2001 Society for Design and Process Science
Printed in the United States of America

AUTOMATIC SYNTHESIS OF BEHAVIORAL OBJECT SPECIFICATIONS FROM SCENARIOS

Ismaïl Khriss
Codagen Technologies Corporation
Montréal, Québec, Canada

Mohammed Elkoutbi
Rabat, Morocco
École Nationale Supérieure d'Informatique
et d'Analyse des Systèmes

Rudolf K. Keller
Zühlke Engineering AG
Schlieren, Switzerland

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 has become a standard 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 automatically generates from a given set of Collaboration diagrams the Statechart diagrams of all the objects involved. An automatic analysis of specifications in respect to consistency and completeness is also provided. Our approach is incremental and is fully compliant with the UML. Furthermore, it provides an elegant solution to the problem of scenario interleaving. The underlying algorithms have been implemented and validated with several examples, and they are fit for integration into CASE tools supporting the UML.

1. Introduction

Over the past years, scenarios have received significant attention and have been used for different purposes such as understanding (Caroll and Rosson, 1992; Potts et al.,1994), human computer interaction analysis (Monk and Wright, 1990; Nardi, 1992), specification or prototype generation (Angluin, 1987;

This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (Natural Sciences and Engineering Research Council of Canada), and NRC (National Research Council Canada). The research was conducted when Rudolf K. Keller was a full-time faculty member at University of Montreal.

Caroll and Rosson, 1992), and object-oriented (OO) analysis and design (Booch, 1994; Jacobson et al., 1992; Rubin et al., 1992; Rumbaugh et al., 1991).

A typical process for requirements engineering based on scenarios is composed of five steps: scenario acquisition, specification generation, specification verification, prototype generation, and specification validation. The analyst first begins by acquiring scenarios from end users. Secondly, a specification that describes a system behavior is generated from scenarios. Thirdly, the analyst verifies the specification to uncover inconsistencies and incompleteness in scenarios. In case of errors, he or she returns to the first step. Fourthly, a prototype of the expected system is constructed on the basis of the scenarios acquired. Finally, in the fifth step, the prototype is used to validate the scenarios with end users. In case of invalid scenarios, the analyst returns to the first step and repeat steps until the validation of scenarios. As long as the process is not supported by automated tools, it remains tedious, time-consuming, and error prone.

OO analysis and design methods offer a good framework for scenarios. In our work, we have adopted the Unified Modeling Language, which is emerging as a unified notation for OO analysis and design. It directly unifies the methods of Booch (1994), Rumbaugh et al (OMT) (1991), and Jacobson et al. (OOSE) (1992).

In this paper, we propose an incremental and automatic approach to support the first three steps[1] of the requirement engineering process. Our work, in contrast to others such as (Koskimies et al., 1998), supports UML Collaboration diagrams with all their facets (iteration, condition, and concurrency) for scenario acquisition and leverages the expressiveness of UML Statechart diagrams (concurrency and hierarchy) for capturing the resultant specifications.

We also resolve the problem of interleaving between scenarios, which means that the generated specifications will capture exactly the behavior given in the input scenarios. For example, if two input scenarios share a common state, the resultant specification may capture more than the two given scenarios: for instance, execution may initially follow the first scenario; when it reaches the common state, execution may continue following the second scenario. Our approach precludes the generation of such overly general specifications.

The integration of our approach into CASE tools that support the UML will make these tools more powerful. The tedious activities of the process of requirements engineering will become a mostly automatic, tool-supported activity.

Organization of the paper

Section 2 gives a brief overview of the UML diagrams relevant for our work and introduces a running example. Section 3 presents an overview of the four activities of our approach. Section 4 describes in detail the algorithm underlying the third activity, and Section 5 details the algorithm of the fourth activity. Section 6 discusses the issue of consistency and completeness of scenarios supported by our approach. Section 7 addresses related work. Section 8 discusses several aspects of our work. Finally, Section 9 provides some concluding remarks and points out future work.

2. Unified Modeling Language

The UML (Rumbaugh et al., 1999) is an expressive language that can be used for problem conceptualization, software system specification as well as implementation. It covers a wide range of issues from use cases and scenarios to state behavior and operation declaration. The UML provides a syntactic notation to describe all major view of a system using different kinds of diagrams. In this section,

[1] Note that a system prototype can also be generated by an extension of our approach (see [11]).

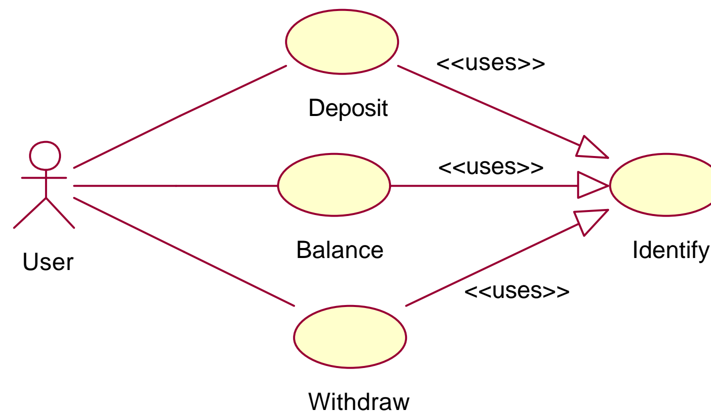

Fig. 1 UsecaseD of the ATM system.

we first discuss the UML diagrams that are relevant for our approach: Use case diagram (UsecaseD), Class diagram (ClassD), Collaboration diagram (CollD), and Statechart diagram (StateD). We conclude the section with an overview of the Object Constraint Language (OCL), which was adopted by the UML for capturing constraints. In our approach, OCL is used for complementing ClassDs. As a running example, we have chosen to study a part of an extended version of the Automatic Teller Machine (ATM) described in (Rumbaugh et al., 1991).

2.1. Use Case diagram

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 dashed line. The direction of a *uses* relation does not imply any order of execution. Figure 1 shows an example of a UsecaseD corresponding to the ATM system. In this UsecaseD, we find one actor ('User') interacting with four use cases ('Identify', 'Withdraw', 'Deposit', and 'Balance'). There are also several *uses* relations, for instance, the use case 'Withdraw' uses the services of the 'Identify' uses case.

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.

2.2. Class diagram

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 classes. Relationships include inheritance, association, and aggregation. The ClassD is the central diagram of a UML model. Figure 2 depicts the ClassD for the ATM system.

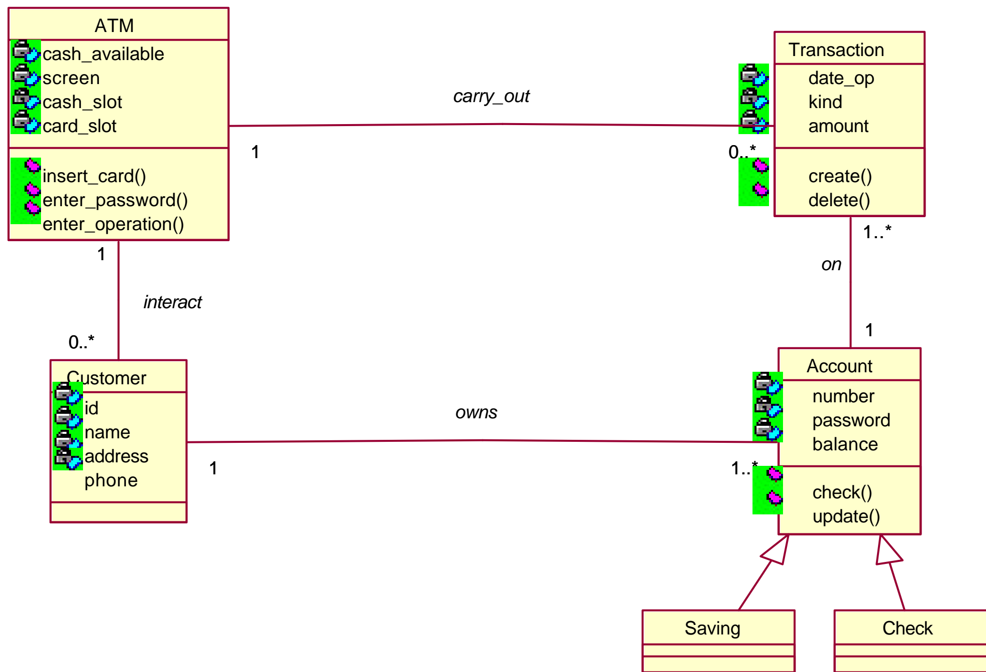

Fig. 2 ClassD of the ATM system.

2.3. Collaboration diagram

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. Conversion from one to the other is possible. For our work, we chose to use CollDs because the UML specification defines them more precisely than SequenceDs. A SequenceD shows interactions among a set of objects in temporal order, which is good for understanding timing issues.

A CollD concentrates on the structure of the interaction between objects and their inter-relationships rather than on the temporal dimensions of a scenario. 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. To a given edge, multiple messages in both directions can be attached.

Each message label includes a sequence number representing the nested procedural calling sequence throughout the scenario, and the message signature. Sequence numbers contain a list of sequence elements separated by dots. Each sequence element consists of a number of parts, such as:

- a compulsory number showing the sequential position of the message, and
- a letter indicating a concurrent thread (see messages 8a and 8b in Figure 3(a)), and
- an iteration indicator * indicating that several messages of the same form are sent sequentially to a single target or concurrently to a set of targets.

Figures 3(a) and 3(b) depict two scenarios (CollDs) of the use case 'Withdraw'. Figure 3(a) represents the scenario where the withdrawal is correctly registered ('regularWithdraw'), and Figure 3(b) represents the case where the balance account is not sufficient ('balanceError').

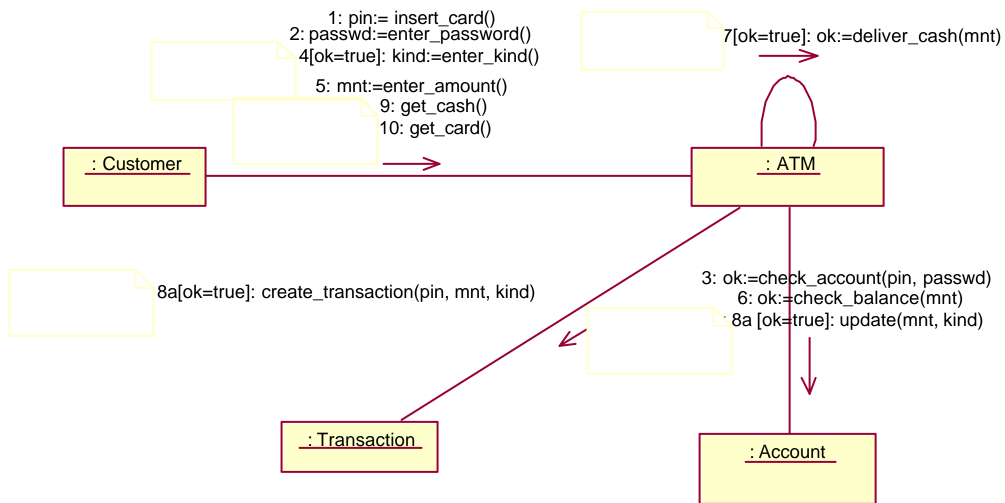

Fig. 3(a) Scenario regularWithdraw of the use case Withdraw.

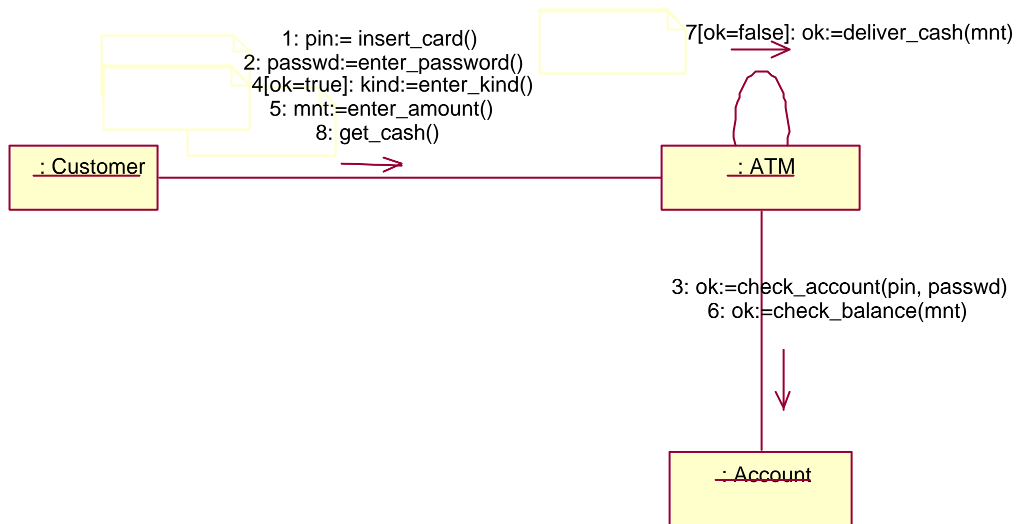

Fig. 3(b) Scenario regularWithdraw of the use case Withdraw.

2.4. Statechart diagram

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 (1987). 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 enters 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 are possible through events. An example of StateD is shown in Figure 6(a).

2.5. Object Constraint Language

OCL offers UML modelers a means to describe a system more accurately than with diagrams alone. OCL is a language in which one can write constraints that contain extra information or restrictions to UML diagrams. Constraints are semantic conditions on UML model elements.

The OCL was originally developed by IBM and subsequently adopted by the Object Management Group (OMG) as a part of the UML specification. It is intended to be simple to read and write and easy to use for non-programmers. The principles of OCL are based on set theory and first order logic, and many of its concepts borrowed from the formal specification language Z (Wordsworth, 1992). OCL has a number of fundamental datatypes (such as boolean, string, and numeric) and collection types that are useful when working with lists of objects.

OCL can be used to specify class invariants, to describe event guard conditions and pre/post class methods. Furthermore, OCL makes navigation through the class model easy and controllable. In our approach, OCL is used for enriching ClassDs with pre-and post-condition of class methods (see Figure 6).

3. Overview of the Approach

In this section, we describe the overall process to derive a system behavior specification. This process provides an automatic way to transform requirements to a formal specification. We consider that the behavior specification of system is given by the behavior specifications of its constituent objects. The approach we define here consists of four major activities (see Figure 4):

(1) Requirement acquisition
(2) Generation of partial object specifications
(3) Analysis of partial object specifications
(4) Integration of partial object specifications.

3.1. Requirement acquisition

Scenario modelling is the key technique mostly used in this activity. It is used in OO methodologies (Booch, 1994; Jacobson et al., 1992; and Rumbaugh et al., 1991) as an approach to requirements engineering. The UML proposes a suitable framework for scenarios acquisition using UsecaseD for capturing system functionalities and SequenceDs or CollDs for describing scenarios.

In this activity, the analyst first elaborates the UsecaseD for the system (see Figure 1). Secondly, he or she elaborates the ClassD of the system (see Figure 2), and for each class of the ClassD, a detailed analysis is done by identifying attributes and methods and defining pre- and post-conditions. An example of a detailed class analysis is given in Figure 5. Finally, the analyst acquires scenarios as CollDs for each use case in the UsecaseD. Figures 3a, and 3b show two sample CollDs corresponding to the use case 'Withdraw' of the ATM system.

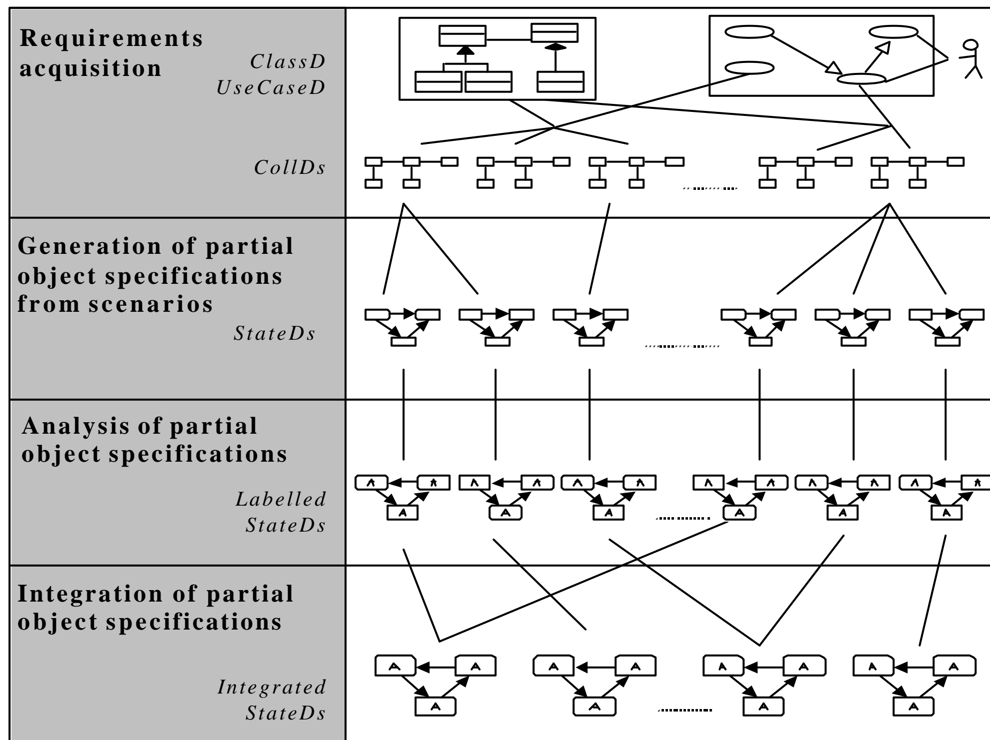

Fig. 4 Overview of the approach.

3.2. Generation of partial object specifications

In this activity, we apply repeatedly on each system CollD the GPS algorithm (Generation of Partial StateDs) (Schönberger et al., 2001) in order to generate automatically partial specifications for all the objects participating in the input scenarios.

Transforming one CollD into StateDs is a process of five sub-steps. Sub-step 1 creates a StateD for every distinct class implied by the objects in the CollD. Sub-step 2 introduces as state variables all variables that are not attributes of the objects of CollD. Sub-step 3 creates transitions for the objects from which messages are sent. Sub-step 4 creates transitions for the objects to which messages are sent. Finally, sub-step 5 brings 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 CollD: iteration messages, conditional messages, concurrent messages and messages with multiple predecessors. Applying the GPS algorithm to the scenarios 'regularWithdraw' and 'balanceError', we obtain for the object ATM the partial StateDs shown in figures 6(a) and 6(b), respectively.

3.3. Analysis of partial object specifications

The partial StateDs generated in the previous activity are unlabelled, i.e., their states do not carry names. However, the IPS algorithm (see next section) is state based, requiring labelled StateDs as input. Furthermore, scenarios acquired may contain errors because of inconsistencies and internal incompleteness. Thus, the objective of this activity is to check the consistency of the input scenarios and to obtain labelled StateDs. This is achieved by a new algorithm, called APS (Analysis of Partial StateDs), based on pre- and post-conditions of class methods (cf. Figure 5). This algorithm is detailed in Section 4.

ATM

cash_available: boolean = true
screen: string = “main”
cash_slot: string = “closed”
card_slot: string = “empty”

insert_card(): string
pre: screen=”main” and cash_slot=”closed” and card_slot=”empty”
post: screen=”enter password” and cash_slot=”closed” and card_slot=”full”

enter_password(): string
pre: screen=”enter password” and cash_slot=”closed” and card_slot=”full”
post: (screen=”enter kind” or screen=”password incorrect”) and cash_slot=”closed” and card_slot=”full”

enter_kind(): character
pre: screen=”enter kind” and cash_slot=”closed” and card_slot=”full”
post: (screen=”deposit” or screen=”withdraw”) and cash_slot=”closed” and card_slot=”full”

enter_amount(): real
pre: (cash_available=true and screen=”withdraw” and cash_slot=”closed” and card_slot=”full”) or (screen=”deposit” and cash_slot=”closed” and card_slot=”full”)
post: (cash_available=true and (screen=”withdraw in progress” or screen=”insufficient funds”) and cash_slot=”closed” and card_slot=”full”) or (screen=”deposit in progress” and cash_slot=”closed” and card_slot=”full”)

deliver_cash(mnt: real): boolean
pre: cash_available=true and scre en=”withdraw in progress” and cash_slot=”closed” and card_slot=”full”
post: (screen=”take cash” or screen=”insufficient funds”) and (cash_slot=”opened” or cash_slot=”closed”) and card_slot=”full”

get_cash()
pre: screen=”take cash” and cash_slot=”opened” and card_slot=”full”
post: screen=”take card” and cash_slot=”closed” and card_slot=”ejected”

get_card()
pre: screen=”take card” and cash_slot=”closed” and card_slot=”ejected”
post: screen=”main” and cash_slot=”closed” and card_slot=”empty”

display_error(): boolean
pre: screen=” insufficient funds” and cash_slot=”opened” and card_slot=”full”
post: screen=”take card” and cash_slot=”closed” and card_slot=”ejected”

Figure 5: The ATM class.

Applying the APS algorithm underlying to the StateD of Figure 6(a), we obtain the StateD shown in Figure 7(a), annotated with the labels explained in the legend of the figure. Applying the algorithm to the StateD of Figure 6(b), we obtain the StateD shown in Figure 7(b).

3.4. Integration of partial object specifications

This activity is to integrate for each object of the system all its partial labelled StateDs into one single StateD. The resultant StateD is then verified in respect to its consistency. This activity is achieved incrementally by a new algorithm, which is called IPS (Integration of Partial StateDs). This algorithm is described in Section 5. Figure 8, for instance, shows the resultant StateD of the ATM object after integrating the two scenarios of the use case ‘Withdraw’.

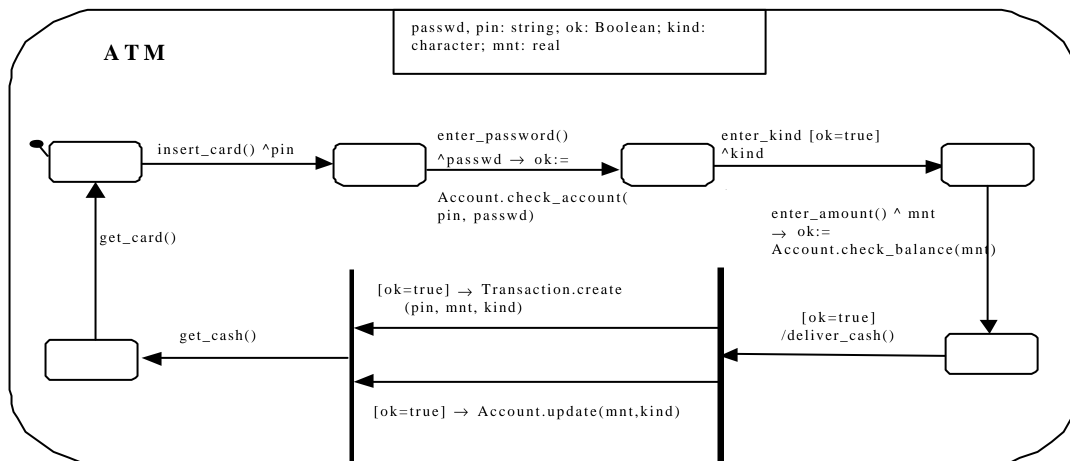

Fig. 6(a) StateD for the object ATM generating by applying the GPS algorithm on the scenario regularWithdraw.

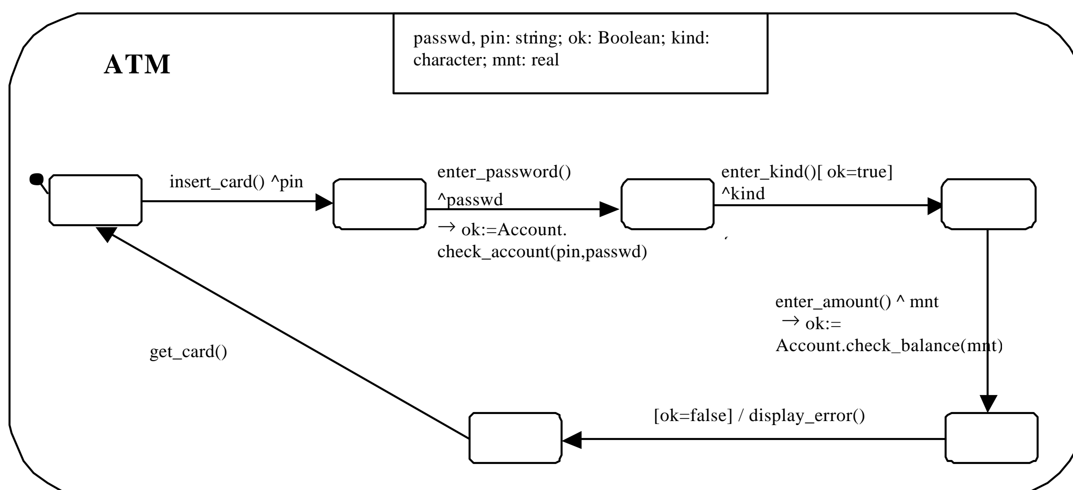

Fig. 6(b) StateD for the object ATM generating by applying the GPS algorithm on the scenario balanceError.

4. Description of the APS Algorithm

The APS algorithm takes as input a description of a class and its unlabeled StateD and outputs a labeled StateD. The algorithm consists of three sub-steps: checking the consistency of the desciption of the class, labeling each state of the unlabeled StateD, and checking the consistency of the resultant StateD. The first and third sub-step will be discussed in Section 6. Before describing the second sub-step of the algorithm, we first introduce the syntax of the pre- and post-condition of class methods.

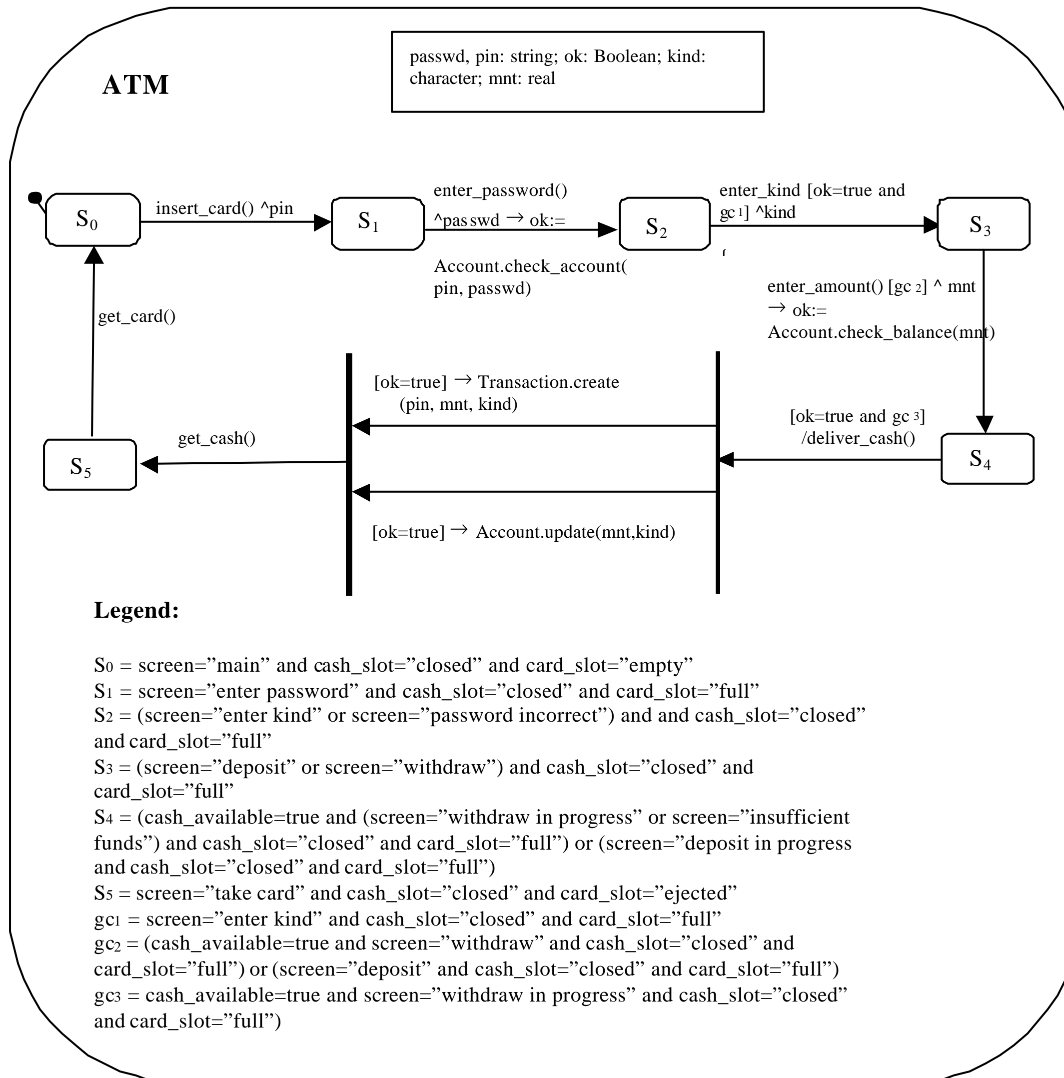

Fig. 7(a) The labeled StateD obtained from the StateD of Figure 6(a).

4.1. Syntax of pre- and post-condition of the class operations

The pre- and post-conditions of class operations are described using OCL. Let *op* be a class operation, and *pre(op)* and *post(op)* be the pre- and post-conditions of that operation, respectively. The conditions are expressed in a disjunctive canonical form, referring to the class attributes. Syntactically, they adhere to the following subset of OCL:

pre(op) : orExpression.
post(op) := (ifExpression {"or" ifExpression}) | orExpression.
ifExpression := "if" orExpression "then" orExpression "endif".
orExpression := andExpression {"or" andExpression}.
andExpression := basicExpression {"and" basicExpression}.
basicExpression := identifier ((("=" | "!=" | "<" | ">" | "<=" | ">=")
(string_literal | character_literal | integer_literal |
floating_point_literal)) | ("=" ("true" | "false")).

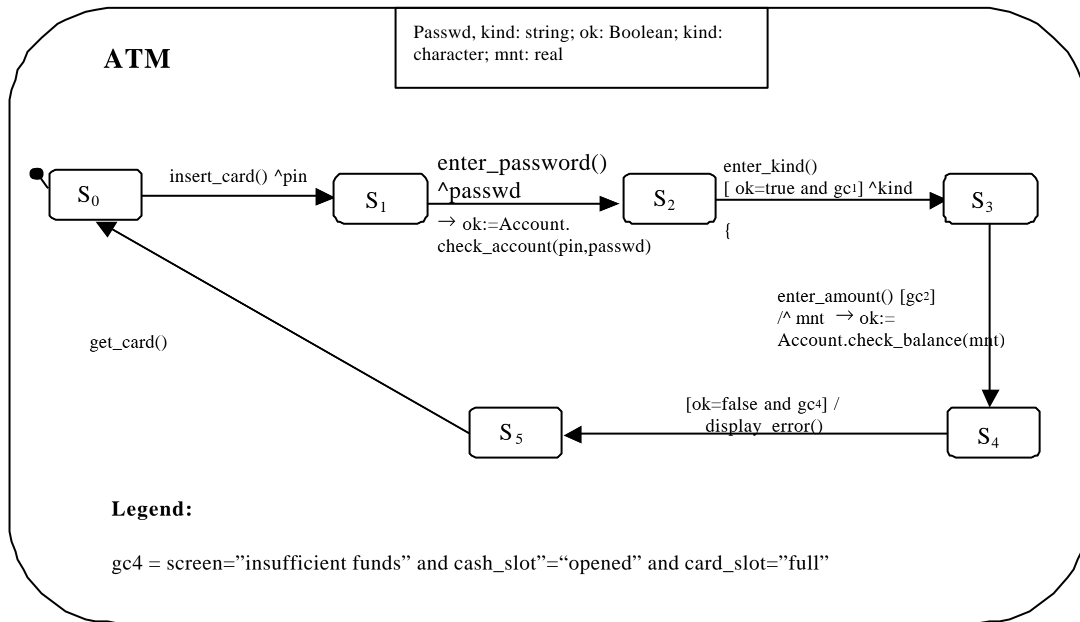

Fig. 7(b) The labeled StateD obtained from the StateD of Figure 6(b).

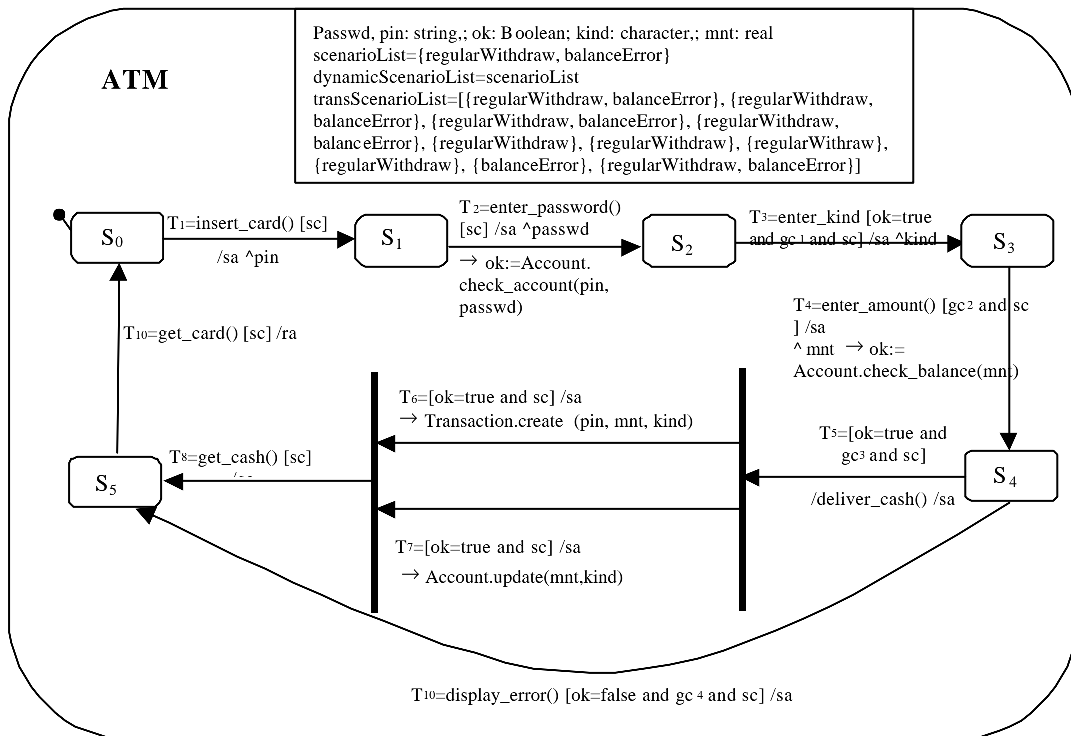

Fig. 8 The resultant StateD for the ATM object after integration of the two scenarios of the use case Withdraw.

```
boolean labelStates(aClass: Class; VAR stateD: StateD)
   begin
      bool := true
      transList := stateD.{transition}
      i := 1
      while (i<size(translist) do
         /* label one transition */
         computeValueOFC1AndC2(aClass, I, stated, c1, c2, backTrackingElements)
         if not (trans.fromNode is a merge bar or a split bar) then
            /* merge bar and split bar are not labeled */
            trans.fromNode.name := c1
         endif
         if not (trans.toNode is a merge bar or a split bar) then
            /* merge bar and split bar are not labeled */
            trans.toNode.name := c2
         endif
         /* end label one transition */
         if (i<=size(transList) - 1) then
            /* check if nextTrans is consistent with its fromNode and toNode states */
            bool := transitionConsistencyChecking(transList[i+1], aClass)
            if (not bool) then
               /* look for the last transition that has still elements in the array
                  backTrackingElements */
                i := lookForIndexOfBackTracking(backTrackingElements, transList)
            else
               i := i + 1
            endif
         endif
      endwhile
      retourn bool
   end labelStates
```

Pseudo-code of the state labeling sub-step.

4.2. Labeling states of the StateD

The objective of the second sub-step is to label each state of the unlabeled StateD. The algorithm consists to traverse the transitions of the StateD (see the pseudo-code below). Note that the transitions are constructed by the GPS algorithm and sorted following the order of message sequencing of the corresponding scenario[2]. For each transition *trans*, the algorithm begins by computing the value of two conditions c_1 and c_2. These conditions constitute the basis of the labeling sub-step. Two cases are considered.

The first case occurs when the post-condition of the event of trans is an *orExpression*, c_1 is equal to the pre-condition of the event of 'trans', c_2 to the post-condition of 'trans'. The second case occurs when the post-condition of the event of 'trans' is equal to *if* pre_1 *then* $post_1$ *endif or .... Or if* pre_n *then* $post_n$ *endif* then the algorithm tries with the first *ifExpression* and c_1 is equal to pre_1 and c_2 to $post_1$. The other *ifExpression* (i.e. *if* pre_2 *then* $post_2$ *endif or ...or if* pre_n *then* $post_n$) are saved in the array *backtrackingElements*. The purpose of backtracking Elements is explained below. The *fromNode* state of 'trans' is labeled with c_1 and *toNode* state with c_2. For instance in Figure 9, the post-condition of the

[2] Concurrent messages are sorted following the alphabetical order of the letters contained in their corresponding sequence numbers. For example if a CollD contains the messages 1, 3, 2a.1, 2b and 2a.2, the messages will be sorted as 1, 2a.1, 2a.2, 2b, 3.

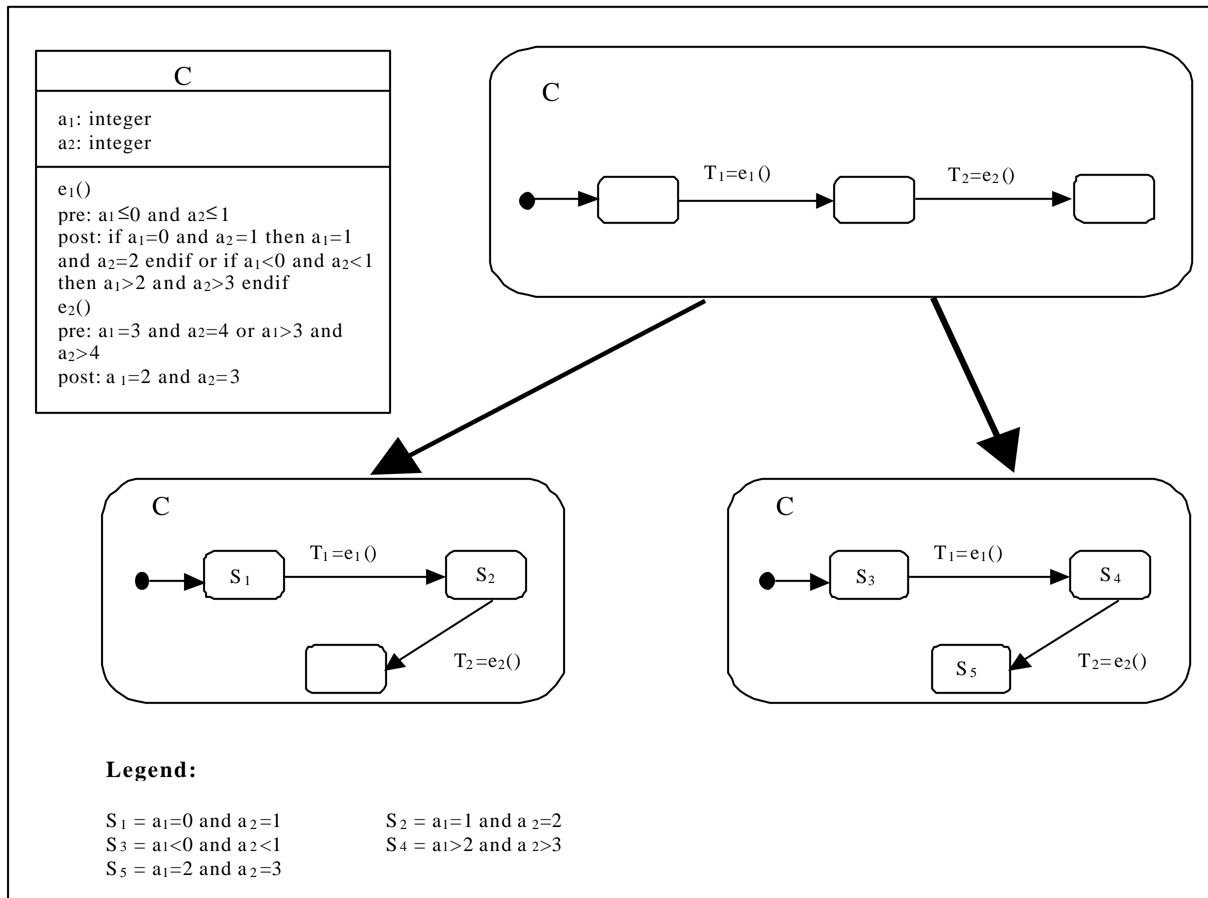

Fig. 9 StateD labeling (a) labeling failed (T_2 is not consistent with its fromNode state) (b) labeling succeded (T_2 is consistent with its fromNode state).

event of 'T_1' (i.e. '$e_1()$') contains an *IfExpression* then c_1 is equal to 'a_1=0 and a_2=1', c_2 to 'a_1=1 and a_2=2'. The *fromNode* state of 'T_1' is equal to c_1 and *toNode* state to c_2.

Then the algorithm verifies whether the next transition 'nextTrans' remains consistent with their *fromNode* and *toNode* states (see definition 2, 4 and 10 in Section 6). If 'nextTrans' is consistent, the algorithm process 'nextTrans' as 'trans'. Otherwise, the algorithm backtracks and looks for the last transition that has still elements to process in the array backTrackingElements. In the example of Figure 9, 'T_2' is not consistent with its *fromNode* state (i.e. 'S_2'). The algorithm backtracks to 'T_1' and retries with another *ifExpression*, c_1 is now equal to 'a_1<0 and a_2<1', c_2 to 'a_1>2 and a_2>3'. The *fromNode* and *toNode* states of 'T_1' becomes repectively 'S_3' and 'S_4'. 'T_4' is now consistent with its *fromNode* (i.e. 'S_4'). The post-condition of the event of 'T_2' (i.e. '$e_2()$') is an *orExpression* then c_1 is equal to 'a_1>2 and a_2>3' and c_2 to 'a_1=2 and a_2=3'. The *fromNode* state of 'T_2' has been already labeled by 'T_1' and its *toNode* state is labeled by c_2 (i.e. 'S_5').

5. Description of the IPS Algorithm

This activity consists to integrate for each object of the system all its partial labeled StateDs into one single StateD. The resultant StateDs are then verified in respect to their consistency. The integration

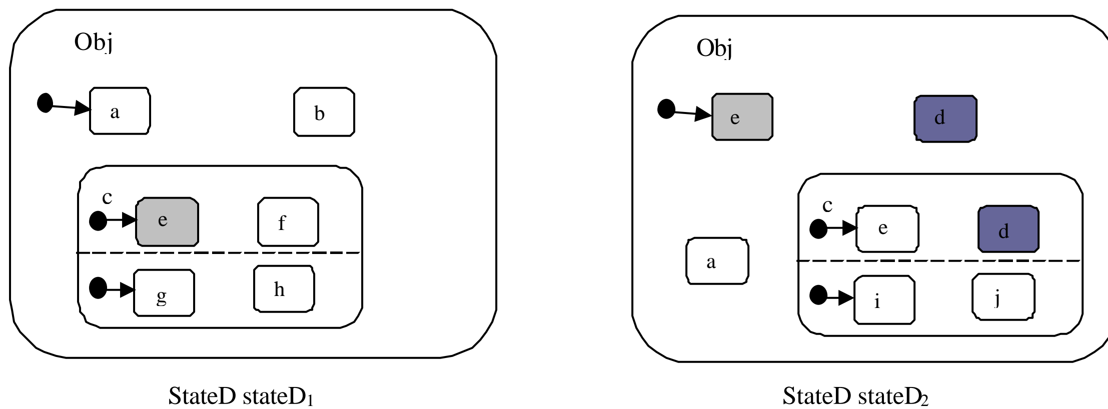

Fig. 10 Examples of detected errors in the state checking sub-step.

algorithm is incremental, and consists of five sub-steps: state checking, state merging, transition merging, suppressing the interleaving problem and checking the consistency of the resultant StateD. The five sub-steps will be discussed in Section 6.

5.1. State checking

Before merging states of two StateDs, the algorithm checks if the same state appears at different levels of hierarchy[3] into the two StateDs. Suppose that the algorithm has to mere the object 'Obj', the StateD 'sd_1' and the StateD 'sd_2' given in Figure 10. The following errors will be detected:

The state d appears at different levels in sd_2 (in levels Obj and c).
The state e is not at the same level in sd_1 (level c) and sd_2 (level Obj).

The analyst must fix the detected errors before continuing the scenario integration activity.

5.2. State merging

When no errors are detected in the state checking sub-step, the algorithm proceeds to merge states of the two StateDs level by level from top to bottom. The state merging sub-step for a given level depends on the type and the initial states (see the pseudo-code below). Three cases are considered:

(1) Case where the same levels in the two StateDs are of type OR (*or-state*): if their initial states are equal, an operation of union between states of these levels is done. This is the case, for example, of level 'c' in 'sd_1' and 'sd_2' of Figure 11, which has the same initial state 'c_1'. If the initial states are distinct, the algorithm merges the two initial states into a state of type AND (*and-state*). This is the case of the level 'Obj' that has a as initial state in 'sd_1' and 'g' in 'sd_2'. An *and-state* is then created as shown in Figure 11(c). For the rest of states, a union operation is performed.

(2) Case where the same levels in the two StateDs are of type AND (and-state): the algorithm performs a union operation between states of the threads that have the same initial states. This is the case of the level 'e' that has the same initial state 'e_1' in 'sd_1' and 'sd_2'. Threads that have

[3] A level is related to a state. The level obj of sd_1 (see Figure 10), which is an or-state , contains the states a, b and c. The level c of sd_1 is an and-state and contains the states e, f, g and h.

```
StateD stateMerging (sd1, sd2: StateD)
   begin
      /* first case */
      if (sd1 is an or-state and sd2 is an or-state) then
         sd1.{substate} := sd1.{substate} ∪ sd2.{substate}
         if (initialState(sd1) != initialState(sd2)) then
            sd1.createANDState(initialState(sd1), initialState(sd2))
         endif
      endif
      /* second and third case */
      if (sd1 is an and-state) then
         threadList1 := sd1.{substate}
         if (sd2 is an and-state) then threadList2 := sd2.{substate}
            else threadList2 := sd2
         endif
         for (i:=1 to threadList2.size()) do
            thread2 := threadList2[i]
            thread1 := threadList1.lookForThreadWithSameInitialState(thread2)
            if (thread1 != null) thread1 := thread1 ∪ thread2
               else threadList1.addThread(thread2)
            endif
         endfor
      endif
      /* process low levels */
      substateList1 := sd1.{substate}
      substateList2 := sd2.{substate}
      for (i:=1 to substateList1.size()) do
         substate1 := substateList1[i]
         substate2 := substateList2.lookForSubstateWithSameName(substate1)
         if (substate1 is a composite state or substate2 is a composite state) then
            stateMerging(substate1, substate2)
         endif
      endfor
```

Pseudo-code of the state merging sub-step.

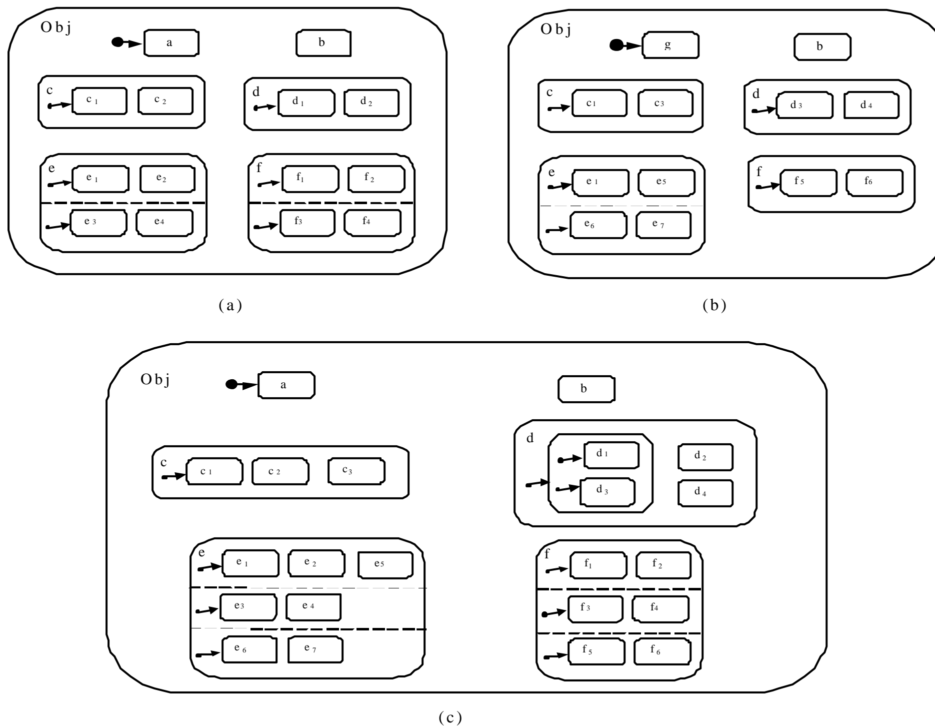

Fig. 11 State merging: (a) StateD sd_1, (b) StateD sd_2, (c) merged StateD sd.

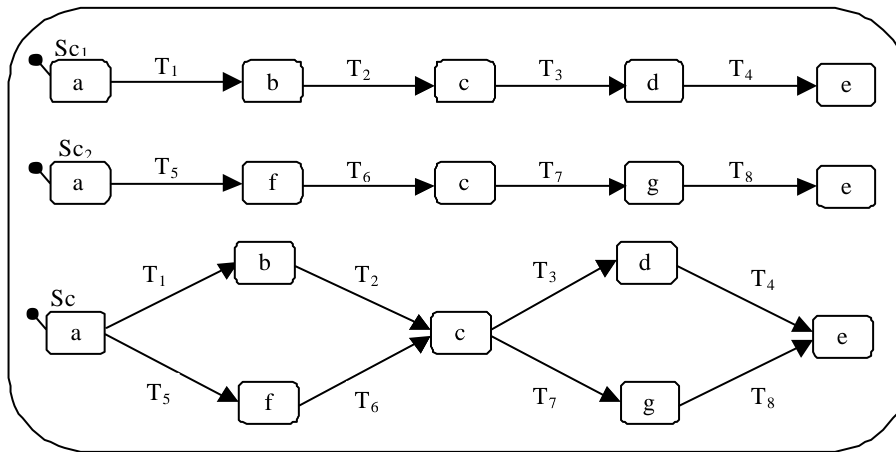

Fig. 12 Interleaving problem between Sc_1 and Sc_2.

not the same initial states are added in the resultant StateD. This is the case of the second threads of level 'e' in 'sd_1' and 'sd_2'.

(3) Case where the same levels in the two StateDs have different types: this case is similar to the previous one. The level of type OR is considered as of type AND with one thread of control. This is the case of the level 'f' that is of type AND in 'sd_1' and of type OR in 'sd_2'. Result of merging is shown in Figure 11(c).

5.3. Transition merging

In this sub-step, the algorithm looks in the two StateDs to be merged for a pair of transitions having the same quintuplet *fromNode*, *toNode*, *event* field, *{/action}*, and *{sendClause}* field. The guard condition of the merged transition becomes the disjunction of the guard conditions of the two conditions.

5.4. Solving the interleaving problem

In general, after integrating several scenarios, the resulting specification will capture more than the initial scenarios. Figure 12 provides an example illustrating this problem (scenarios are represented as StateDs). Suppose we merge the two scenarios 'Sc_1' and 'Sc_2'. Then the resultant specification 'Sc' will not only capture 'Sc_1' and 'Sc_2', but also two new scenarios, corresponding to transaction sequences (T_1, T_2, T_7, T_8) and (T_5, T_6, T_3, T_4), respectively.

To solve this problem, we have defined three composition variables: *scenarioList*, *dynamicScenarioList* and *transScenarioList*. scenarioList is a set of scenario names (see Figure 8), 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]* $\cap$ *dynamicScenarioList)* $\neq \emptyset$*]* (*tr* is the index of a transition); and a special action *sa* which is equal to *dynamicScenarioList:= dynamicScenarioList* $\cap$ *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 8, after the execution of transitions 'T_1', 'T_2', 'T_3' and 'T_4', the dynamicScenarioList variable stays equal to the variable 'scenarioList = {regularWithdraw, balanceError}'. If 'T_9' occurs after 'T_4', the dynamicScenarioList variable will be updated by the action 'sa:= dynamicScenarioList $\cap$ transScenarioList[T_9] = {regularWithdraw, balanceError} $\cap$ {balanceError} = {balanceError}'.

6. Verification Issues

Two major issues arise in the process of the verification of user requirements acquired as scenarios: consistency and completeness. In this section, we discuss how the APS and IPS algorithms address these issues. But first we give a set of definitions that help understand the verification process. As object specifications generated from scenarios are described with StateDs, most of the verification operations are performed on StateDs.

Recall that a state of an object is defined as a condition over its attributes. We also introduce conditions on equality between states and conditions, *or-states* and *and-states* in order to be consistent with the spirit of UML.

6.1. Definitions

Definition 1. Let c be a condition of type orExpression over a set of variables[4] v_i. Eval(c) is a function that returns a set of tuples of values of the variables v_i that verifies the condition c.

Definition 2. Two conditions of type orExpression c_1 and c_2 are equal if and only if Eval(c_1) is equal to Eval(c_2).

Definition 3. Two states s_1 and s_2 are equal if and only if their conditions c(s_1) and c(s_2) are equal.

Definition 4. A condition of type orExpression c_1 refines a condition of type orExpression c_2 if and only if Eval(c_1) $\subseteq$ Eval(c_2).

Definition 5. A state s is an or-state if and only if "ss_i sub-state of s: ss_i verifies the conditions:

(1) $\forall i, 1 \leq i \leq n$ c(ss_i) refines c(s) and

(2) $\forall i, j, 1 \leq i \leq n, 1 \leq j \leq n$ and $i \neq j$: Eval(ss_i) $\cap$ Eval(ss_j) = $\emptyset$.

Definition 6. A state s is an and-state if and only if s contains n concurrent sub-states ss_k ($n \geq 2$) such as $\forall i, j, 1 \leq i \leq n, 1 \leq j \leq n$ and $i \neq j$: c(ss_i) does not refine c(ss_j), c(ss_j) does not refine c(ss_i) and Eval(ss_i) $\cap$ Eval(ss_j) = $\emptyset$.

Definition 7. A condition c of type orExpression is consistent if and only if Eval(c) $\neq \emptyset$.

Definition 8. A condition c of type 'if pre_1 then $post_1$ endif or … or if pre_n then $post_n$ endif' (pre_i and $post_i$, for $1 \leq i \leq n$ are all orExpression) is consistent if and only if $\forall i, 1 \leq i \leq n$, Eval($pre_i$) $\neq \emptyset$ and Eval($post_i$) $\neq \emptyset$.

Definition 9. Let op be an operation of a class C and pre(op) and post(op) are respectively a pre- and post-condition of the operation. post(op) is consistent with pre(op) if and only if post(op) is an orExpression or post(op) is equal to 'if pre_1 then $post_1$ endif or … or if pre_n then $post_n$ endif' (pre_i and $post_i$, for $1 \leq i \leq n$ are all orExpression) and $\forall i, 1 \leq i \leq n$ pre_i refines pre(op).

[4] A variable may be a class attribute, an operation parameter of a class or a variable of a StateD.

Definition 10. Let trans be a condition of a StateD. trans is consistent with its fromNode and toNode states if and only if the pre-conditon of the trans event refines c(fromNode) and c(toNode) refines the post-condition of the trans event.

Definition 11. A StateD exhibits a non-deterministic behavior if and only only if there exists two transitions $trans_1$ equal to '$fromNode_1$ $event_1$ [$guardCondition_1$] {/$action_1$} {$sendClause_1$} ^$returnValue_1$ $toNode_1$, and $trans_2$ equal to '$fromNode_2$ $event_2$ [$guardCondition_2$] {/$action_2$} {$sendClause_2$} ^$returnValue_2$ $toNode_2$' such as c($fromNode_1$) refines c($fromNode_2$) or c($fromNode_2$) refines c($fromNode_1$), $event_1$ = $event_2$, $guardCondition_1$ and $guardCondition_2$ are not exclusive (i.e. Eval($guardCondition_1$) $\cap$ Eval($guardCondition_2$) $\neq \emptyset$) and:

(1) ({/$action_1$} $\neq$ {/$action_2$} or {$sendClause_1$} $\neq$ {$sendClause_2$}) and $toNode_1$ = $toNode_2$, or

(2) ({/$action_1$} = {/$action_2$} or {$sendClause_1$} = {$sendClause_2$}) and $toNode_1 \neq toNode_2$.

Definition 12. A StateD of a class C is consistent if and only if it satisfies the conditions below:

(1) Guard conditions of all its transitions are consistent,
(2) It does not exhibit a non-deterministic behavior,
(3) All its or-states satisfy Defintion 5,
(4) All its and-states satisfy Definition 6.

Proposition 1. If a StateD exhibits a non-deterministic behavior caused by the existence of a set of transitions that satisfies the condition (2) of Definition 12, and the toNode states of these transitions are at the same hierarchical level and satisfy the condition of Definition 6, then the non-determinism can be resolved.

To resolve this non-determinism, we create an *and-state* that contains all *toNode* states of these transitions; and we keep only one transition. The newly created *and-state*, which satisfies Definition 6 becomes the *toNode* state of the kept transition.

6.2. Consistency

Three operations are defined for the verification of scenario consistency: class description consistency, consistency of a transition with its *fromNode* and its *toNode* states, and StateD consistency. Note that when a StateD captures one scenario, checking the consistency of a StateD means that we check whether the corresponding scenario is consistent; and when a StateD integrates several scenarios, we check the consistency between those scenarios.

Class description consistency

The objective of this operation is to check the consistency of the description of a given class (such as in Figure 5). This operation looks for all pre- and post-conditions of the class methods. These conditions must be consistent (see definitions 7 and 9) and the post-condition of each method must be consistent with its pre-condition (see Definition 9). In case of errors, the algorithm invites the analyst to make correction on the description of the given class.

Consistency of a transition with its fromNode and toNode states

This operation checks whether a transition remains consistent with its *fromNode* and *toNode* states (see Definition 10). This kind of inconsistency may be caused by inconsistent or incomplete description of a given scenario. In the two cases, the algorithm invites the analyst to make correction on the description of the scenario. For example, if we assume that in the description of the scenario 'regularWithdraw' (see Figure 3(a)), the message 2:passwd:=enter_password()$\rightarrow$ was omitted, the generated StateD of the ATM class would be as described in Figure 13. During the activity of partial object specifications analysis, the APS algorithm would detect that T_2 is not consistent with its *fromNode* state labeled by 'S_1'.

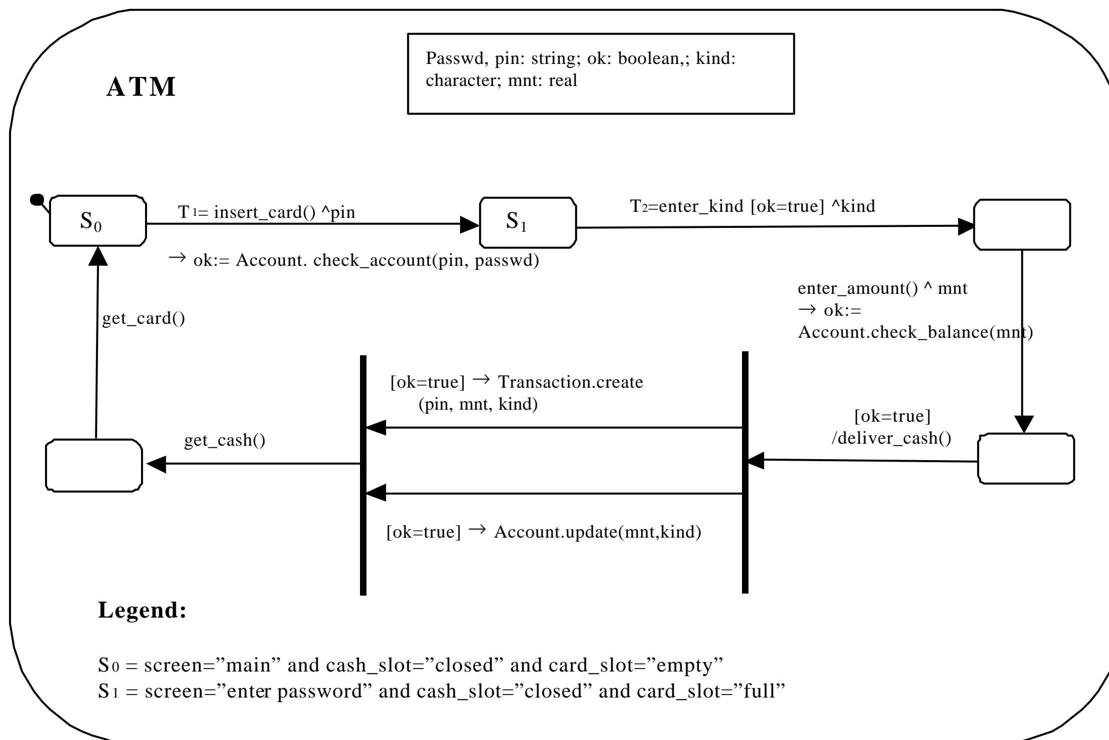

Fig. 13 StateD for the class ATM generated from an altered description of the scenario given in Figure 3(a).

StateD consistency

This operation checks if a given StateD is consistent (see Definition 12). Two cases are considered. If the conditions (1), (3) and (4) have caused the StateD inconsistency then the algorithm invites the analyst to make correction on the scenarios that causes the inconsistency. Otherwise, i.e. the inconsistency is caused by the fact that the StateD is not deterministic, the algorithm proposes when possible (see Proposition 1), a new StateD where the non-determinism is resolved and invites the analyst to confirm the result (since non-determinism often hides dangerous incompleteness).

6.3. Completeness

There are several possible ways to define a complete specification. Indeed, a complete specification can be defined as one that contains all the behaviors required by the users. Another definition is that a complete specification is one that contains all the aspects about the described system even those that are not defined in user requirements. Our approach adheres to the first definition, as specifications are automatically generated from scenarios. Furthermore, a consistency check may reveal incomplete scenarios caused by errors in their descriptions. For the second kind of completeness, we can use guidelines provided by Heimdahl and Leveson (1996). Specifically, they check the completeness of a given specification in respect to a set of criteria related to robustness, that is, a response is specified for every possible input. For a StateD, robustness implies the following rules:

(1) Every state must have a behavior, that is, a transition for every possible input.
(2) The logical OR of the conditions on every transition out of any state must form a tautology.

(3) Every state must have a behavior, i.e. a transition, defined in case there is no input for a given period of time (time-out).

7. Related Work

In this section, we review related work in the area of scenario integration and verification of requirements. Table 1 shows a comparison between our approach and eleven other approaches in respect to the following seven criteria: vision, notation, range of support, degree of support, basis of integration, interleaving problem and verification.

In the realm of OO development the work of Koskimies and Mäkinen (1998), Mäkinen and Systä (2000), and Whittle and Schumann (2000) comes closed to ours. All these approaches aim to derive a set of specifications for the system objects from scenarios, whereas all other approaches are interested in synthesizing one specification for the whole system.

The various approaches differ in the notations that they support for the description of scenarios and specifications. Some, such as Dano et al. (1997) and the SCR method (1998), use a tabular notation for capturing scenarios, whereas others, such as Whittle and Schumann and ourselves, use SequenceDs or CollDs. In contrast, Deharnais et al. (1998) describe scenarios with relations. For the specification description, a variety of notations are used. For example, StateDs are supported by our approach and by Koskimies and Mäkinen, whereas Petri nets are used by Elkoutbi and Keller (2000) and by Lee et al. (1998).

One of the most prominent features of our approach is that it supports many kinds of scenarios (sequential, iterative and concurrent), whereas most of the other approaches can handle only sequential scenarios. Note that Somé et al. (1996) are more interested in timing issues in describing scenarios. These kinds of scenarios are not supported by our approach.

Most of the related approaches are semi-automatic since they require that states are determined manually, whereas in our approach, they are synthesized from the sequence of messages in scenarios as well as from pre- and post-conditions of class operations corresponding to those messages.

Similarly to our approach, most of the other approaches are state-based in the process of scenario integration. In contrast, Koskimies et al. (Koskimies et al., 1998) as well as Mäkinen and Systä (2000) address synthesis as an inductive problem basing their algorithm on the method of Biermann and Krishnaswamy (1976) for the former and on Angluin's framework (1987) for the latter. Desharnais et al. (1998) view scenario integration as a composition of scenario relations. Lee et al. (1998) use Constraint-based Modular Petri Nets for capturing use cases where shared places and transitions are used to synchronize between these use cases.

The problem of scenario interleaving is discussed by Mäkinen and Systä (2000). Elkoutbi and Keller (2000) and ourselves present solutions to that problem.

Finally, many approaches, including our own, support the verification of scenario consistency and completeness. Note that two of these approaches focus on the verification aspect. Heimdahl and Leveson (1996) propose a tool for automatically analyzing state-based requirements for some aspects of completeness and consistency. Thus, their tool can analyze our generated StateDs. Their work scales up to large systems by decomposing the specification into smaller, analyzable parts and then using functional composition rules to ensure that verified properties hold for the entire specification. In contrast, our approach handles the issue of scalability by analyzing each scenario partially, and by analyzing the integrated specification incrementally. Some of the composition rules defined by Heimdahl and Leveson (1996) are also identified in our approach. For instance, a rule for *union composition* stating that *no two transitions out of the same state can be satisfied at the same time,* is also identified by us (see Definition 19). Note

Table 1: Comparison between approaches

Criteria / Approaches	Vision	Notation	Range of support	Degree of support	Basis of integration	Interleaving	Verification
Our approach	Object	CollD + StateD	Seq[1] + Iter[2] + Concur[3]	Automatic	State	Yes	Yes
Dano et al. (1997)	System	Table + Petri net	Seq + Iter + Concur	Semi-automatic	State	No	No
Deharnais et al. (1998)	System	Relation	Seq	Semi-automatic	Composition	No	No
Elkoutbi et al. (2000)	System	Sequence D + Petri net	Seq + Iter + Concur	Semi-automatic	State	Yes	No
Heimdahl and Leveson (1996)	System	StateD	N/A[4]	N/A	N/A	N/A	Yes
Koskimies et al. (1998)	Object	Sequence D + StateD	Seq	Automatic	Inference	No	No
Lee et al. (1998)	System	Table + Petri net	Seq + Concur	Semi-automatic	Synchronization	No	Yes
Mäkinen and Systä (2000)	Object	Sequence D + StateD	Seq	Semi-automatic	Inference	Yes	No
Heitmeyer et al. (1998)	System	Table + FSM	Seq	Semi-automatic	State	No	Yes
Somé et al. (1996)	System	Table + Timed automata	seq + Time based	Automatic	State	No	Yes
Harel et al. (1990)	System	StateD	N/A	N/A	N/A	N/A	Yes
Whittle and Schumann (2000)	Object	Sequence D + StateD	Seq	Automatic	State	No	No

that their work assumes that if a guard condition consists of a set of predicates where some of these conditions are dependent, these dependencies must be specified explicitly.

Harel et al. (1990) have developed STATEMATE, a commercial tool, which provides automatic code generation as well as system simulation for verification purposes, such as reachability, non-determinism and deadlock. We consider STATEMATE as a complementary tool in respect to our approach. In fact, StateDs synthesized by a tool such as ours may be passed to STATEMATE for simulation and further analysis.

8. Discussion of Approach

Below, we discuss our approach in respect to some interesting aspects: restriction over conditions, interleaving problem, complexity of algorithms, evaluation and relevance of the approach in the development process.

Restriction over conditions

We have seen that the syntax of a condition (i.e. pre- and post-condition of an operation) is restricted to a subset of OCL (see Section 4). This restriction has not a big impact on the generality of our approach. Indeed, a large number of systems can be supported by our work and our assumption is less restrictive than that of other works (for instance, the work of Heimdahl and Leveson (1996) described above).

Also, this restriction allows us to perform many operations on conditions such as *equality* between conditions (see Definition 2) and *refines* operation (see Definition 4) in a polynomial time. In contrast, Heimdahl and Leveson have indicated that such operations are exponential (1996). In fact, in our approach these operations are performed using the function *Eval* (see Definition 1), which has a polynomial complexity. For the work of Heimdahl and Leveson, these operations are performed using the prepositional calculus.

Problem of interleaving between scenarios

In this work, we have solved the problem of interleaving between scenarios by defining and introducing composition variables, without changing the syntax and semantic of StateDs. Note that sometimes interleaving is needed to capture scenarios that are not already considered. By executing or not executing sub-step 4 of the IPS algorithm, one has the choice of allowing or preventing the interleaving between scenarios.

Problem of interleaving between scenarios

Our approach is supported by three algorithms the GPS algorithm, the APS algorithm and the IPS algorithm. Let N_M be the number of messages in a CollD. The worse-case complexity of the GPS algorithm is $O(N_M^2)$ (see (Schönberger et al., 2001)). For the discussion of the worst-case complexity C_A of the APS algorithm, let N_O be the number of operations in the description of a class, N_T the number of transitions in a StateD, N_S the number of states in a StateD and N_{OR} the maximum number of ifExpression that may contain a post-condition of an operation. Since the algorithm consists of three sub-steps, C_A is the sum of C_{A1}, C_{A2}, and C_{A3}, providing that C_{Ai} represents the complexity of Sub-step i. Sub-step 1 checks the consistency for all operations in a class. C_{A1} is therefore $O(N_O)$. Sub-step 2 labels each transition of a StateD. It consists to traverse, with backtracking, all transitions of the StateD in order to find the right sequence of the post-condition ifExprssion of the operations of the corresponding event transitions. C_{A2} is therefore $O(N_{OR}^{NT})$. Sub-step 3 checks the consistency of the resultant labeled StateD, which consists to check the coherence of the guard condition of all transitions, the existence of a non-determinism behavior, and the coherence between sub-states of each composite states of the StateD. C_{A3} is therefore $O(N_T+N_T^2+N_S^2)$. Consequently C_A is $O(N_{OR}^{NT})$.

Note that the exponential complexity of the APS algorithm has no effect because the algorithm processes partial StateDs where the transitions number is not high since each of these StateDs specifies only one scenario.

For the discussion of the worst-case complexity C_I of the IPS algorithm, which integrates two StateDs, let N_{T1} (respectively, N_{T2}) be the number of transitions in the first StateD (respectively, the second StateD), N_{S1} (respectively, N_{S2}) be the number of states in the first StateD (respectively, the second StateD). Since the algorithm consists of five sub-steps, CI is the sum of C_{I1}, C_{I2}, C_{I3}, C_{I4} and C_{I5} providing that C_{Ii} represents the complexity of Sub-step i. Sub-step 1 checks the existence of conflicts between states of the two StateDs. C_{I1} is therefore $O(N_{S1} * N_{S2})$. Sub-step 2 merges states of the two StateDs level by level. C_{I2} is therefore $O(\max(N_{S1}^2, N_{S2}^2))$. Sub-step 3 merges transitions of the two StateDs. C_{I3} is therefore $O(N_{T1} * N_{T2})$. Sub-step 4 introduces three composition variables in the two StateDs. C_{I4} is therefore $O(\max(N_{T1}, N_{T2}))$. Sub-step 5 checks the consistency of the integrated StateD. C_{I5} is equal to C_{A3} and therefore is $O(\max(N_{T1}, N_{T2}) + \max N_{T1}, N_{T2})^2 + \max(N_{S1}, N_{S2})^2)$. Consequently CA is $O(\max(N_{T1}, N_{T2})^2+\max(N_{S1}, N_{S2})^2)$.

Validation of approach

The three algorithms that constitute the basis of our approach have been implemented with a system of 22 classes and about 6000 lines of code in the Java language (comments not included). For validation purposes, we have adopted a textual format for scenario acquisition and the presentation of the resulting specifications.

Our approach has been successfully applied to a number of examples such as the ATM system presented in this paper, a library system, a gas station simulator (Coleman et al., 1994) and a filing system (Derr, 1996).

Relevance of approach in the development process

Current OO CASE tools support various graphical notations for modeling a system from different views, but lack the possibility of automatic transformation between models and analysis of models. The incorporation of our work into such CASE tools will ease the activity of scenario-based requirements engineering.

Furthermore, our incremental-based approach enables us to have an iterative process for scenario-based requirements engineering. In case of changes in some scenarios that have already been integrated, new partially labeled StateDs are generated after reapplication of the activities two and three of our approach. The integration algorithm (activity four) is then reapplied over the partially labeled StateDs corresponding to the unchanged scenarios and the new ones.

9. Conclusion and Future Work

The work presented in this paper proposes an automatic approach for generating and analyzing a system specification from scenarios. Scenarios are acquired as UML CollDs. These CollDs are transformed into partial object specifications through an existing algorithm. Then, the resultant specifications are analyzed and merged using the two algorithms detailed above. These algorithms also support requirements verification in respect to consistency and completeness aspects.

The most interesting features of our approach can be summarized in two points. The first point concerns the form of integration, which in our case is quite general, addressing not only sequential integration but also concurrency and hierarchy. The second point consists in solving the problem of interleaving between scenarios in the resulting specification.

As future work, we plan to investigate how we can adapt our approach in order to support real-time systems. On the practical side, we aim to integrate the prototype implementation of our algorithms into a commercial CASE tool.

10. References

Anderson, J.S. and Durney, B., 1993, "Using Scenarios in Deficiency-driven Requirements Engineering," Proceedings of the Conference on Requirements Engineering 1993, IEEE Computer Society Press, pp. 134-141.

Angluin, D., 1987, "Learning Regular Sets from Queries and Counterexamples," Information Computing, Vol. 75, pp. 87-106

Biermann, A.W. and Krishnaswamy, R., 1976, "Constructing Programs from Example Computations," IEEE Transactions on Software Engineering, Vol. 2 Num 3, pp. 141-151.

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

Caroll, J.M. and Rosson, M.B., 1992, "Getting around the Task-Artifact Cycle: How to Make Claims and Design by Scenario," ACM Transactions on Information Systems, Vol. 10 Num. 2, 181-212.

Coleman, D., Arnold, P., Bodoff, S., Dollin, Ch., Gilchrist, H., Hayes, F. and Jeremaes, P., 1994, "Object-Oriented Development: The Fusion Method," Prentice-Hall, Inc.

Dano, B., Briand, H. and Barbier, F., 1997, "An Approach Based on the Concept of Use Cases to Produce Dynamic Object-Oriented Specifications," Proceedings of the Third IEEE International Symposium on Requirements Engineering 1997, Annapolis, pp. 56-64.

Desharnais, J., Frappier, M., Khédri, R. and Mili, A., 1998, "Integration of Sequential Scenarios," IEEE Transactions on Software Engineering, Vol. 24 Num. 9, pp. 695-708.

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

Elkoutbi, M. and Keller, R.K., 2000, " User Interface Prototyping based on UML Scenarios and High-level Petri Nets," Application and Theory of Petri Nets 2000 (Proc. of 21st Intl. Conf. on ATPN), Aarhus, Denmark, June 2000. Springer. LNCS 1825, pp. 166-186.

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

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Polit, M., Sherman, R. and Shtull-Tauring, A., 1990, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," IEEE Transactions on Software Engineering, Vol. 16 Num. 4, pp. 403-414.

Harel, D., 1987, "A Visual Formalism for Complex Systems," Science of Computer Programming, Vol. 8, pp. 231-274.

Heimdahl, M.P.E. and Leveson, N.G., 1996, "Completeness and Consistency in Hierarchical State-Based Requirements," IEEE Transactions on Software Engineering, Vol. 22 Num. 6, pp. 363-377.

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

Hsia, P., Yuang, T.A., 1998, "Screen-based Scenario Generator: A Tool for Scenario-based Prototyping," Proceedings of the twenty-first Annual Hawaii International Conference 1998, IEEE Computer Society, pp. 455-461.

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

Koskimies, K., Systä, T., Tuomi, J. and Mannisto, T., 1998, "Automatic Support for Modeling OO Software," IEEE Software, Vol. 15 Num. 1, pp. 42-50.

Lee, W.J., Cha, S.D., Kwon, Y.R., 1998, "Integrating and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering," IEEE Transactions on Software Engineering, Vol. 24 Num. 12, pp. 1115-1130.

Mäkinen, E. and Systä, T., 2000, An Interactive Approach for Synthesizing UML Statechart Diagrams from Sequence Diagrams," Proceedings of OOPSLA 2000 Workshop: Scenario-based round-trip engineering, pp. 7-12.

Monk, A.F., Wright, P.C., 1990, "Observations and Inventions: New Appraoches to the Study of Human Computer Interaction," Interacting With Computers, Vol. 3 Num. 2, pp. 204-216.

Nardi, B.A., 1992, "The Use Of Scenarios In Design," SIGCHI Bulletin, Vol. 24 Num. 4.

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

Rubin, K.S. and Goldberg, A., 1992, "Object Behavior Analysis," Communications of the ACM, Vol. 35 Num. 9, pp. 48-62.

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

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

Somé, S., Dssouli, R., Vaucher, J., 1996, "Towards an Automation of Requirements Engineering using Scenarios," Journal of Computing and Information, Vol. 2 Num. 1, pp. 1110-1132.

Schönberger, S., Keller, R.K. and Khriss, I., 2001, "Algorithmic Support for Transformations in Object-Oriented Software Development," Concurrency and Computation: Practice and Experience, 13(5):351-383, April 2001. Object Systems Section. John Wiley and Sons.

Whittle, J. and Schumann, J., 2000, "Generating Starechart Designs from Scenarios," Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, ACM Press, pp. 314-323.

Wordsworth, J.B. 1992, "Software Development with Z: A Practical Approach to Formal Methods in Software Engineering," Addison-Wesley, Inc.