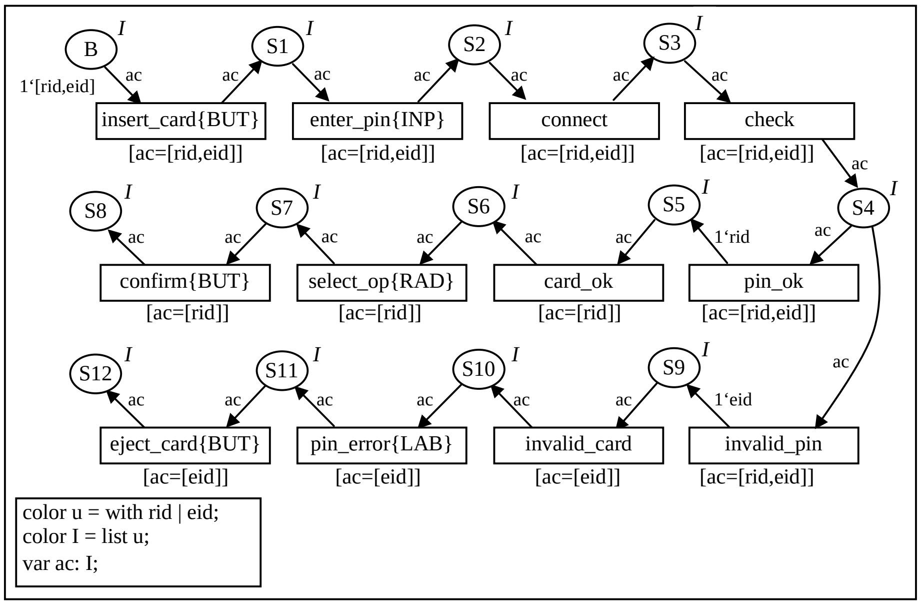


Figure 12: CPN of the use case *Identify* after merging the scenarios *regularIdentify* and *errorIdentify*.

When integrating several scenarios, the resulting specification captures in general not only the input scenarios, but perhaps even more. Figure 13 gives an example illustrating this issue: the resulting scenario *Sc* will capture the initial scenarios Sc_1 (T_1,T_2,T_3,T_4,T_5) and Sc_2 (T_1,T_6,T_7,T_4,T_9), as well as the two new scenarios (T_1,T_2,T_3,T_4,T_9) and (T_1,T_6,T_7,T_4,T_5). After integrating the two scenarios, the initial place *B* (see Figure 13) will be shared, yet we do not know which scenario will be executed, and neither the color of Sc_1 nor the color of Sc_2 can be assigned to *B*. This problem was described by Koskimies and Makinen [12], and we refer to it as *interleaving problem* [6].

To solve the interleaving problem, we introduce a *chameleon token*, i.e., a token that can take on several colors [6]. Using designCPN, a chameleon token is modeled by a list of colors. Upon visiting the places of the integrated net, it will be marked by the intersection of its colors and the colors of the place being visited. When the token passes to the place S_1, it keeps the composite color $[sc_1, sc_2]$, and if it passes from S_1 to S_2, its color changes to $[sc_1]$ and will remain unchanged for the rest of its journey, or if it passes from S_1 to S_6, its color changes to $[sc_2]$ and will remain unchanged for the rest of its journey.

Transitions that belong to only one of the scenarios Sc_1 and Sc_2 will be guarded by the color of their respective scenarios (see T_3,T_5,T_7, and T_9 in Figure 13(c)). But this is not the case for the transitions T_2 and T_6 which are required to transform tokens from the composite color (list of colors) to a single color. Therefore, they must be guarded by the list of colors. For transitions that are shared by the two scenarios Sc_1 and Sc_2, they will be guarded by the composite color (see T_1 in Figure 13(c)) or by a disjunction of single colors of scenarios (see T_4 in Figure 13(c)).

An integrated CPN corresponding to a given use case can be connected to the CPN derived from the UsecaseD through a transition that is appended to the place *B* of the integrated CPN. This transition transforms the uncolored tokens of the CPN of the UsecaseD to the composite color of the integrated CPN.

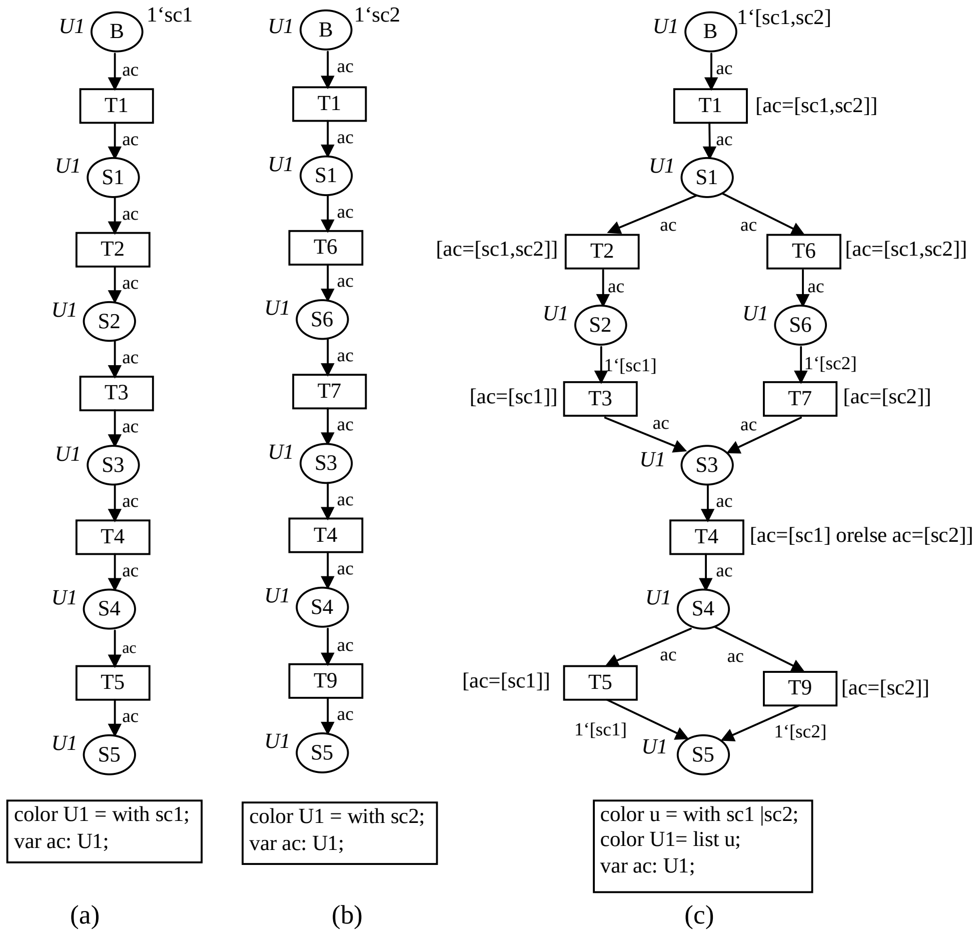


Figure 13: Interleaving problem of scenario integration: (a) scenario Sc_1, (b) scenario Sc_2 and (c) integrated scenario *Sc*.

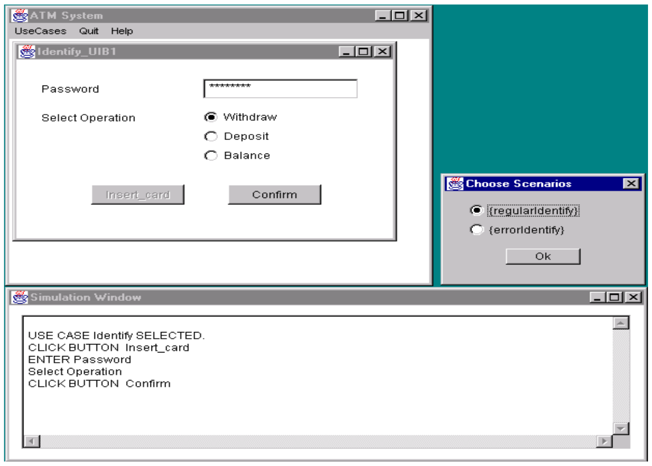
3.4 User interface prototype generation

In this activity, we derive from the CPN specifications a UI prototype of the system. The generated prototype is standalone and comprises a menu to switch between the different use cases. The various screens of the prototype represent the static aspect of the UI; the dynamic aspect of the UI, as captured in the CPN specifications, maps into the dialog control of the prototype. In our current implementation, prototypes are Java applications comprising each a number of frames and navigation functionality (see Figure 14).

The activity of prototype generation is composed of the following five operations which are described in detail in [6].

- Generating graph of transitions.
- Masking non-interactive transitions.
- Identifying UI blocks.
- Composing UI blocks.
- Generating frames from composed UI blocks.

These operations follow closely the corresponding operations in the Statechart-based approach [6], except for the operation of generating the graph of transitions. This operation consists of deriving a directed graph of transitions (GT) from the CPN of a given use case. Transitions of the CPN will represent the nodes of the GT, and edges will indicate the precedence of execution among transitions: if two transitions T_1 and T_2 are consecutively executed, there will be an edge between the nodes representing T_1 and T_2.
A GT has a list of nodes *nodeList*, a list of edges *edgeList*, and a list of initial nodes *initialNodeList* (entry nodes of the graph). The list of nodes *nodeList* of a GT is easily obtained since it corresponds to the transition list of the CPN at hand. The list of edges *edgeList* of a GT is obtained by linking the transitions in •p with the ones in p• for each place p in the CPN.



Password ********

Select Operation [✓ Withdraw] Deposit Balance

[✓ {regularIdentify}] {errorIdentify}

```
USE CASE Identify SELECTED.
CLICK BUTTON  Insert_card
ENTER Password
Select Operation
CLICK BUTTON  Confirm
```

Figure 14: Prototype execution of the ATM system.

To support prototype execution, a *Simulation Window* is generated (Figure 14, bottom window), as well as a dialog box to *Choose Scenarios* (Figure 14, middle-right window). For example, after selecting the use case *Identify* from the *UseCases* menu (Figure 14, top window), a message is displayed in the simulation window that confirms the use case selection and prompts the user to click the button *Insert_card*. When this button is clicked, the fields *Password* and *Select Operation* are enabled.

Then, the simulator prompts the user for information entry. When execution reaches a node in GT from which several continuation paths are possible (e.g., button *Confirm* clicked), the prototype displays the dialog box for scenario selection. In the example of Figure 14, the upper selection corresponds to the scenario *regularIdentify* and the lower one to the scenario e*rrorIdentify*. Once a path has been selected, the traversal of GT continues.

4 Related Work

In this section, we review some related work in the area of UI specification and automatic generation based on Petri nets. Each of the three approaches presented below suggests some high-level UI specification, yet none of them generates these specifications from scenario descriptions. For a discussion of work dealing with the simulation of specifications and with UI generation from specifications other than Petri nets refer to [5].

In the *TADEUS* approach [20] (TAsk-based DEvelopment of User interface Software), the development of the UI follows three stages: the requirements analysis and specification, the dialogue design, and the generation of the UI prototype. During the dialogue design stage, two levels of dialogues are manually built: the navigation dialogue that describes the sequencing between different task presentation units (dialogue views), and the processing dialogue that describes the changes on UI objects inside a dialogue view. The result of generation is a script file for an existing UI management system. Dialogue graphs is the formalism used to specify dialogues. It is Petri net based, and it permits to model easily different types of dialogues (single, multi, modal, etc.). In this approach, dialogues are manually defined by the UI developer.

Palanque et al. [15] provide the Interactive Cooperative Objects (ICO) formalism for designing interactive systems. Using ICO, an object of the system has four components: a data structure, a set of operations representing the services offered by the object, an object control structure defining the object behavior with high-level Petri nets, and a presentation structured as a set of widgets. The windows of the UI and their interrelations are modeled with ICO objects. This approach is completely manual, wheras in our approach the only manual operation is the building of the object state tables (see Section 3.2.2).

De Rosis et al. [17] propose a task-based approach using Petri nets to describe interactive behavior. They represent a UI by a CPN where transitions are the tasks of the system and places represent the information display after a task execution. A logical projection function is associated to places and transitions of the CPN in order to describe in form of conditions the user actions and the information displayed as a result of performing actions. To complete the UI description, the designer links the physical aspect of the interface to the CPN by means of a physical projection function. This work focuses on specifying the UI, yet does not address UI generation.

5 Discussion of Approach

Below, we discuss our approach in respect to the following points: scenario-based approach, system and object views, visual formalisms for modeling interactive systems, and validation of approach.

5.1 Scenario-based approach

Our approach to UI generation exhibits the advantages of scenario-based techniques. In contrast to many data-oriented methods, UIs are generated from specifications describing dynamic system behavior, which are derived from task analysis. Once they are generated, data specifications may be used as the basis for further refinements. In line with Rosson [18] who advocates a “model-first, middle-out design approach” that interleaves the modeling of objects and scenarios, we put the emphasis on the (dynamic) modeling aspect, and generate the dynamic core of UIs rather than focus on screen design and the user-system dialog.

As scenarios are partial descriptions, there is a need to elicit all possible scenarios of the system to produce a complete specification. In our approach, colors of Petri nets are used to inhibit scenario interleaving, that is, the resulting specifications will capture exactly the input scenarios. The integration algorithm can be configured to allow scenario interleaving and to capture more than the mere input scenarios. In this way, new scenarios may be generated from already existing ones.

It is well known that scalability is an inherent problem when dealing with scenarios for large applications. Our approach eases this problem by integrating scenarios on a per use case basis, rather than treating them as an unstructured mass of scenarios.

5.2 System and object views

In this paper, we focus on using scenarios for the restructuring of the formal system specification (*system view*). In our previous work [5, 13], we have addressed the specification of individual objects in interactive systems (*object view*). In a scenario-based approach, the specification of the behavior of a given object can be seen as the projection of the set of acquired scenarios on that object. When projecting scenarios onto an object, only messages entering and exiting the object are considered. In this way, the sequence order of messages in the scenarios is lost, and this can lead to capture undesirable behaviors. Furthermore, in an object view, all interface objects must be explicitly identified in the underlying set of scenarios. On the other hand, in a system view, there is no loss of precision, and the UI can be generated from the system specification without the need to explicitly identify UI objects.

For the purpose of UI generation, a system view is appropriate when in all use cases and associated scenarios only one user interacts with the system. In the case of collaborative tasks (more than one user interacts with the use cases), however, an object view will be more suitable.

5.3 Visual formalisms for modeling interactive systems

Petri nets and Statecharts are among the most powerful visual formalisms used for specifying complex and interactive systems. Statecharts are an extension of state machines to include hierarchies by allowing state refinement, and concurrency by describing independent parts of a system. Concurrency in Statecharts is modeled via orthogonal states. An orthogonal state is a Cartesian product of two or more Statecharts (threads). Sending messages between threads of an orthogonal state is forbidden. Leaving a state of a thread of an orthogonal state leads to exit all threads of this orthogonal state. These restrictions do not apply to the Petri net formalism. Petri nets are known for their support of pure concurrency, all transitions having a sufficient number of tokens in their input places may concurrently be fired. Petri nets in their basic form do not support hierarchy, but the extension of Petri nets used in tools such as designCPN allow for hierarchies in the specification.

Non-deterministic choices can more easily be modeled using Petri nets than based on Statecharts. When two or more transitions share the same input places, the system chooses randomly to fire one of these transitions. In Statecharts, non-deterministic choices cannot directly be modeled because messages are ordered and broadcasted to all concurrent threads.

In many UIs, we have to model exclusive executions (modal windows). This can easily be done using the history states of Statecharts. When entering a modal window, an event must be broadcasted to all concurrent threads to enter their history states. After exiting the modal window, all concurrent threads must be re-entered in the states they were before. Modeling exclusive execution with Petri nets requires extensions of the formalism, as discussed for instance in [11].

Tokens that are specific to Petri nets can be used both in controlling and simulating system behavior, and in modeling data and resources of the system. If the place *Begin* of the Figure 6 contains only one token, the system can only execute one use case at a time. When the place contains *n* tokens, *n* concurrent executions of different use cases are possible. It may even be possible to execute *n* scenarios of the same use case (multiple instances).

Table 3 summarizes the differences between Petri nets and Statecharts based on the above discussion, indicating the strengths (+ for good and ++ for very good) and weaknesses (-) of the two formalisms. We believe that the considered criteria are all highly relevant to modern UIs. Depending on the type of UI at hand, one or the other formalism and modeling approach will be more appropriate.

Criterion	Petri net	Statechart
Concurrency	++	+
Non-determinism	+	-
History states	-	+
Multiple instances	+	-

Table 3: Differences between Petri nets and Statecharts from a UI perspective.

5.4 Validation of approach

The scenario integration and prototype generation activities have all been implemented in Java, resulting in about 4,000 commented lines of code. The Java code generated for the UI prototype is fully compatible with the interface builder of Visual Café. For obtaining CPNs in serialized form, as required by the scenario integration algorithm, the XML tools mentioned at the end of Section 3.2.2 are used.

Our approach has been successfully applied to a number of small-sized examples. For further validation, we have started implementing the suite of examples used for validating *SUIP* [22], the implementation of our Statechart-based approach [5]. This suite includes a library system, a gas station simulator, and a filing system. The gas station simulator will be of particular interest since it involves two actors. “Extreme” examples, i.e., examples that lend themselves particularly well to the Petri net or the Statechart based approach, respectively, will also be examined, in order to further investigate the differences between the two approaches (cf. previous subsection and discussion of system versus object views).

6 Conclusion and Future Work

The work presented in this paper proposes a new approach to the generation of UI prototypes from scenarios. Scenarios are acquired as SequenceDs enriched with UI information. These SequenceDs are transformed into CPN specifications from which the UI prototype of the system is generated. Both static and dynamic aspects of the UI are derived from the CPN specifications.

The most interesting features of our approach lie in the automation brought upon by the deployed algorithms, in the use of the scenario approach addressing not only sequential scenarios but also scenarios in the sense of the UML (which supports, for instance, concurrency in scenarios), and in the derivation of executable prototypes that are embedded in a UI builder environment for refinement. The obtained prototype can be used for scenario validation with end users and can be evolved towards the target application.

As future work, we plan to move in three directions. As mentioned above, we intend to further pursue our comparison of modeling approaches. This will also include the study of the interrelationship between modeling formalism and the UI paradigm being supported. Second, we wish to investigate backward engineering, that is, allowing the automatic modification of scenarios through the UI prototype. Finally, we plan to further study the verification aspect of scenario-based modeling [4], especially the completeness of the system specification obtained from partial descriptions in form of scenarios.

Acknowledgments

We would like to thank Mazen Fahmi for helping us in the implementation part of this work. Our thanks also go to our colleagues in the *Forspec* project, Gilbert Babin, Jules Desharnais, François Lustman, and Ali Mili.

Bibliography

[1] J. Clark. SX: An SGML System Conforming to International Standard ISO 8879, <http://www.jclark.com/sp/sx.htm>.

[2] K. W. Derr, Applying OMT: A practical step-by-step guide to using the Object Modeling Technique, SIGS BOOKS/Prentice Hall, 1996.

[3] designCPN: version 3, Meta Software Corp. <http://www.daimi.aau.dk/~designcpn>.

[4] M. Elkoutbi, User Interface Engineering based on Prototyping and Formal Methods. PhD thesis, Université de Montréal, Canada, April 2000. French title: Ingénierie des interfaces usagers à l'aide du prototypage et des méthodes formelles. To appear.

[5] M. Elkoutbi, I. Khriss, and R. K. Keller, Generating User Interface Prototypes from Scenarios, in Proceedings of the Fourth IEEE International Symposium on Requirements Engineering (RE'99), pages 150-158, Limerick, Ireland, June 1999.

[6] M. Elkoutbi and R. K. Keller, Modeling Interactive Systems with Hierarchical Colored Petri Nets, In Proc. of 1998 Adv. Simulation Technologies Conf., pp. 432-437, Boston, MA, April 1998. Soc. for Comp. Simulation Intl. HPC98 Special session on Petri-Nets.

[7] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal approach to scenario analysis, IEEE Software, (11)2, March 1994, pp. 33-41.

[8] IBM, Systems Application Architecture: Common User Access – Guide to User Interface Design – Advanced Interface Design Reference, IBM, 1991.

[9] IBM, XML Parser for Java, in Java Report's February 1999, <http://www.alphaworks.ibm.com/formula/xml>.

[10] K. Jensen, Coloured Petri Nets, Basic concepts, Analysis methods and Pratical Use, Springer, 1995.

[11] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Coupling asynchrony and interrupts: Place chart nets. In P. Azema and G. Balbo, editors, ATPN'97, pp.328-347, Toulouse, France, June 1997. Springer-Verlag. LNCS 1248.

[12] Koskimies K. and Makinen E.: Automatic Synthesis of State Machine from Trace Diagrams, Software Practice & Experience (1994), 643-658.

[13] Ismail Khriss, Mohammed Elkoutbi, and R. K. Keller. Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams. In J. Bezivin and P. A. Muller, ed., <<UML>>'98: Beyond the Notation, pp.132-147. Springer, '99. LNCS 1618.

[14] B. A. Nardi, The Use of Scenarios in Design, SIGCHI Bulletin, 24(4), October 1992.

[15] P. Palanque and R. Bastide, Modeling clients and servers in the Web using Interactive Cooperative Objects, Formals Methods in HCI,Springer, 1997, pp.175-194.

[16] C. Potts, K. Takahashi and A. Anton, Inquiry-Based Scenario Analysis of System Requirements, Technical Report GIT-CC-94/14, Georgia Institute of Technology, 1994.

[17] F. de Rosis, S. Pizzutilo and B De Carolis: A tool to support specification and evaluation of context-customized interfaces. SIGCHI Bulletin, 28, 3, 1996.

[18] M. B. Rosson. Integrating Development of Task and Object Models, Communications of the ACM, 42(1), January 1999, pp. 49-56.

[19] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual, Addison Wesley, Inc., 1999.

[20] E. Schlungbaum and T. Elwert, Modeling a Netscape-like browser Using TADEUS Dialogue graphs, In Handout of CHI'96 Workshop on Formal Methods in Computer Human Interaction: Comparison, Benefits, Open Questions, Vancouver, 1996, pp.19-24.

[21] Siegfried Schönberger, Rudolf K. Keller, and Ismail Khriss. Algorithmic Support for Model Transformation in Object-Oriented Software Development. Theory and Practice of Object Systems (TAPOS), 2000. John Wiley and Sons. to appear.

[22] SUIP. Scenario-based user interface prototyping: Website and public domain software distribution, September 1999. <http://www.iro.umontreal.ca/labs/gelo/suip/>.

[23] Symantec, Inc. Visual Café for Java : User Guide, Symantec, Inc., 1997.