Generating User Interface Prototypes from Scenarios

Mohammed Elkoutbi
Département IRO
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal, QC H3C 3J7, Canada
elkoutbi@iro.umontreal.ca

Ismaïl Khriss
Département IRO
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal, QC H3C 3J7, Canada
khriss@iro.umontreal.ca

Rudolf K. Keller
Département IRO
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal, QC H3C 3J7, Canada
keller@iro.umontreal.ca

Abstract

Requirements capture by scenarios and user interface prototyping have become popular techniques. Yet, the transition from scenarios to formal specifications is still ill-defined, and prototyping remains weak in linking the application domain with the user interface. Most importantly, the prototyping and the scenario approaches lack integration in the overall requirements engineering process. In this paper, we suggest a process that generates a user interface prototype from scenarios and yields a formal specification of the application[1]. The approach is based on the Unified Modeling Language (UML), and the generated prototypes are embedded in a user interface builder environment for further refinement. The algorithms underlying the approach have been implemented and applied on a number of examples.

1. Introduction

Over the past years, scenarios have received significant attention and have been used for different purposes such as human computer interaction analysis [15], specification generation [1], object-oriented analysis and design [4, 10, 17], and requirements engineering [9,16]. A typical process for requirements engineering based on scenarios [9] 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.

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).

In this paper, we suggest an approach for requirements engineering supporting the Unified Modeling Language (UML). The approach provides a five activities process with limited manual intervention for deriving a prototype of the UI from scenarios and generating a formal specification of the application. Scenarios are acquired in the form of UML collaboration diagrams and enriched with UI information. These diagrams are automatically transformed, based on our previous work [13, 19], into the UML Statechart specifications of all the objects involved. An algorithm is applied to generate a UI prototype from the set of obtained specifications. The prototype is embedded in a UI builder environment for further refinement.

Section 2 of this paper gives a brief overview of the UML diagrams relevant for our work and introduces a running example. Section 3 presents the five activities of our approach. Section 4 describes in detail the fifth of these activities, the algorithm for deriving a UI prototype from dynamic specifications. Section 5 addresses related work. In Section 6, we discuss several aspects of our work. Finally, Section 7 provides some concluding remarks and points out future work.

2. Unified Modeling Language

The UML [18] 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 approach: Use Case diagram (UsecaseD), Collaboration diagram (CollD), and Statechart diagram (StateD). As a running example, we have chosen to study a part of an extended version of the library system described in [5].

2.1. 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

[1] This work is in part supported by FCAR and by the SPOOL project organized by CSER (Consortium Software Engineering Research) which is funded by Bell Canada, NSERC (Nat. Sciences and Research Council of Canada), and NRC (Nat. Research Council of Canada).

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 we represent it by a directed dashed line. Figure 1 shows, as an example, the UsecaseD for the library system. 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.

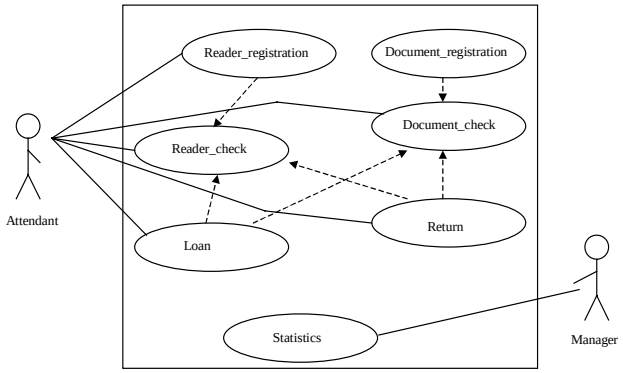

Figure 1: UsecaseD for the library system.

2.2. Collaboration diagram (CollD)

A scenario shows a particular series of interactions among objects in a single execution of a use case of a system (execution instance of a use case). Scenarios can be viewed in two different ways: through sequence diagrams (SequenceDs) or CollDs. 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 CollDs because the UML documentation defines them more precisely than SequenceDs.

A CollD is a graph where nodes are objects participating in the scenario and edges represent structural relations between objects (association, aggregation, inheritance, etc.). Messages sent between objects are labeled with a text string and a direction arrow. One edge can be used to send many messages in both directions. Each message label contains a sequence number representing the nested procedural calling sequence throughout the scenario, an optional widget mark (see Section 3.1), and the message name.

Sequence numbers contain a list of sequence elements separated by dots. Each sequence element may consist of several parts, such as:

- a compulsory number showing the sequential position of the message,
- a letter indicating a concurrent thread (see messages *1.11a* and *1.11b* in Figure 2(a)), and
- an iteration indicator * indicating that several messages of the same form are sent to the specified target.

For a complete definition of CollDs refer to [18].

Figures 2(a), 2(b), and 2(c) depict three scenarios (CollDs) of the use case *Loan*. Figure 2(a) represents the scenario where the loan is correctly registered, Figure 2(b) represents the case where the loan is canceled, and Figure 2(c) shows the scenario where the user is not registered yet in the system.

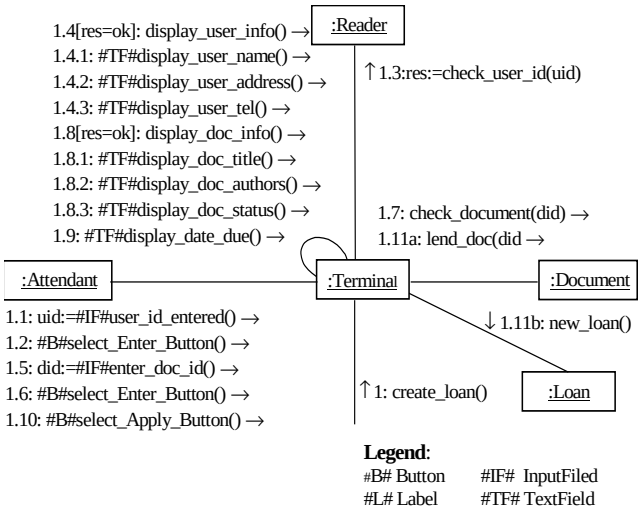

Figure 2(a): Scenario *regularLoan*.

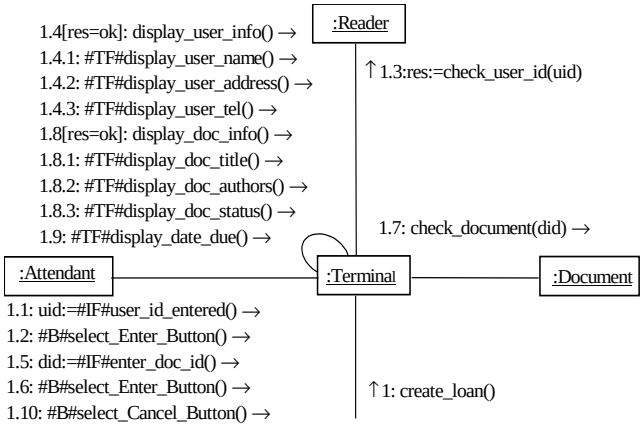

Figure 2(b): Scenario *cancelLoan*.

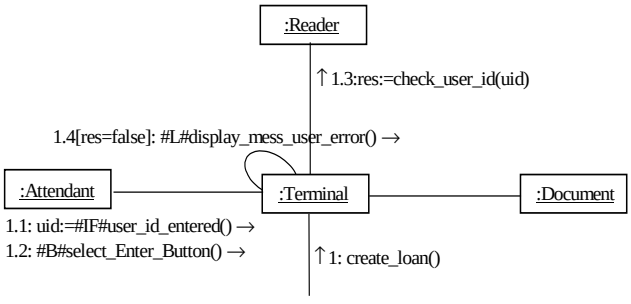

Figure 2(c): Scenario *errorUserLoan*.

2.3. Statechart diagram (StateD)

A StateD shows the sequence of states that an object goes through during its life cycle in response to stimuli. Generally, a StateD may be attached to a class of objects with an interesting dynamic behavior.

The formalism (notation and semantics) used in StateDs is derived from Statecharts as defined by Harel [6]. Any state in a StateD can be recursively decomposed into exclusive states (*or-state*) or concurrent states *(and-state*).

As an illustration, Figure 4 depicts the StateD of the object *Terminal*. The state *waitingForApplyOrCancel*, for instance, is an *and-state* composed of two concurrent substates separated by a dashed line.

3. Description of the Approach

In this section, we describe the overall approach to derive a UI prototype of a system. The approach consists of five activities (see Figure 3), which are detailed below:

- Requirements acquisition
- Generation of partial specifications from scenarios
- Analysis of partial specifications
- Integration of partial specifications
- User interface prototype generation.

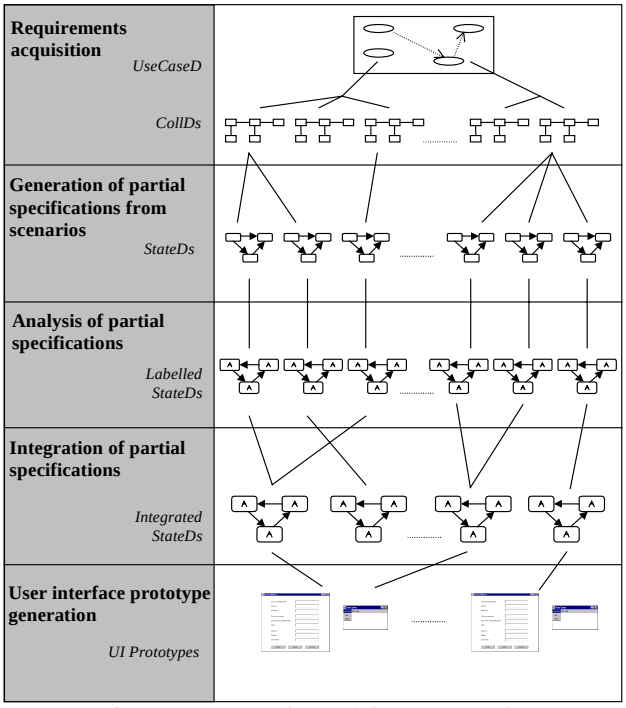

Figure 3: Overview of the approach.

3.1. Requirements acquisition

In this activity, the analyst first elaborates the UsecaseD of the system (see Figure 1). Then, he or she acquires scenarios as CollDs for each use case in the UsecaseD. Figures 2(a), 2(b), and 2(c) show the three sample CollDs corresponding to the use case *Loan* of the library system.

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 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 *Loan* has one normal scenario (scenario *regularLoan* with frequency 10) and two scenarios of exception (scenario *cancelLoan* with frequency 3 and scenario *errorUserLoan* with frequency 5).

In our example, the object *Terminal* is a special object called *interface object*. An interface object is defined as an object through which the user interacts with the system to enter input data and receive results. For UI generation purposes, messages corresponding to user interactions are marked in the CollDs with the type of interaction objects (i.e., widgets) that the analyst wants to find in the resulting UI. For example in Figure 2(a), the mark *#B#* at the beginning of the name of message *1.2* means that this message corresponds to a user interaction with the *Button* widget. Note that *#TF#* stands for *Text Field* as in message *1.4.1*, *#IF#* for *Input Field* as in message *1.1*, and *#L#* for *Label* as in message *1.4* in Figure 2(c).

3.2. Generation of partial specifications from scenarios

In this activity, we repeatedly apply on each CollD the CollD-To-StateD transformation algorithm described in [19]. As a result, for each object and each scenario in which it participates, a partial specification (StateD) is obtained.

3.3. Analysis of partial specifications

In order to integrate multiple StateDs of a given object (activity four, see Section 3.4 below), the analyst must identify equivalent states and give them common state names. Unique states are labeled with unique state names.

3.4. Integration of partial specifications

The objective of this activity is to integrate for each object and each use case in which it participates all its partial StateDs into one single StateD per use case [13]. As an example, Figure 4 shows the resultant StateD of the *Terminal* object after the integration of the three scenarios of use case *Loan*. Note that StateDs can be integrated across use cases, and thus the set of "global" StateDs as needed for subsequent design and implementation can be easily obtained, too.

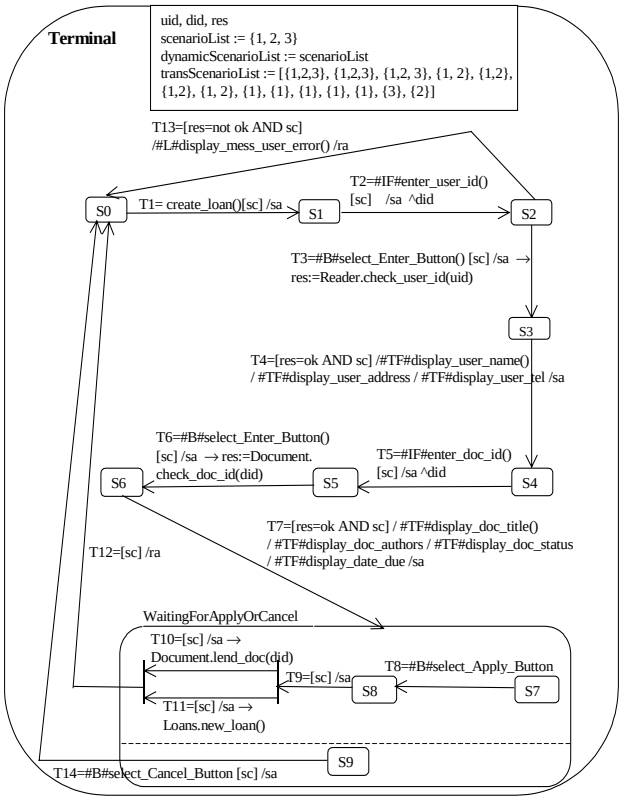

Figure 4: Resultant StateD for the *Terminal* object.

3.5. User interface prototype generation

In this activity, we derive UI prototypes for all the interface objects found in the system. Both the static and the dynamic aspects of the UI prototypes are generated from the StateDs of the underlying interface objects. For each interface object, we generate from its StateDs, as found in the various use cases, a standalone prototype. This prototype comprises a menu to switch between the different use cases. The different screens of the prototype visualize the static aspect of the object; the dynamic aspect of the object 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 Figures 5 and 9). The details of prototype generation are described in the next section.

4. Algorithm for User Interface Prototype Generation

In this section, we detail the process of prototype generation from interface object behavior specifications. This process can be summarized in the following algorithm (in the pseudocode, we use the "dot"-notation known from object-oriented languages).

```
Let IO be the set of interface objects in the
system,
Let UC={uc1, uc2,..., ucn} be the set of use cases
of the system,
For each io in IO
    For each uci in UC
        If io.usedInUsecase(uci) then
            sd = io.getStateDforUsecase(uci)
            sd.generatePrototype()
        End If
    End For
    io.generateCompletePrototype()
End For
```

The operation *usedInUsecase(uc$_i$)*, applied to the object *io*, checks if the object *io* participates or not in one or more of the CollDs associated with use case *uc$_i$*. If the operation returns *true*, the operation *getStateDforUsecase(uc$_i$)* is called, which retrieves *sd*, the StateD capturing the behavior of object *io* that is related to this use case. From StateD *sd*, a UI prototype is generated using the operation *generatePrototype()*.

The operation *generateCompletePrototype()* integrates the prototypes generated for the various use cases into one single application. This application comprises a menu (see Figure 5) providing as options the different use cases in which object *io* participates.

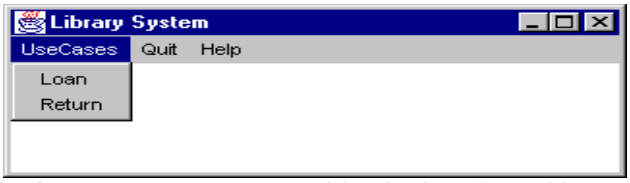

Figure 5: Menu generated for the interface object *Terminal*.

The operation of prototype generation (*generatePrototype()*) is composed of five operations, which are described in the sections below:

- generating graph of transitions
- masking non-interactive transitions
- identifying user interface blocks
- composing user interface blocks
- generating the user interface from composed blocks.

4.1. Generating graph of transitions

This operation consists of deriving a directed graph of transitions (GT) from the StateD of an interface object *io* related to a use case *uc$_i$*. Transitions of the StateD will represent the nodes of the GT. Edges will indicate the precedence of execution between transitions. If transition t_1 precedes transition t_2 in execution, we will have 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 for the graph). The *nodeList* of a GT is easily obtained since it corresponds to the transition list of the StateD at hand. The *edgeList* of a GT is obtained by identifying for each transition *t* all the transitions that

enter the state from which *t* can be triggered. All these transitions precede the transition *t* and hence define each an edge to node *t*.

The following algorithm details how to get *nodeList*, *edgeList*, and *initialNodeList* of the GT from a given StateD *sd*.

```
// returns the list of transitions in StateD sd
nodeList = sd.TransitionList()

// edgeList computation:
edgeList = ∅
For each t_i ∈ nodeList
    s = sd.FromState(t_i)
    // returns the state from which
    // the transition t_i is originating
    List = sd.inputTransitions(s)
    // returns the list of transitions that
    // enter the state s
    For t_e ∈ List edgeList.addEdge(t_e,t_i)
    // case where s is an initial state of sd:
    If s.type == initialState
        s_s=s.superState()
        // returns the parent state of s
        List = sd.inputTransitions(s_s)
        For t_e ∈ list
            edgeList.addEdge(t_e,t_i)
    End If
    // case where s is a composite state
    // (and-state, or-state):
    If (s.type==andState) or (s.type==orState)
        List = s.transitionsInside()
        // returns the list of transitions inside
        // the composite state s
        For t_e ∈ List edgeList.addEdge(t_e,t_i)
    End If
End For

// initialNodeList computation:
initialNodeList = ∅
LIS=sd.initialStates()
For each s ∈ LIS
    OT = s.outputTransitions()
    // returns the list of transitions that
    // fan out from s
    initialNodeList = initialNodeList ∪ OT
End For
```

Given the StateD of *Terminal* for the use case *Loan* (see Figure 4), the above algorithm generates the GT shown in Figure 6(a). The star character (*) is used to mark initial nodes in the graph.

4.2. Masking non-interactive transitions

This operation consists of removing all transitions that do not directly affect the UI (i.e., that do not carry widgets). These transitions are called *non-interactive* transitions. All such transitions are removed from the list of nodes *nodeList* and from the list of initial nodes *initialNodeList*, and all edges defined by those transitions are removed from *edgeList*.

When a transition *t* is removed from *nodeList*, we remove all edges where *t* takes part, and we add new edges in order to "bridge" the removed transition nodes. If the *initialNodeList* list of initial transitions contains any non-interactive transitions, they are replaced by their successor nodes. The following pseudocode details this operation (the update of *initialNodeList* is not shown):

```
For each t ∈ nodeList
    If t.widget()='' then
        nodeList.delete(t)
        ITL=edgeList.inputEdge(t)
        // returns the list of transition t_i
        // with (t_i,t) ∈ edgeList
        OTL=edgeList.outputEdge(t)
        // returns the list of transitions t_e
        // with (t,t_e) ∈ edgeList
        For each t_i ∈ ITL
            For each t_e ∈ OTL
                edgeList.addEdge(t_i,t_e)
                edgeList.deleteEdge(t_i,t)
                edgeList.deleteEdge(t,t_e)
             End For
        End For
    End If
End For
```

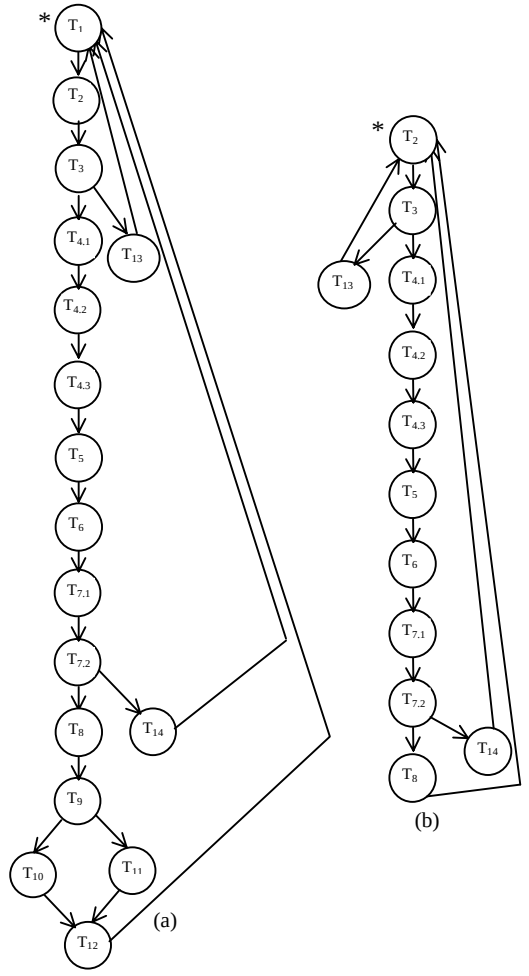

Figure 6: (a) Transition graph for the object *Terminal* and the use case *Loan* (GT). (b) Transition graph after masking non-interactive transitions (GT').

The result of this operation on the graph of Figure 6(a) is given in Figure 6(b).

4.3. Identifying user interface blocks

This operation consists of constructing a directed graph where nodes represent *User Interface Blocks* (UIB). A UIB is a subgraph of GT' consisting of a sequence of transition nodes that is characterized by a single input and a single output edge. The beginning and the end of each UIB is identified from the graph GT' based on the following rules:

(Rule 1) An initial node of GT' is the beginning of a UIB.

(Rule 2) A node that has more than one input edge is the beginning of a UIB.

(Rule 3) A successor of a node that has more than one output edge is the beginning of a UIB.

(Rule 4) A predecessor of a node that has more than one input edge ends a UIB.

(Rule 5) A node that has more than one output edge ends a UIB.

Applying these rules to the graph of Figure 6(b), we obtain the graph GB shown in Figure 7.

In this example, Rule 1 determines the beginning of B_1 (T_2) and Rule 5 the end of B_1 (T_3). Rules 3 and 5 determine the UIB B_2. The UIBs B_3, B_4, and B_5 are generated by applying Rule 3.

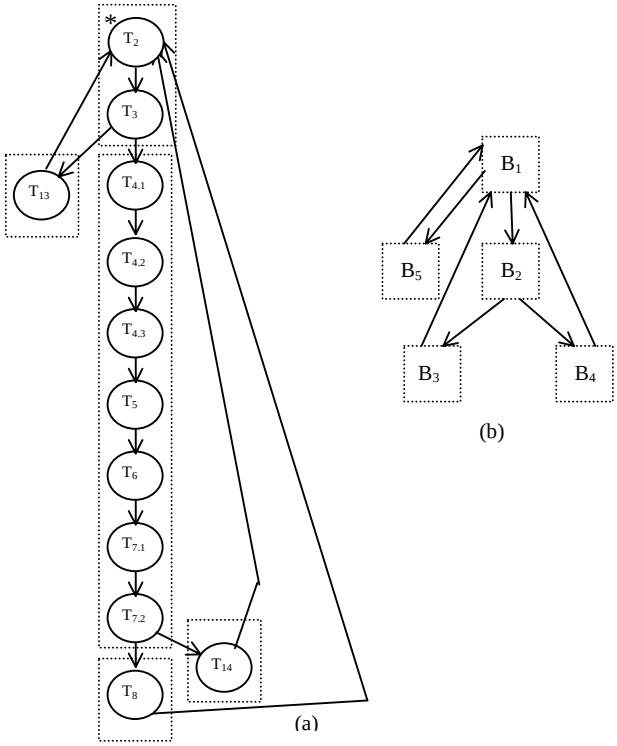

Figure 7: Graph GB resulting from UIB identification on the graph GT' of Figure 6(b): elapsed view (a) and collapsed view (b).

4.4. Composing user interface blocks

Generally, the UI blocks obtained from the previous operation contain only few widgets and represent only small parts of the overall use case functionality. Our approach supports the combination of UIBs in order to have more interesting blocks which can be transformed into suitable graphic windows. We use the following rules (heuristics) to merge the UIBs of a use case :

(Rule 6) Adjacent UIBs belonging to the same scenario are merged (scenario membership).

(Rule 7) The operation of composition begins with scenarios having the highest frequency (scenario classification, see Section 3.1).

(Rule 8) Two UIBs can only be grouped if the total of their widgets is less than 20 (ergonomic criterion).

The following algorithm explains the operation of composition.

```
Let uc_i be a given use case ∈ UC. Let GB be a
graph of UIBs that is derived from a StateD
belonging to uc_i
SL = uc_i.OrderedScenariolist()
// returns the scenario list of uc_i ordered by
// type and frequency
For sc ∈ SL
    BL = GB.LookforUIB(sc)
    // returns the list of (adjacent) UIBs
    // involved in scenario sc
    b = BL[1]
    i = 2
    While i < BL.size()
        While ergonomic(b,BL[i])
        // checks number of widgets < limit
        // (Rule 8)
            b.fusion(BL[i])
            // combine the UIB BL[i]
            // with its precedent UIB
            GB.delete(BL[i])
            // suppress B[i] from GB and update
            // the edges in GB
            i = i + 1
        End While
        B = BL[i]
        i = i + 1
    End While
End For
```

Applying this algorithm to the GB of Figure 7 results in the graph GB' of UIBs shown in Figure 8.

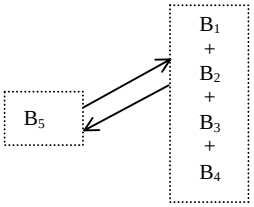

Figure 8: Graph GB' resulting from user interface block composition on the graph GB of Figure 7.

4.5. Generating the user interface from composed blocks

In this operation, we generate for each UIB of GB' a graphic frame. The generated frame contains the widgets of all the transitions belonging to the concerned UIB. Edges between UIBs in GB' are transformed to call functions in the appropriate frame classes. In our current implementation, Java code is generated that is compatible with the interface builder of *Visual Café* [20]. This gives the analyst the opportunity to customize the visual aspect of the generated frames.

The two frames derived from the composed blocks of the graph GB' of Figure 8 are shown in Figure 9.

The dynamic aspect of the UI is controlled by the behavior specification (StateD) of the underlying interface object. Running the generated prototype means symbolic execution of the StateD, or in our case, traversal of the transition graph GT'. The prototype responds to all user interaction events captured in GT', and ignores all other events.

To support prototype execution, a *Simulation Window* is generated (Figure 10, bottom window), as well as a dialog box to *Choose Scenarios* (Figure 10, middle right window). For example, after selecting the use case *Loan* from the *UseCases* menu (see Figure 10, top window), a message is displayed in the simulation window that confirms the use case selection and prompts the user to input the user identification and to click the *Enter* button. When execution reaches a node in GT' from which several continuation paths are possible, the prototype displays the dialog box for scenario selection. In the example of Figure 10, the upper selection corresponds to the scenario *errorUserLoan*, and the lower one to the scenarios *regularLoan* and *cancelLoan*. Once a path has been selected, the traversal of GT' continues.

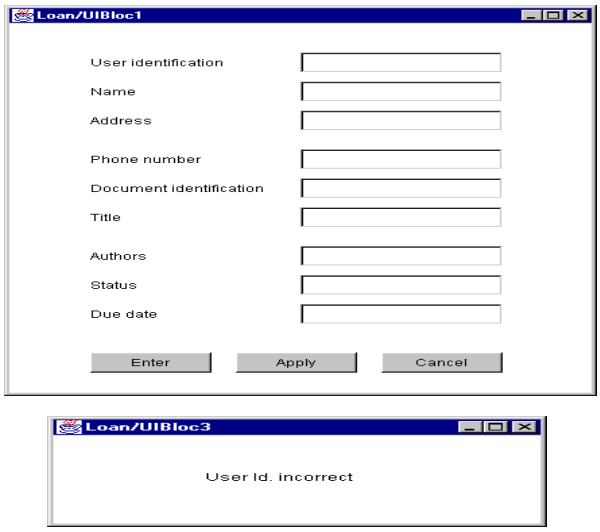

Figure 9: Frames generated for the use case *Loan*.

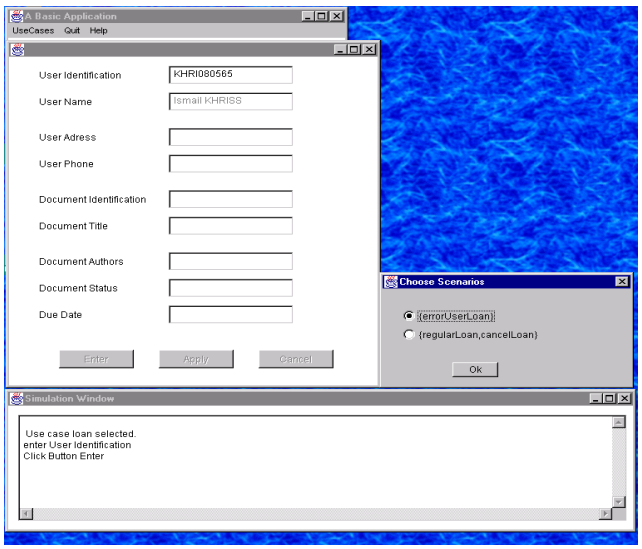

Figure 10: Prototype execution.

5. Related Work

In this section, we first review some related work in the area of automatic generation of UIs from specifications. Then, we address research dealing with the simulation of specifications.

A number of methods have been suggested for deriving the UI from specifications of the application domain. Typically, data attributes serve as input for the selection of interaction objects according to rules based on style guidelines such as *CUA* (Common User Access) [11]. Such methods include the *Genius*, *Janus*, and *TRIDENT* approaches.

In *Genius* [12], the application domain is captured in data models that are extended entity-relationship models. The analyst defines a number of views, where each view is a subset of entities and relationships of the overall data model, and specifies how these views are interconnected by means of a Petri-net based dialogue description. From these views and dialog specifications, Genius generates the UI. Note, however, that the specification process is completely manual.

Janus [2] derives the different windows of a UI from object models. Non-abstract classes are transformed into windows, whereas attributes and methods that are marked as irrelevant for the UI are ignored in the transformation process. Janus does not address the dynamic aspect of UIs.

Note that, in contrast to our approach, both Genius and Janus use data structure specifications for UI generation, but ignore task analysis altogether. As a consequence, such methods are little useful for systems other than data-oriented applications.

TRIDENT [3] leverages both task analysis and functional requirements analysis. Task analysis proceeds by decomposing the application into interactive tasks and by determining task attributes such as importance and user stereotype (user's task experience, user's system experience, etc.). Functional requirements analysis builds an entity-relationship model for the data and extracts from task analysis the tasks that should be treated as internal functions. An activity-chaining graph is drawn to connect interactive tasks to data and functions. This graph serves as the input for the selection of different windows, referred to as *presentation units*. TRIDENT addresses only the static aspect of UIs.

Simulation of specifications is supported by a variety of methods and tools, including *STATEMATE* and *SCR*, and the work by Koskimies et al.

STATEMATE [7] is a commercial tool, which provides graphical and diagrammatic languages for describing a system under development in three different views: structural, functional, and behavioral. Behavioral views are captured by StateDs. The tool supports system simulation for verification purposes as well as automatic code generation. UI generation is not supported. We consider STATEMATE as a complementary tool in respect to our approach: StateDs synthesized by a tool such as ours may be passed to STATEMATE for simulation and analysis, and conversely, StateDs of interface objects specified with STATEMATE may be complemented with a UI prototype using our approach.

The *SCR* method [8] suggests a tabular notation for specifying requirements and provides a set of tools for simulation and for automatic error detection. The formal model of specifications is the classic state machine model, and therefore, in contrast to StateDs, concurrency is not supported. The SCR simulator tool allows for the integration of UIs; yet, the UIs must be constructed manually using a GUI builder.

Koskimies et al. [14], finally, present an algorithm for synthesizing state machines (StateDs) from a set of scenarios (the differences to our synthesis algorithm are detailed in [19]). They propose an approach for design called *design by animation*. During the simulation of the synthesized state machines, new scenarios are generated which may in turn fuel the synthesis of more comprehensive state machines. Scenario generation can be supported via a UI, which must be crafted manually.

6. Discussion of Approach

Below, we discuss our approach in respect to the following points: scope and limitations, rapid and evolutionary prototyping, validation, and practicality.

Scope and limitations of approach

The scope of our approach is threefold: (1) it proposes a process for requirements engineering compliant with the UML, (2) it provides automatic support for building object specifications, and (3) it supports UI prototyping. Yet, at least three limitations apply. First, the analyst has the *manual* task of eliciting scenarios of the system and of labeling the generated partial StateDs. Second, our approach may be applied to windows and widgets interfaces, yet fails to support in its current form alternative UI paradigms. Finally, verification of characteristics such as coherence, completeness, etc. is not supported. Rather, we have to rely on external tools such as STATEMATE to verify the specifications.

Rapid and evolutionary prototyping

In the proposed framework, we aim at rapid prototyping for the purpose of end user validation at an early stage of development. The generated prototype serves as a vehicle for evaluating and enhancing the UI and the underlying specification. Since the prototype is generated in Java source code, it can be *evolved* at the code level towards the target application, to cover data and functional aspects. Since our framework is embedded in the UML, these aspects are provided as class diagrams and activity diagrams, respectively, that may be transformed into Java classes by use of a CASE tool.

Validation of approach

The three algorithms (see Sections 3.2, 3.4, and 3.5) that constitute the core of our approach have all been implemented in Java. For scenario acquisition and for the presentation of the resulting specifications, we have adopted two textual formats. The analyst may eventually be shielded from these formats by graphical editors for CollDs and StateDs, like the ones found in commercial CASE tools. The Java code generated for the UI prototype is fully compatible with the interface builder of *Visual Café* [20].

Note that the three algorithms have polynomial complexity. Our approach has been successfully applied to a number of examples such as the library system presented in this paper, a gas station simulator, an ATM (Automatic Teller Machine) system [17], and a filing system. On the average, one hour per scenario was spent to convert an informal scenario description into a newly generated UI prototype. For instance, in the case of the ATM example, comprising two use cases with a total of five scenarios, half a day's work yielded the overall UI prototype, as well as the complete set of StateDs of all interface and non-interface objects involved. We estimate that coding the prototype and synthesizing the StateDs by hand would have taken us double the time or more.

Practicality of approach

Our vision of a professional tool that supports our approach is a CASE tool supplying, beyond the functionality of the algorithms of the approach, graphical editors for the UML diagrams needed, as well as a "widget tool" for the specification by direct manipulation

of UI information within CollDs. Furthermore, such a tool may support a wider range of widget types than is currently being provided. At the conceptual level, to further practicality, the activity of analysis of partial object specifications (see Section 3.3) should be automated, and the rules for UI generation (see Sections 4.3 and 4.4) may be refined.

7. Conclusion and Future Work

The work presented in this paper proposes a new approach to the generation of UI prototypes from scenarios. The most interesting features lie in the automation brought upon by the deployed algorithms, in the support of scenarios that accommodate concurrent behavior, and in the derivation of executable prototypes that are embedded in a UI builder environment for refinement. The obtained prototypes can be used for scenario validation with end users and can be evolved towards the target application.

As future work, we aim to provide automatic support for verification of scenarios and specifications. Furthermore, we want to address the formal specification of data in order to eliminate the manual activities of our approach and to allow for the generation of interaction objects from data specifications.

References

[1] J. S. Anderson, and B. Durney, "Using Scenarios in Deficiency-driven Requirements Engineering", *Requirements Engineering'93*, IEEE Computer Society Press, 1993, pp. 134-141.

[2] H. Balzert, "From OOA to GUIs: The Janus System", *IEEE Software*, 8(9), February 1996, pp. 43-47.

[3] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, and J. Vanderdonckt, "A Model-based Approach to Presentation: A Continuum from Task Analysis to Prototype", *Proceedings of the Eurographics Workshop on Design, Specification, Verification of Interactive Systems*, Carrara, Italy, June 1994, Focus on Computer Graphics, Springer-Verlag, Berlin, pp.77-94.

[4] G. Booch, *Object Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994.

[5] H-E. Eriksson, and M. Pinker, *UML-Toolkit*, John Wiley and Sons, 1998.

[6] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 8, June 1987, pp. 231-274.

[7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, (16)4, April 1990, pp. 403-414.

[8] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, "SCR*: A Toolset for Specifying and Analyzing Software Requirements", *Proc. of the 10th Annual Conference on Computer-Aided Verification, (CAV'98)*, Vancouver, Canada, 1998, pp. 526-531.

[9] 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.

[10] I. Jacobson, M. Christerson, P. Jonson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

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

[12] C. Janssen, A. Weisbecker, and U. Ziegler, "Generating User Interfaces from Data Models and Dialogue Net Specifications", *Proc. of the Conference on Human Factors in Computing Systems (CHI'93)*, Amsterdam, The Netherlands, April 1993, pp. 418-423.

[13] I. Khriss, M. Elkoutbi, and R. K. Keller, "Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams", *Proc. of the Intl. Workshop on the Unified Modeling Language UML: Beyond the Notation*, Mulhouse, France, June 1998, pp. 115-126bis.

[14] K. Koskimies, T. Systa, J. Tuomi and T. Mannisto, "Automatic support for modeling OO software", *IEEE Software*, 15(1), January/February 1998, pp. 42-50.

[15] B. A. Nardi, "The Use Of Scenarios In Design", *SIGCHI Bulletin*, 24(4), October 1992.

[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] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented Modeling and Design*, Prentice-Hall, Inc., 1991.

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

[19] S. Schönberger, R. K. Keller and I. Khriss, *Algorithmic Support for Transformations in Object-Oriented Software Development*, Technical Rep. GELO-83, Univ. de Montréal, Montréal, Qc, Canada, April 1998.

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