

Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams¹

Ismail Khriiss, 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: {khriiss, elkoutbi, keller}@iro.umontreal.ca
<http://www.iro.umontreal.ca/~{khriiss, elkoutbi, keller}>

Abstract

The use of scenarios has become a popular technique for requirements elicitation and specification building. Since scenarios capture only partial descriptions of system behavior, an approach for scenario composition and integration is needed to produce more complete specifications.

The Unified Modeling Language (UML), which is emerging as a unified notation for object-oriented modeling, provides a suitable framework for scenario acquisition using Use Case diagrams and Collaboration diagrams and for behavioral specification using Statechart diagrams; yet it does not propose any specific modeling process, let alone a process for transforming scenarios into behavioral specifications.

In this paper we suggest a four step process for synthesizing behavioral specifications from scenarios. It generates from a given set of Collaboration diagrams the Statechart diagrams of all the objects involved. Our approach is incremental and is fully compliant with UML. Furthermore, it provides an elegant solution to the problem of scenario interleaving. The underlying algorithm has been implemented and validated with several examples, and is fit for integration into CASE tools supporting UML.

Keywords

Unified Modeling Language (UML), specification synthesis, scenario engineering.

1 Introduction

Recently, two major aspects have received more attention in object-oriented development: the emergence of the Unified Modeling Language (UML) as a unified notation that methods can use to express designs, and a growing consensus on Use Case (or scenario) approach to software development.

UML [RMH+97] is the successor of the wave of object-oriented analysis and design methods that appears in the late '80s and early '90s. It directly unifies the methods of Booch [Boo94], Rumbaugh (OMT) [RBP+91], and Jacobson (OOSE) [JCJO92]. UML is expected to be the standard modeling language in the future. It gives notation to describe all views of a system, but does not define any specific process for software development, beyond the preliminary work described in [RATL98].

Scenarios have received more significant attention and have been used for different purposes such as understanding requirements [PTA94], Human Computer Interaction analysis [Nar92], specification or prototype generation [AD93] and within object-oriented design methods like Booch, OMT, OBA [RG92] and OOSE.

In this paper, we extend our prior work [SK97] by proposing an incremental approach for dynamic modeling. It provides a four activities process with limited manual intervention for deriving objects dynamic specification from scenarios. Our work, in contrast to others such as [KSTM98], supports UML Collaboration diagrams with all their facets (iteration, condition, concurrency) for scenario acquisition and leverages the expressiveness of UML Statechart diagrams expressiveness (concurrency, hierarchy) for capturing the resultant specifications.

We also resolve the problem of interleaving between scenarios which means that the generated specifications captures exactly the behavior given in the input scenarios. For example, if we have two scenarios that share a

¹ This work is in part supported by FCAR (Fonds pour la formation des chercheurs et l'aide à la recherche au Québec) and by the SPOOL project organized by CSER (Consortium Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada).

common state, the resultant specification may capture more than the two given scenarios. During execution, the system begins with one scenario and when it reaches the common state, it can continue with the other.

The integration of our approach in CASE tools that support UML notation gives the possibility of automatic transformation between dynamic models.

Section 2 of this paper gives a brief overview of the UML diagrams relevant of our work and introduces a running example. Section 3 presents the four activities of our approach. Section 4 addresses related work in the area of specification generation from scenarios. In section 5, we discuss several aspects of our work. Finally, section 6 provides some concluding remarks and points out future work.

2 Unified Modelling Language

The UML [RMH+97] provides a syntactic notation to describe all views of a system using different kinds of diagrams. In this section, we will only discuss diagrams we have used in 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 a library system.

2.1 Use case diagram (UsecaseD)

The UsecaseD is concerned with 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 which 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 dashed line. The direction of a uses relation does not imply order of execution. Figure 1 shows an example of a use case diagram corresponding to the library system. In this UsecaseD, we find two actors (Attendant and Manager) interacting with seven use cases (Reader_check, Document_check, Reader_registration, Document_registration, Lend_service, Return_service and Statistics). There are also many relations uses, for example the use case *Lend_service* uses the services of *Reader_check* and *Document_check* uses cases.

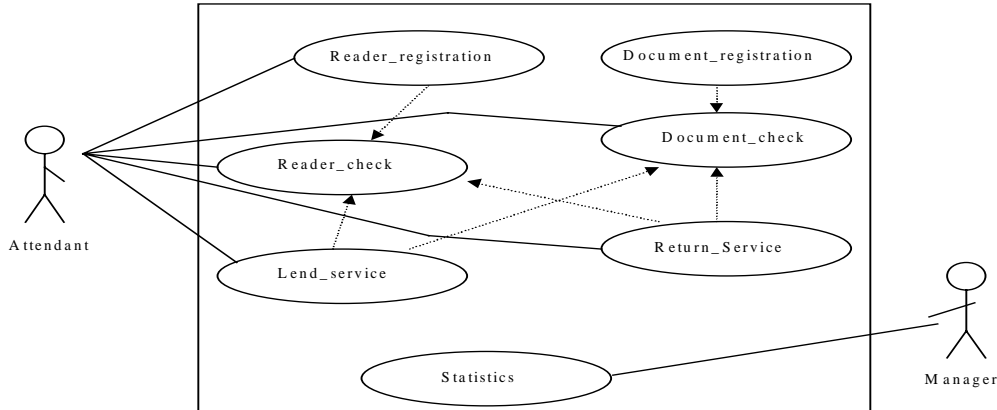


Figure 1: UsecaseD for the library system.

A UsecaseD is very helpful in visualizing the context of a system and the boundaries of the system's behaviour. A given use case is typically characterized by multiple scenarios.

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 SequenceDs (Sequence diagrams) or CollDs. Both type of diagrams rely on the same underlying semantics. Conversion from one to the other is possible.

A SequenceD shows interactions among a set of objects in temporal order, which is good for understanding timing issues.

<<UML>>'98

A CollID concentrates on the structure of the interaction between objects and their inter-relationships rather than focuses the temporal dimensions of a scenario. A CollID 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 labelled 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.

Sequence numbers contain a list of sequence elements separated by dots. Each element can have the following parts:

- a letter indicating a concurrent thread (see the message 1.4.2.1a in Figure 2a),
- a number showing the sequential position of the message,
- an iteration indicator * (see the message 1.4 in Figure 2a) indicating that several messages of the same form are sent sequentially to a single target or concurrently to a set of targets,
- etc.

Figures 2a, 2b and 2c give three scenarios (CollIDs) of the use case *Lend_service*. Figure 2a represents the scenario where the loan is correctly registered, Figure 2b represents the case where the document loaned is not registered in the system and Figure 2c shows the scenario where the user is not registered yet in the system.

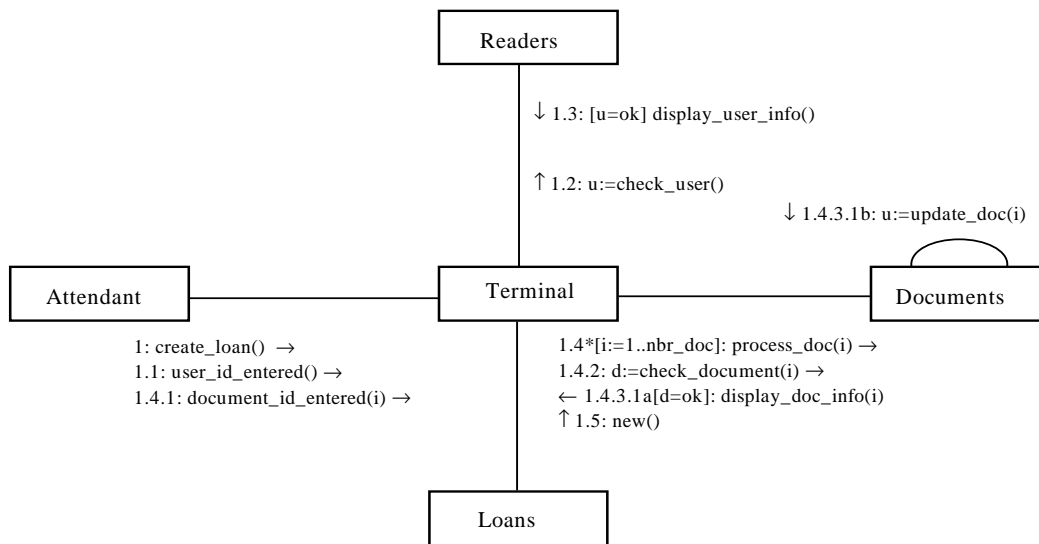


Figure 2a: scenario 1 corresponding to the use case *Lend_service*.

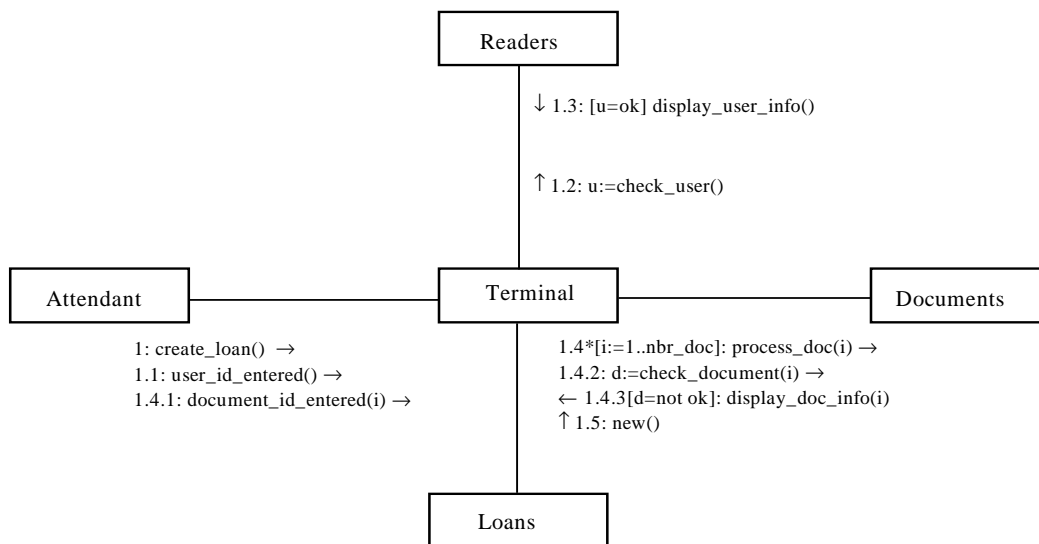


Figure 2b: scenario 2 corresponding to the use case *Lend_service*.

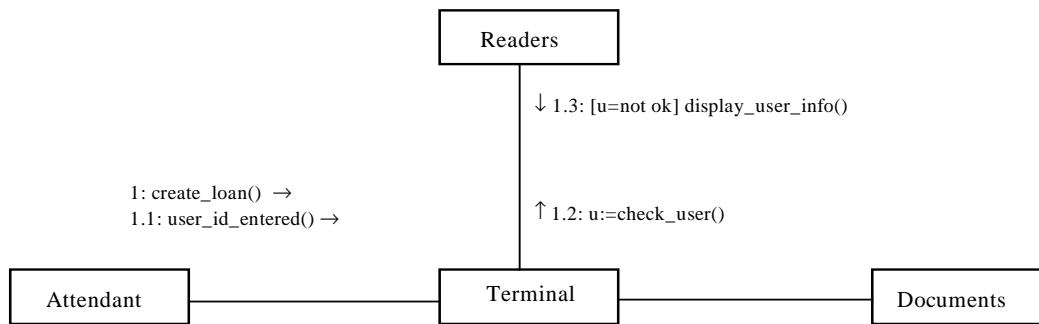


Figure 2c: scenario 3 corresponding to the use case *Lend_service*.

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 behaviour.

The formalism (notation and semantics) used in StateDs is derived from StateCharts as defined by Harel [Harel87]. StateCharts are an extension of state-event diagrams to include hierarchy and concurrency. Any state in a StateChart can be recursively decomposed into exclusive states (*or-state*) or concurrent states (*and-state*). When a transition in a Statechart is triggered (event received and guard condition tested), the object leaves its current state, initiates the action(s) for that transition and enter a new state. Any internal or external event is broadcasted to all states of all objects in the system. Transitions between concurrent states are not allowed, but synchronization and information exchange is possible through events.

As illustration, Figure 5a gives a StateD for the object Terminal where we can see an example of hierarchy and concurrency. The state *Processing_document_list* (*or-state*) is composed of two sub-states *check_list_doc* and *Processing_document* (*and-state*) which itself contains two concurrent sub-states separated by a dashed line.

3 Description of the approach

In this section, we describe the overall process to derive a system behaviour specification. This process will provide an automatic way to transform requirement informations to a formal specification. We consider that the behaviour specification of a system is given by the behaviour specifications of its constituent objects. The approach we define here consists of four major activities (see Figure 3):

1. Requirement acquisition
2. Generation of partial object specifications from scenarios
3. Analysis of partial specifications
4. Object specifications integration.

3.1 Requirement acquisition

Scenarios are one of techniques mostly used in this activity. They are used in object-oriented methodologies [JCJO92, RBP+91 and Booch94] as an approach to requirements engineering. The UML (which represents the unification of these o-o methodologies) propose a suitable framework for scenarios acquisition using UseCaseD for capturing system functionalities and SequenceDs or CollIDs for describing scenarios.

In this phase, the analyst begins by elaborating the UseCaseD for the system that consists to identify use cases and external actors interacting with. An example of such diagram was given in Figure 1. Then, he acquires scenarios as CollIDs for each use case in the UseCaseD. We have already shown in Figures 2a, 2b and 2c examples of three CollIDs corresponding to the use case *Lend_service* of the Library system.

At the end of this activity, we get as result:

- a set of use cases $UC = \{uc_1, uc_2, \dots, uc_n\}$ representing all functionalities of the studied system,
- a set of objects $OB = \{o_1, o_2, \dots, o_m\}$ participating in different scenarios of the system,
- a set of scenarios corresponding to each uc_i , the union of these sets is $CD = \{cd_1, cd_2, \dots, cd_k\}$.

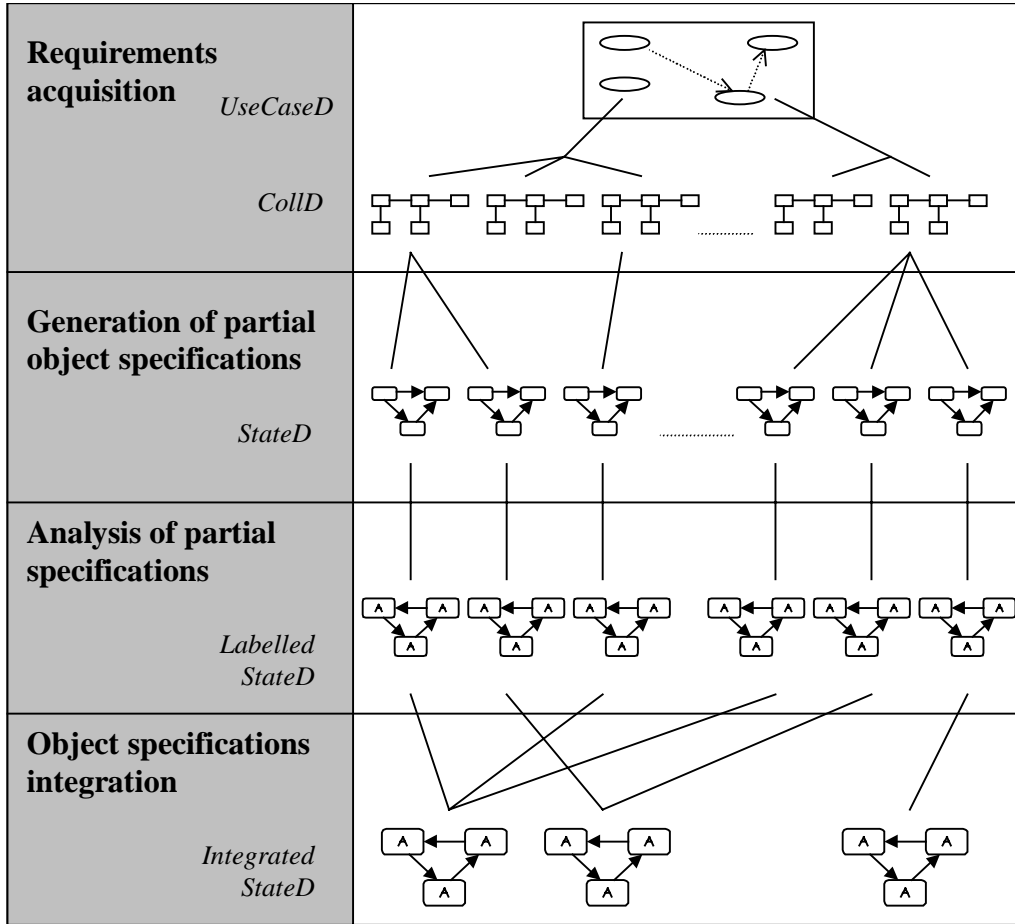


Figure 3: Overview of the approach.

3.2 Generation of partial object specifications from scenarios

In this step, we apply repeatedly on each element cd_i of the set CD the CTS (CollID To StateD) algorithm [SK97] in order to generate automatically partial specifications for objects participating in scenarios of the system.

Transforming one CollID into StateDs is a process of five steps [SK97]. Step 1 creates a StateD for every distinct class implied by the objects in the CollID. Step 2 introduces as state variables all variables which are not attributes of the objects of CollID. Step 3 creates transitions for the objects from which messages are sent. Step 4 creates transitions for the objects to which messages are sent. Finally, step 5 has a role to bring for all StateDs the set of generated transitions into correct sequences, connecting them by states, split bars and merge bars. The sequencing follows the type of messages in a CollID: iteration messages, conditional messages, concurrent messages and messages with multiple predecessors. Note that the CTS algorithm generates StateDs with concurrent states (see Figure 4a).

The result of this step is a set of StateDs $SD = \{sd_{i,j}, 1 \leq i \leq k, 1 \leq j \leq m\}$ where i refers to the CollID cd_i and j to the object o_j .

For illustration, we show in Figures 4a, 4b and 4c the StateDs generated by the CTS algorithm for the object Terminal.

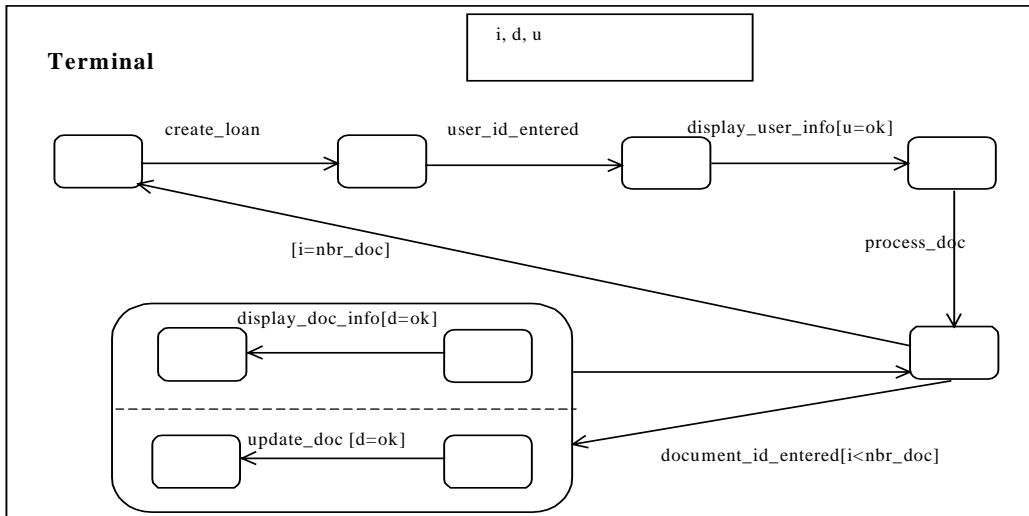


Figure 4a: StateD for object Terminal generated from scenario 1 given in Figure 2a.

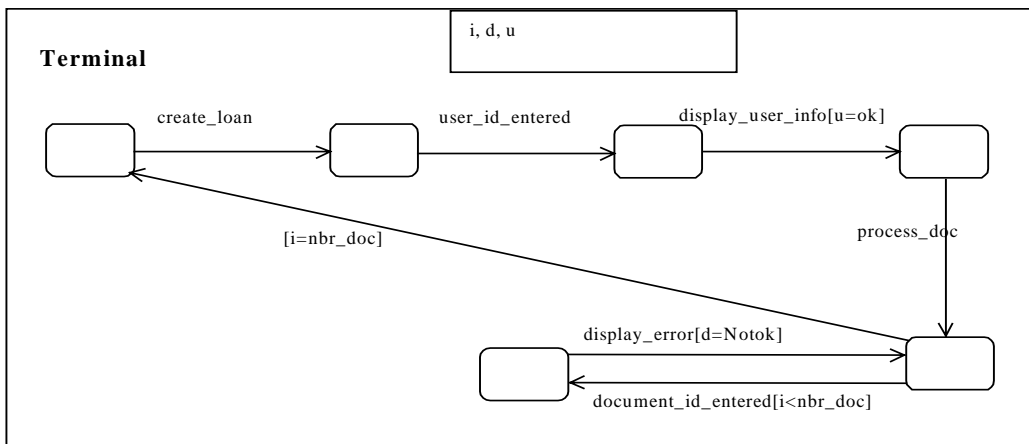


Figure 4b: StateD for object Terminal generated from scenario 2 given in Figure 2b.

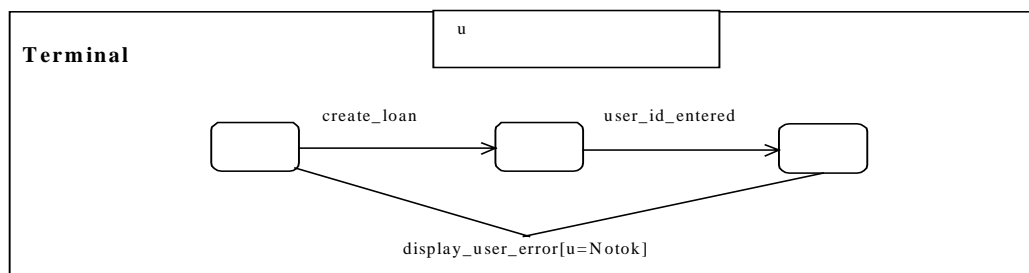


Figure 4c: StateD for object Terminal generated from scenario 3 given in Figure 2c.

3.3 Analysis of partial specifications

The previous activity generates StateDs with unlabeled states. In respect to the fourth activity, the analyst must add state names to the generated StateDs. In fact, our algorithm is based on state names as we will see later. He can also add structural informations like grouping states. Figures 5a, 5b and 5c show some added informations to the StateDs generated for the object Terminal (Figure 4a, 4b and 4c).

The result of this step is a set of StateDs SD' which is the same as SD plus additional textual and structural informations.

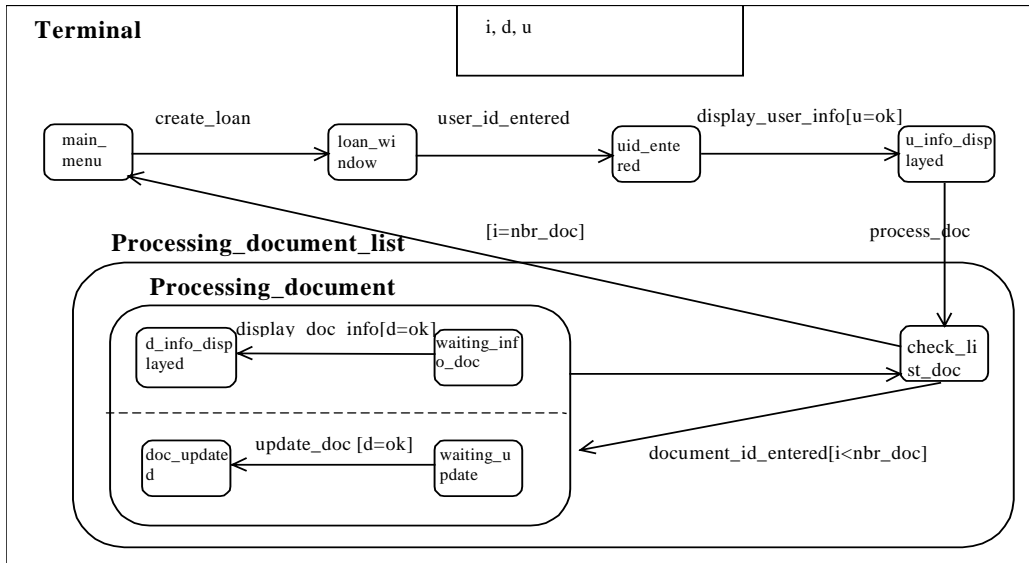


Figure 5a: the modified StateD for object Terminal corresponding to the StateD in Figure 4a.

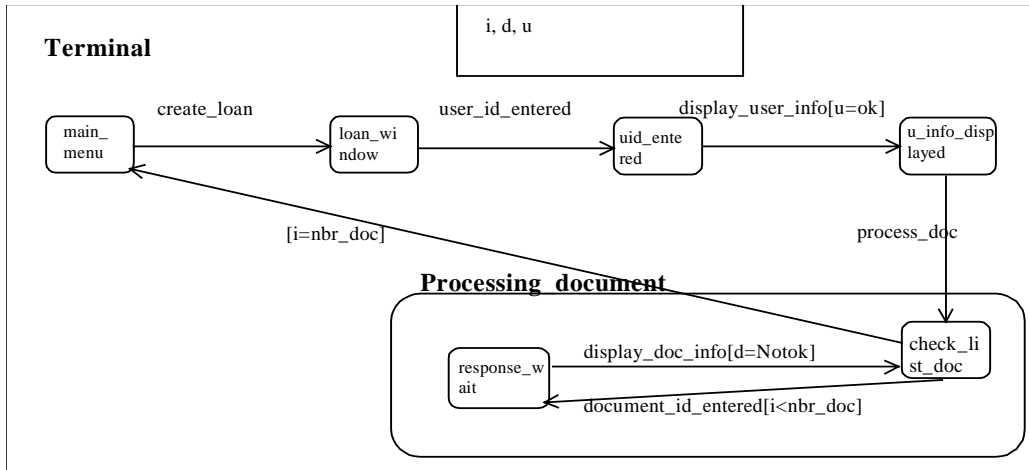


Figure 5b: the modified StateD for object Terminal corresponding to the StateD in Figure 4b.

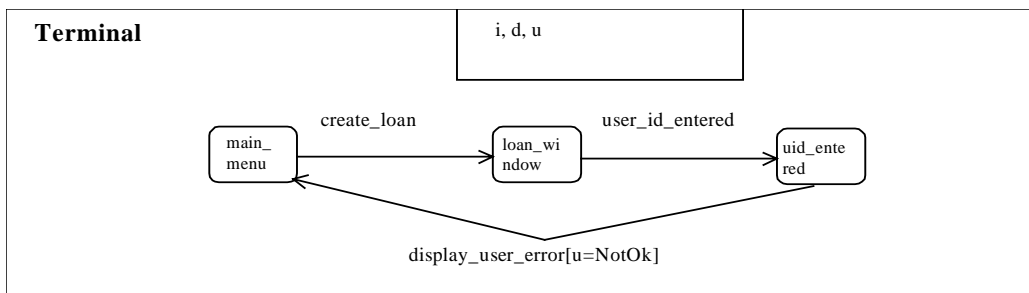


Figure 5c: the modified StateD for object Terminal corresponding to the StateD in Figure 4c.

3.4 Object specification integration

This activity represents the most important part of our approach. Indeed, we have defined a new algorithm for merging scenarios [KEK98] based on Statechart formalism. As the algorithm is incremental, we will discuss here the

integration of the two StateDs and we will refer for illustration to StateDs of object Terminal (StateD1 in figure 5a and StateD2 in figure 5b).

The integration algorithm is a process of five steps:

- validation of state names
- incorporation of scenario variables
- integration of state variable lists
- integration of substate lists
- integration of transition lists.

Step 1 consists to validate state names introduced by the analyst in the activity three of our approach (see section 3.3). It checks the absence of conflicts between StateDs (a conflict occurs when one state appears in the two StateDs at different levels of hierarchy). Step 2 incorporates composition variables in each StateD. The goal of composition variables is to solve the problem of interleaving between scenarios (see below). Step 3 merges state variables lists of the two StateDs into one. Step 4 merges hierarchically states of the two StateDs. Two kinds of merging are considered: an or-merging and an and-merging. An or-merging, as at Terminal state (the high level of StateD1 and StateD2) of Figure 6, occurs when one state has the same initial substates in the two StateDs. An and-merging, as at processing-document-list state of Figure 6, occurs when one state has different initial substates in the two StateDs. Figure 6 shows the result of algorithm application on StateD1 and StateD2. Finally, step 5 consists to integrate transition lists of the two StateDs into one transition list of the resultant StateD. Let transList1 be the transition list of StateD1 and transList2 the transition list of StateD2. For each transition trans2 in transList2, the algorithm looks for a transition trans1 in transList1 which has the same triple fromNode state, toNode state and the event field. If trans1 does not exist, trans2 is added to transList1. Otherwise this step checks the guardCondition and action parts of trans1 and trans2. Three cases have to be considered. The first case is when trans1 and trans2 have the same guardCondition and different action parts, the algorithm adds trans2 to transList1 and outputs a message indicating that the resultant StateD will have a non-deterministic behavior. The second case is when trans1 and trans2 have different guardCondition fields and same action parts, guardCondition field of trans1 becomes equal to [trans1{guardCondition} OR trans2{guardCondition}]. The last case occurs when trans1 and trans2 have different guardCondition fields and different action parts, trans2 is added to transList1. The transition list of the resultant StateD is therefore transList1.

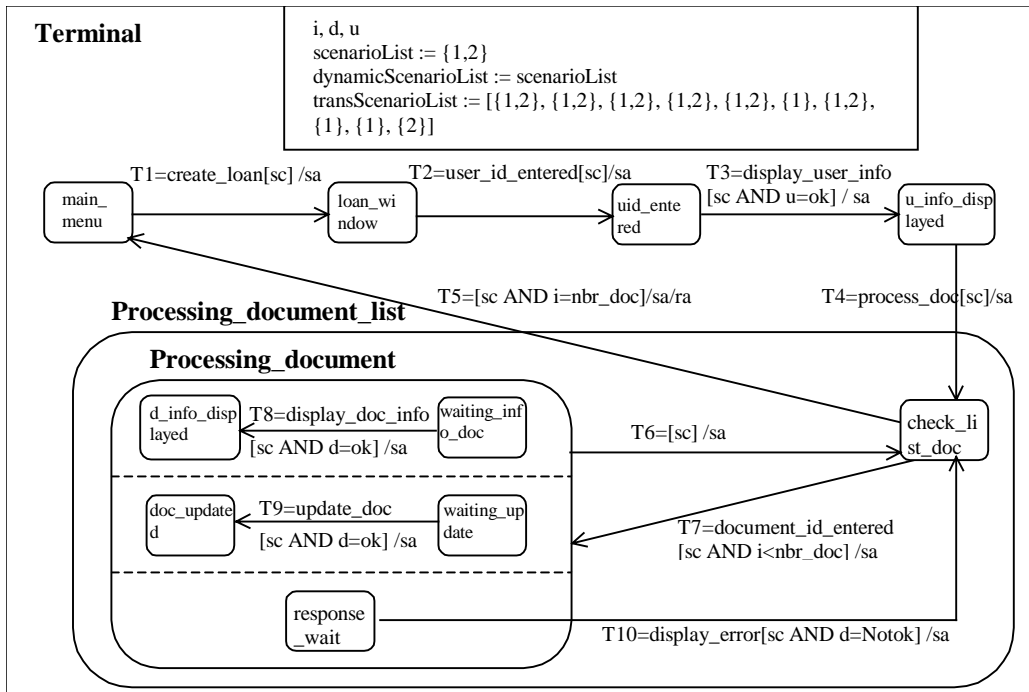


Figure 6: the resultant StateD for the Terminal object after integration of StateD1 and StateD2.

In Figure 7, we show the resultant StateD of object Terminal after integration of the three scenarios.

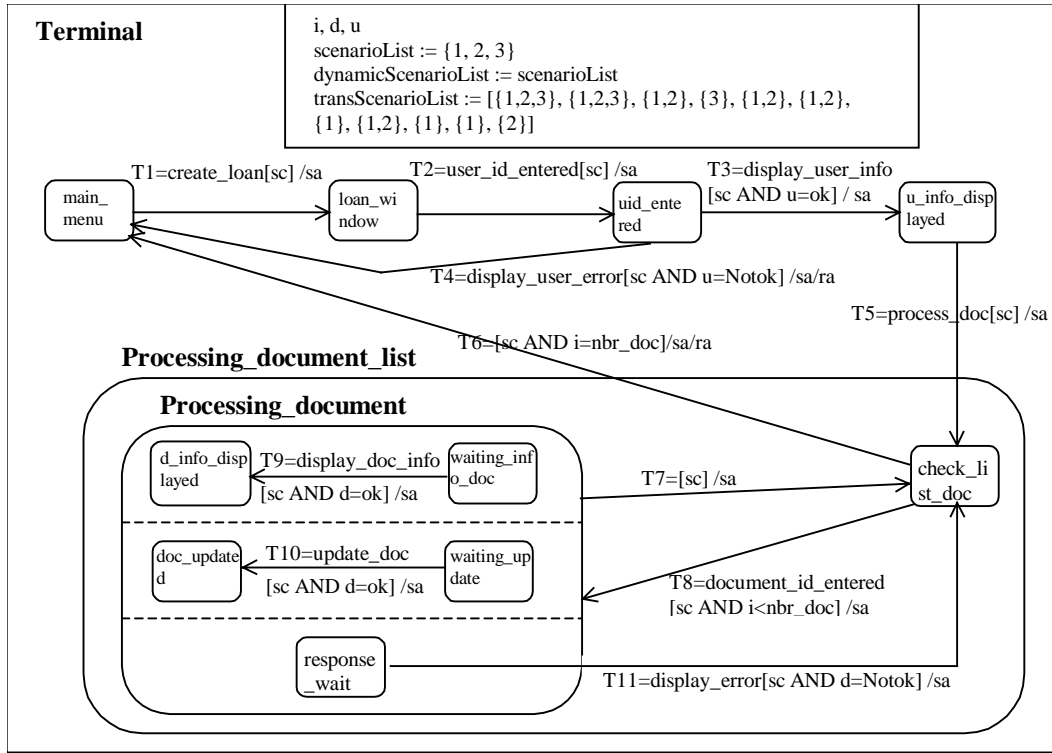


Figure 7: the resultant StateD for the Terminal object after integration of the three scenarios.

Technique for avoiding interleaving problem

To solve the problem of interleaving between scenarios in a StateD, we have defined three composition variables: scenarioList, dynamicScenarioList and transScenarioList. scenarioList is a set of scenario names, it keeps scenario names that the StateD captures. dynamicScenarioList is also a set of scenario names. It is initialized to scenarioList and can change during the execution of the StateD. At each time of execution, it saves scenario names that remain possible in the next execution. transScenarioList is an array of sets of scenario names. It keeps the scenario names concerned by each transition of the StateD.

For each transition in a StateD, we introduce a special condition sc which is equal to $[(transScenarioList[tr] \leftrightarrow dynamicScenarioList) \neq \emptyset]$ (tr is the index of a transition); and a special action sa which is equal to $dynamicScenarioList := dynamicScenarioList \leftrightarrow transScenarioList[tr]$ excepting for transitions that end one scenario where we introduce a re-initialization action ra which is equal to $dynamicScenarioList := scenarioList$.

In Figure 7, the StateD of object Terminal can never execute for example the scenario with the sequence [T1, T2, T3, T5, T8, T9, T11, T6] which is different of the three input scenarios.

Note that, in this work, we are more interested in generating specifications capturing exactly the input scenarios, rather than focusing in verification aspects like coherence and completeness that we plan to study in the future work.

4 Related Work

In the area of scenario integration, most research has only addressed the problem of integration of sequential scenario and few researchers have been interested in a more general form of integration. In this section, we will describe some of these works pointing their advantages and weakness.

Koskimies et al.[KSTM98] present an algorithm SMS (state machine synthesis) for synthesizing state machines (statechart diagram) from a set of scenarios. He has addressed synthesis as an inductive problem and he has based his algorithm on Biermann’s method [Bie76]. The main idea behind SMS is to infer a statechart diagram able to execute all the given input traces. The SMS algorithm can not be used within concurrent systems and the resultant state machine presents no structural informations like hierarchy.

Desharnais et al. [DKFM97] define a scenario as the union of two relations R_e and R_s where R_e represents the relation of the environment which captures all the possible actions of the environment and R_s the relation corresponding to the system reaction. The scenario integration is given by the composition of the scenarios relations. This approach uses Z notation to represent scenario relations (R_e and R_s) and have the same weakness of the previous one. It does not support neither hierarchy nor concurrency.

Glinz [Gli95] gives a way for composing scenarios represented by StateCharts using some operators (conditional, iterative and concurrent). Like he considers a scenario as a use case, he does not give a method to solve the semantic integration problems that arise when separate scenarios have to be brought together (scenario overlapping). In his approach, hierarchy and concurrency are well supported, but he represents each use case of the system by only one scenario. So his approach is more related to use cases composition and not to scenarios integration.

Somé et al. [SDV95] represent scenarios as timed automaton. In his work, he is more interested by timing issues in describing scenarios. He defines a scenario integration algorithm to generate a timed automaton modeling the behavior of the entire system. The generated specification does not present hierarchy and concurrency features.

Dano et al. [DBB97] have proposed a formalization of scenarios with Petri nets, he composes them using a list of temporal relations (begin at the same time, end at the same time, before, just before, etc.). Since he bases his approach on Petri nets, he addresses the problem of concurrency between scenarios.

Citrin et al. [CCKH95] give a formalism for Temporal Message-flow Diagrams (TMFDs) which are similar to sequence diagrams. They provide a set of tools, called collectively Cara, that support many aspects of using TMFDs in developing communicating systems such as edition, simulation and derivation of rule-based specifications from TMFDs. As specifications generated are rule-based, concurrency and hierarchy are not supported.

Elkoutbi et Keller [EK98] give an approach based on colored Petri nets. They also address problems of concurrency, hierarchy and scenarios interleaving. But in their work the hierarchy is only limited to two levels (use case level and scenario level).

Excepting the last work, all researchers do not address the problem of interleaving between scenarios.

5 Discussion of Approach

Below we will discuss our approach in respect to some interesting aspects: evaluation of the approach, state name based integration, interleaving problem and relevance of the approach in the development process.

Validation of approach

The two algorithms which constitute the basis of our approach have been implemented in the Java language. For the validation purposes, we have adopted a textual format for scenario acquisition and specification visualization (while waiting to develop graphical editors for CollIDs and StateDs). Note that the two algorithms have a polynomial complexity.

Our approach has been successfully applied to a number of system examples such as the library system presented in this paper, a gas station simulator [CAB+94], an ATM (Automatic Teller Machine) system [RBP+91], and a filing system [Der96].

State name based integration

We have seen that our integration algorithm is based on state names. This way of integration has been chosen after studying other possibilities such as changing the CTS algorithm to become incremental. But we have found that even if we introduce additional informations like relations between scenarios, the problem of overlapping between scenarios remains unsolved. Recall that we were interested in a more general form of integration.

Moreover, informations included in CollIDs do not enable us to generate labelled StateDs. One solution consists on extending notation of CollIDs with state names of Objects in events. We have not adopted this solution for two main reasons. Firstly, we do not prefer to extend the notation of UML without an effective benefit of expressiveness. Secondly, we think that adding object states in CollIDs will change considerably the purpose of these diagrams.

Problem of interleaving between scenarios

In this work, we have solved the problem of interleaving between scenarios by defining and introducing the composition variables without changing syntax and semantic of UML statechart diagrams.

Sometimes, the interleaving is needed to capture scenarios that are not already considered. In this case, our integration algorithm can easily (by executing or not step 2 of the algorithm) give a choice to avoid or to allow interleaving between scenarios.

Relevance of approach in the development process

Current object-oriented CASE tools support various graphical notations for modeling a system from different views but lack the possibility of automatic transformations between models. The incorporation of our work in a such CASE

tool will make easier the activity of dynamic modeling. In fact, a CASE tool can use our algorithms to generate object StateDs from scenarios acquired as CollDs using its graphical editor.

Furthermore, our incremental algorithm of integration enables us to have an iterative process for a dynamic modeling. In case of changes in some scenarios that have already been integrated, new partial labelled StateDs are generated after reapplication of activity two and three of our approach. The integration algorithm (activity four) is then reapplied over partial labelled StateDs corresponding to the unchanged scenarios and the new ones. Note that the partial labelled StateDs are always kept even after integration.

6 Conclusion and Future Work

The work presented here proposes a new approach based on UML for generating system specification from scenarios. Scenarios are acquired as collaboration diagrams. These collaboration diagrams are transformed into partial object specifications through an existing algorithm. Then, these partial object specifications are merged using a new algorithm that we have defined.

The main interesting features of our approach can be summarized in two points. The first point concerns the kind of integrated scenarios which are, in our case, more general and addresses not only sequential scenarios but also scenarios that exhibit concurrent behaviour. The second point consists in solving the problem of interleaving between scenarios in the resultant specification.

As future work, we aim to provide an automatic support for verification aspects such as coherence and completeness in scenarios. We will extend our approach in order to generate user interface prototypes from object specifications. Also, we plan to develop graphical editors for statechart diagrams and collaboration diagrams, to ease scenario acquisition and allow for the visualization of the generated behavior specifications.

References

- [AD93] John S. Anderson and Brian Durney. *Using Scenario in Deficiency-driven Requirements Engineering*. In Requirements Engineering'93, IEEE Computer Society Press, 134-141, 1993.
- [BK76] Alan W. Biermann and Ramachandran Krishnaswamy. *Constructing programs from example computations*. IEEE Transactions on Software Engineering, 2(3):141-153, March 1976.
- [Boo94] Grady Booch. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994. Second edition.
- [CCKH95] Wayne Citrin, Alistair Cockburn, Jürg V. Käne and Rainer Hauser. *Using Formalized Temporal Message-flow Diagrams*. Software-Practice & Experience, 25(12):1367-1401, December 1995.
- [CAB+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, Inc., 1994.
- [DBB97] B. Dano, H. Briand and F. Barbier. *An Approach Based on the Concept of Use Cases to Produce Dynamic Object-Oriented Specifications*. In Proceedings of the Third IEEE International Symposium on Requirements Engineering, 1997.
- [Der96] Kurt W. Derr. *Applying OMT: A practical step-by-step guide to using the Object Modelling Technique*. SIGS BOOKS/Prentice Hall, 1996.
- [DKFM97] Jules Desharnais, Ridha Khédri, Marc Frappier and Ali Mili. *Integration of sequential scenarios*. In Sixth European Software Engineering Conference:310-326, september 1997.
- [EK98] Mohammed Elkoutbi and Rudolf K. Keller. *Modelling Interactive Systems with Hierarchical Petri Nets*. To appear in a High-Performance Computing Workshop of Advanced Simulation Technology Conference, Boston, april 1998.
- [Gli95] Martin Glinz. *An integrated formal model of scenarios based on statecharts*. In Fifth European Software Engineering Conference, (1989) of Lecture Notes in Computer Science:254-271, 1995.
- [Har87] David Harel. *Statecharts: A visual formalism for complex systems*. Science of Computer Programming, 8:231-274, June 1987.
- [JCJO92] I. Jacobson, M. Christerson, P. Jonson, and G. Overgaard. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, 1992.

<<UML>>'98

- [KEK98] Ismaïl Khriess, Mohammed Elkoutbi and Rudolf K. Keller. *A New Approach to the Synthesis of Behavioral Specifications from Scenarios*. Technical Report GELO-82, Université de Montréal, Montréal, Québec, Canada, January 1998.
- [KSTM98] Kai Koskimies, Tarja Systa, Jyrki Tuomi and Tatu Mannisto. *Automatic support for modeling oo software*. *IEEE Software*, 15(1):42–50, January/February 1998.
- [Nar92] Bonnie A. Nardi. *The Use Of Scenarios In Design*. SIGCHI Bulletin, 24(4), october 1992.
- [PAT94] Colin Potts, Kenji Takahashi and Annie Anton. *Inquiry-Based Scenario Analysis of System Requirements*. Technical Report GIT-CC-94/14, Georgia Institute of Technology, 1994.
- [RAT98] Rational Software Corporation. *Rational Objectory Process 4.1 - Your UML Process*. Santa Clara, CA, February 1998.
- [RMH+97] Rational Software Corporation , Microsoft, Hewlett-Packard , Oracle, Sterling, MCI, Unisys, ICON , IntelliCorp, i-Logix, IBM, ObjecTime, Platinum, Ptech, Taskon, Reich Technologies, and Softeam. *UML notation guide*, version 1.1, Rational Software Corporation, Santa Clara, CA, Spetember 1997.
- [RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., 1991.
- [RG92] Kenneth S. Rubin and Adele Goldberg. *Object Behavior Analysis*. *Communications of the ACM*, 35(9):48-62, september 1992.
- [SDV96] Stéphane Somé, Rachida Dssouli, and Jean Vaucher. *Toward an automation of requirements engineering using scenarios*. *Journal of Computing and Information (JCI)*, 2(1):1110–1132, January 1996.
- [SK97] Siegfried Schönberger and Rudolf K. Keller. *Algorithmic Support for Transformations in Object-Oriented Software Development*. Technical Report DIRO-1080, Université de Montréal, Montréal, Québec, Canada, August 1997. Submitted for publication.