Bridging the Gap between Analysis and Design Using Dependency Diagrams

Simona Vasilache, Jiro Tanaka
University of Tsukuba, Japan
{simona,jiro}@iplab.cs.tsukuba.ac.jp

Abstract

Requirements specifications often make use of a number of scenarios that are interrelated and that depend on each other in many ways. However, they are often treated separately, one by one. We propose a new type of diagrams, named dependency diagrams, which are able to illustrate the various kinds of relationships existing between scenarios. We make use of a scenario matrix for each scenario and we describe the transformation process of scenarios into state machines, based on the information in the scenario matrices, and that in the dependency diagrams. The result is a number of state machines that can be used for detailed design models and code can further be generated from them. Using our approach, we can bridge the gap between analysis and design and we can bring the developer one step closer to the implementation.

1. Introduction

The requirements of a system constitute the constraints, desires and hopes we have concerning the system under development [1].

As a first phase in the software development process, the requirements analysis is a crucial one, because it represents the starting point from where the whole application will be developed and ultimately put into practice. The main task of the requirements analysis is to generate specifications that describe the behaviour of a system unambiguously, consistently and completely [2]. UML [3], as well as other notations and object-oriented methodologies, make use of scenarios as a handy means of capturing requirements specifications. They are also helpful as a means of communication between users and software developers. Their usefulness relies not only on the ability to capture requirements, but also on their applicability when used in conjunction with other models. We specifically refer to what is called "behaviour models", that is models that describe the behaviour of a system. When it comes to these behavioural aspects, state machines (particularly statecharts, originally introduced by D. Harel [4]), represent a compact and elegant way of describing them. While scenarios represent a single trace of behaviour of a complete set of objects, state machines represent the complete behaviour of a single object. The two concepts together provide an orthogonal view of a system.

During the software development process, there is often a gap between analysis and design; it is often difficult to understand and have an overview of all the behavioural aspects of all the parts that will constitute the objects in the system. State machines can be used not only for behavioural requirements specifications, but also for detailed design models close to implementation [19]. Together with class diagrams, the information contained in the state machines can be used during design, allowing the representation of the behavioural aspects in a compact and elegant manner. Moreover, code can be generated from these state machines. Our contribution smoothens the transition from analysis to design and helps with the implementation process.

Scenarios are generally not independent of each other; various relationships and dependencies connect them. We make a classification of these relationships and in order to represent them we propose a new type of diagrams. We have called these diagrams dependency diagrams. We propose an algorithm of transformation of scenarios into state machines based on the information in the scenario matrix of each scenario (a matrix of tuples including all the messages exchanged between all objects part of that scenario) and on the information in the dependency diagrams. In this paper we detail this transformation, along with an example illustrating the whole process.

The remainder of the paper is organized as follows: section 2 offers an overview of scenarios and state machines, while section 3 describes the dependency diagrams and their properties. In section 4 we describe the scenario matrix, we detail the algorithm of transformation of scenarios into state machines, and furthermore we offer an example that illustrates our approach. Section 5 deals with related work and is followed by conclusions in section 6.

2. Scenarios and state machines

2.1. Scenarios as sequence diagrams

A scenario is a sequence of events that occurs during one particular execution of a system [2], it is one particular "story" of using a system. In UML, scenarios are represented as sequence diagrams. Sequence diagrams illustrate how objects interact with each other. They focus on showing the sequence of messages sent between objects, that is the interaction between objects from a temporal point of view.

Sequence diagrams have two axes: the vertical axis shows time and the horizontal axis shows a set of objects. An object is represented by a rectangle and a vertical bar called the object's lifeline. Objects communicate by exchanging messages, represented by horizontal arrows drawn from the message sender to the message recipient. The message sending order is indicated by the position of the message on the vertical axis.

Scenarios represent a powerful means of expressing the requirements specification of a system.

2.2. State machines

The representation used in UML for state machine diagrams, called statechart diagrams, is inspired from Harel's statecharts [4]. State machine diagrams describe which states an object can have during its life cycle and the behaviour in those states, along with what events cause the state to change. All objects have a state that is a result of previous activities performed by the object. An object changes state when something happens, which is called an event.

State machine diagrams have proved their usefulness in the dynamic description of the behaviour of a system. Together with class diagrams, they can be used during design models and, furthermore, they can be used for generating code directly from them.

3. Dependency diagrams

3.1. Dependency diagram notation

In order to describe completely the requirements specification, a number of scenarios are needed; this is because one scenario represents only one particular "story" of the use of a system. These scenarios are not independent of each other, several relationships and dependencies interconnect them. When transforming the scenarios into state machines, different relationships between scenarios result in different state machine structures. This fact is of considerable importance and this is what determines us to believe that the relationships between scenarios should be taken into account and should be given a proper representation.

In order to be able to represent and make use of the relationships existing between these various scenarios, we have introduced dependency diagrams. The notation used in these diagrams is based on the notation used in Message Sequence Charts [5]. One scenario is represented as a rounded rectangle, with connectors for start point and end point (corresponding to entry and exit points). The positioning in space of different scenarios shows the order of execution. A connection node (represented as a circle) helps connecting different branches. Fig. 1 shows the basic notation used in dependency diagrams.

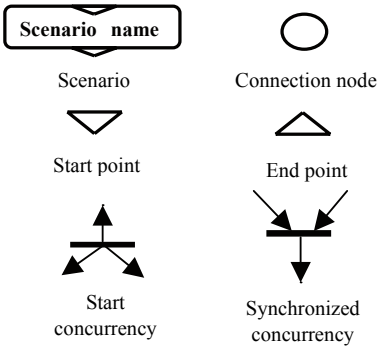

Fig.1. Basic notation for dependency diagrams

3.2. Representation of dependencies

Depending on the application, the number of scenarios varies; however small the number of all possible scenarios, relationships and dependencies exist between them. In numerous cases the order and the timing of the execution of scenarios is not random, but well established. Two or more scenarios can be related in many ways: the execution of a scenario can depend on the execution time of another one (e.g. it can only be executed after/before another scenario), the necessary conditions for a scenario to be executed are fulfilled in a different one, one scenario represents a part of another, a set of scenarios are very similar with each other, representing a variant of a basic scenario, and so on.

To illustrate our point, let us consider a simplified example of an ATM (Automated Teller Machine). A consortium of banks shares the ATMs. Each ATM accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash and prints receipts. We will use (simplified) typical scenarios for user interaction with an ATM machine, like inserting or removing a card, entering a password, deciding upon a certain type of transaction

(withdrawal, deposit or transfer) etc. For example, if we consider the scenario depicting the action of withdrawing cash, this can be executed only in the situation of the user possessing a valid card. The scenario of the user applying for a card with a bank must precede the scenarios involving transactions with the bank in the user's name.

We have made a classification of the dependencies between scenarios as follows:

- time dependencies;
- cause-effect dependencies;
- generalization dependencies.

Time dependencies reflect the fact that scenarios are related in terms of time, that is one scenario has to be performed before/after/simultaneously with another scenario. For instance, as described above, a user must first prepare a card and only then (s)he can perform transactions through the ATM. Therefore, the scenario of creating a card precedes the scenario of withdrawing cash and the two scenarios together reflect a time dependency.

A cause-effect dependency illustrates the fact that the execution of a scenario can take place only the moment certain conditions (established in another scenario) become valid. For instance, an ATM can satisfy the user's request for withdrawing cash only if it has been previously provided with a large enough number of bills and coins. The scenario of withdrawing cash depends on the scenario of the ATM machine being "loaded" with a sufficient amount of cash (considered to cover the maximum amount that could be withdrawn during a whole day). The condition of "being able to provide enough cash" is established in a different scenario from the one where the transaction itself takes place.

Although from the point of view of the representation in the dependency diagrams, the time dependency and cause-effect dependency are equivalent (they can be represented in the same way), our belief is that it is important to differentiate between them. We want to emphasize when the dependency arises from a specific time sequence (like having to insert the card and password first, and only after that being able to perform a transaction) and when a dependency arises from certain conditions that are not explicitly time-related (at least, not necessarily). For instance, a user could withdraw cash only if the ATM has been provided with bills and coins. We believe it is not so important to emphasize the time sequence (supplying the bills and coins first and then being able to satisfy the user's request for cash), as it is important to emphasize that having the bills is a necessary condition, which if it is not met, the operation cannot take place.

Finally, a generalization dependency (or abstraction, as it is described in [6]) appears when one scenario is a part of another one or a variant of it. Two scenarios can be very similar and they can be generalized under one scenario.

The execution order of a number of scenarios (defining the time dependencies) falls into one of the following categories: succession (one scenario follows another one), disjunction (at a certain moment in time only one of the scenarios involved is executed), conjunction (the scenarios are executed simultaneously) and recurrence (a scenario is executed a certain number of times). (A similar classification has been made in [6]; the authors used the terms "sequence", "alternative", "concurrency" and "iteration".)

A simple example of a dependency diagram is shown in Fig. 2. It is based on the same example of ATM introduced earlier. Here we consider *Scenario start* as the initial scenario. The user approaches the ATM, inserts the card, the card is validated and the main options screen is displayed. From this point, the user can select any of the 3 operations of withdrawing cash, depositing cash or transferring cash, that is either *Scenario withdraw* or *Scenario deposit* or *Scenario transfer* respectively. We also suppose that when the user changes his(her) password (*Scenario chg. pass.*), the scenario *Scenario videotape* takes place simultaneously, that is, the user is being videotaped during the operation of changing the password. (Although this is a simplified version of an ATM system, it facilitates the illustration of the points we intend to make).

Fig. 2 illustrates time-dependencies between several scenarios, namely succession (*Scenario start* precedes the other ones), the disjunction of 3 scenarios, *Scenario withdraw*, *Scenario deposit* and *Scenario transfer* (any of them can be executed after *Scenario start*), as well as the conjunction of 2 scenarios, *Scenario chg. pass.* and *Scenario videotape*.

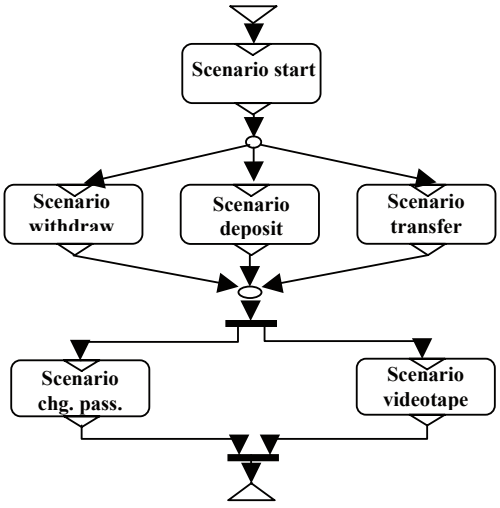

Fig.2. Dependency diagram for several scenarios of an ATM system

We consider that dependency diagrams offer several benefits in the process of requirements analysis and throughout the whole development process of a system.

By representing the relationships between various scenarios, we can easily tell what other scenarios would be affected if one scenario were changed. This contributes considerably to the enhancement of traceability. Also, we beneficiate of an improved readability; by seeing how the different scenarios are related to each other, we can have a better overview of the requirements of the system.

Furthermore, by carefully representing all the possible relationships, we can easily generate a multitude of test cases. We know that, as Dijkstra famously stated three and a half decades ago, "Program testing can be used to show the presence of bugs, but never to show their absence" [23]. We cannot find the errors only on the basis of testing, but, nevertheless, by providing the opportunity to derive numerous test cases, we can narrow down the number of possible inconsistencies and errors in the intended system.

4. Algorithm of transformation

4.1. Scenario matrix representation

In a scenario, more exactly in its representation as a sequence diagram, there are a number of messages exchanged between objects. Each such message is a tuple: (O_i, O_j, $M_{ijk,}$W).

O_i and O_j belong to the set of all objects involved in the system. M_{ijk} depicts the message exchanged between object i and object j. There can be more messages exchanged between the same objects, so k is used to denote these different messages. O_i represents the source of the message, while O_j represents the destination.

W symbolizes the type of message and its value can be: 0 if the message is simple, 1 if the message is synchronous or 2 if it is asynchronous. A simple message denotes a flat flow of control; the control is passed from one object to another, without any details about the communication (these details are either not known or not relevant). A synchronous message means that the operation that handles the message is completed before the caller resumes execution. An asynchronous message reflects that there is no explicit return to the caller; the sender continues to execute after sending the message without waiting for it to be handled (this is typically used in real-time systems where objects execute concurrently).

Therefore, we can represent a scenario as a matrix of tuples including all the messages exchanged between all objects part of that scenario.

For example, if we consider the ATM system, let us assume a scenario Sc_1 (represented in Fig. 3) where 4 objects are involved: *User, ATM, Consortium* and *Bank.*

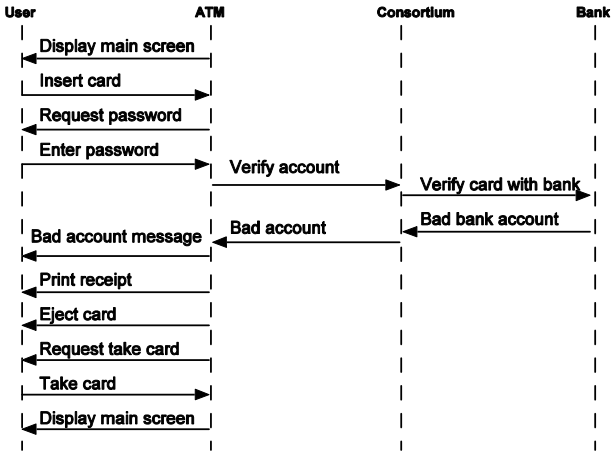

Fig.3. Scenario (sequence diagram) for an ATM

In this scenario, after the user enters the card and then the password, the ATM verifies the card with the consortium, which, in turn, verifies it with the bank. The bank sends a bad bank account event to the consortium, and the consortium sends a bad account event to the ATM. The ATM in turn sends a bad account message event to the user. In the end, a receipt is issued, the card is ejected and the user is requested to take the card back. The messages exchanged in this example scenario are: displaying the main screen (from the ATM to the user), inserting a card (from the user to the ATM), requesting password (from the ATM to the user), entering password (from the user to the ATM), and so on.

Let us represent the objects *User, ATM, Consortium* and *Bank* as O_1, O_2, O_3 and O_4 respectively. Our scenario Sc_1 will therefore be represented in the scenario matrix in Fig. 4. (All the messages involved in this scenario are synchronous messages.) There are 4 objects involved in this scenario; from this scenario only, we can synthesize 4 state machine diagrams, one for each object.

$$Sc_1 = \begin{pmatrix} O_2, O_1, M_{211}, 1 \\ O_1, O_2, M_{121}, 1 \\ O_2, O_1, M_{212}, 1 \\ O_1, O_2, M_{122}, 1 \\ O_2, O_3, M_{231}, 1 \\ O_3, O_4, M_{341}, 1 \\ O_4, O_3, M_{431}, 1 \\ O_3, O_2, M_{321}, 1 \\ O_2, O_1, M_{213}, 1 \\ O_2, O_1, M_{214}, 1 \\ O_2, O_1, M_{215}, 1 \\ O_2, O_1, M_{216}, 1 \\ O_1, O_2, M_{123}, 1 \\ O_2, O_1, M_{211}, 1 \end{pmatrix}$$

Fig.4. Scenario matrix for scenario in Fig.3

In a complete description of this system, if we assume that there are N scenarios, with a total number of P objects, we will have N matrixes including all the transitions between objects. The total number of state machine diagrams will be equal to the total number of objects in all scenarios. We will therefore have a number of P final state machine diagrams.

4.2. Algorithm main phases

Our ultimate purpose is the synthesis of state machines, one for each object existing in the collection of scenarios. The algorithm of synthesis of state machine involves the following major phases**:**

I. identify and represent (as sequence diagrams) all single scenarios;
II. identify and represent (as dependency diagrams) the relationships between all scenarios;
III. synthesize the state machines diagrams, based on the information acquired in the previous two phases.

As stated above, the number of state machine diagrams will be equal to the total number of objects involved in all the scenarios. For each object, the synthesis of state machine (phase III) involves two steps:

1. creating the initial state machines, that is one state machine diagram for each scenario;
2. synthesizing the final state machine diagram by combining all the initial state machine diagrams.

For the creation of the initial state machines, the sequence of events in one sequence diagram corresponds to paths through the initial state machine diagrams of the corresponding objects. We have to consider the vertical line that corresponds to the desired object. For an object in a sequence diagram, incoming messages represent events received by the object and they become *transitions* in the state machine diagram. Outgoing messages are actions and they become *actions* of the transitions leading to the states. The intervals between events become *states*. Before receiving any event, the object is in the *default state*.

Sequentially, when we create initial state machines, we have to do the following:

1. create empty state machine diagrams, one for each scenario where the object appears;
2. for each state diagram, create all events (corresponding to transitions to the object);
3. for all transitions from the object, create actions that will lead to states and create the respective states;
4. set the right time sequence for the transitions.

In general, for an object O_x, we first have to identify all scenarios where O_x is a participant. For each tuple (O_i, O_j, M_{ijk}, W) where O_x appears we will have:

- In case of (O_x, O_j, M_{xjk}, W), that is O_x is the originator of the message, a state S_{xjk} (with the same name as M_{xjk}) is born;
- In case of (O_i, O_x, M_{ixk}, W), that is O_x is the receiver of the message, a transition T_{ixk} (with the same name as M_{ixk}) is born.

In this manner we will obtain two lists for the state machine of object O_x: one containing all the states and one containing all the transitions of the state machine.

Returning to our example, let us focus on the scenario matrix that appears in Fig. 4. If we want to create the state machine for object ATM (that is object O_2), we will obtain the following list of states: S_{211} (*Display main screen*), S_{212} (*Request password*), S_{231} (*Verify account*), S_{213} (*Bad account msg.*), S_{214} (*Request take card*), S_{215} (*Eject card*), S_{216} (*Print receipt*), and finally again S_{211} (*Display main screen*). The following is the list of transitions for object O_2 (that is the ATM object): T_{121} *Insert card*), T_{122} (*Enter password*), T_{321} (*Bad account*), T_{123} (*Take card*).

The last step in creating the initial state machine is setting the right time sequence. The scenario matrix preserves the time sequence in the sequence diagrams, where time flows from top to bottom. Thus, we will follow the time sequence in scenario matrix Sc_1.

The resulting state machine of object ATM, synthesized from scenario Sc_1, appears in Fig. 5

The default state has to be specified; in our case, it is *Display main screen*.

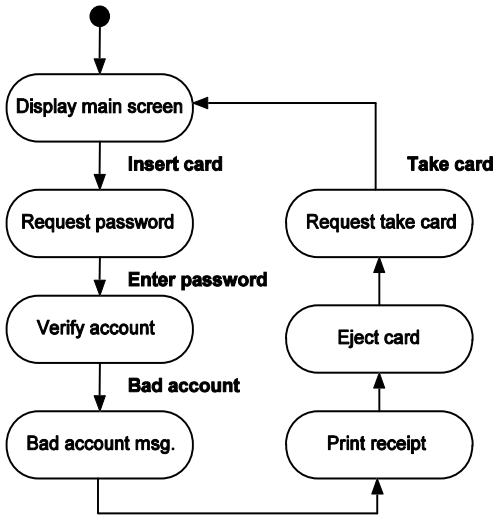

Fig.5. State machine diagram for object ATM (O_2)

We can observe here that there is no event received by the ATM object in between the states *Print receipt*, *Eject card* and *Request take card*. This means that we could merge the 3 states into a single one, since there is nothing

that could alter this succession of states.

4.3. Synthesis of final state machine

In order to obtain the final state machines (one for each object involved in the totality of scenarios), we will make use of the information in the dependency diagrams. As described previously, the dependency diagrams show the possible relationships existing between scenarios. Based on the classification of relationships between scenarios, there are several rules that need to be followed:

- In a succession of two scenarios, the two corresponding state machine diagrams will merge in the final state machine diagram.
- If a transition is common to two scenarios, it will be taken only once in the final state machine.
- For two scenarios related with a disjunction relationship, their corresponding state machines should be combined with OR.
- If two scenarios are executed at the same time, their corresponding state machines must be combined with AND.

Finally, the state machine diagrams need to be refined, with respect to aggregation of states and generalization of states.

To illustrate the synthesis of a state machine, let us consider 3 scenarios of using the ATM: one for withdrawing cash, one for depositing cash and one for transferring money. The scenario for withdrawing cash, *Scenario_withdraw_initial*, and its corresponding scenario matrix are represented in Fig. 6.

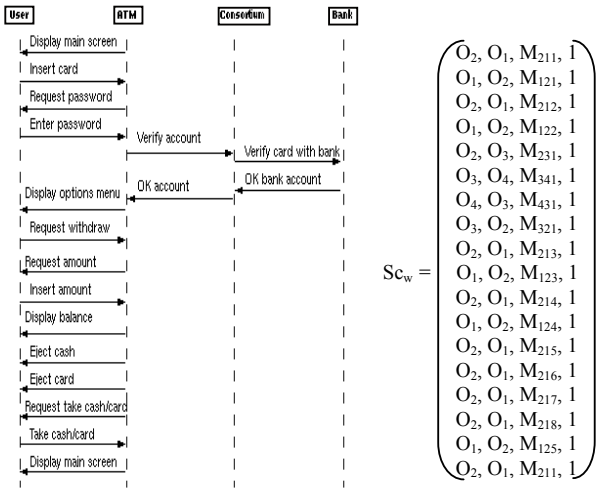

Fig.6. Sequence diagram and scenario matrix for *Scenario_withdraw_initial*

Scenarios *Scenario_deposit_initial* and *Scenario_transfer_initial*, for depositing cash and transferring money, have a similar representation. In the case of each of these 3 scenarios, in the beginning, the user inserts the card and password and they are validated by the bank. This part is common to all 3 scenarios and therefore we will separate it and call it *Scenario start*. We will call the remaining parts of each of the 3 scenarios *Scenario withdraw*, *Scenario deposit* and *Scenario transfer* respectively.

Scenario start precedes all the other scenarios, thus we deal with a succession relationship. The other 3 scenarios are related by disjunction (only one of them can take place at a certain moment in time). The relationships between all these scenarios appear in the dependency diagram in Fig. 7.

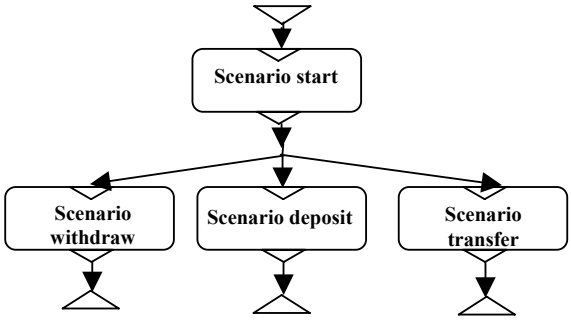

Fig.7. Dependency diagram for scenarios *Scenario start, Scenario withdraw, Scenario deposit* and *Scenario transfer*

Based on this dependency diagram and on the corresponding scenario matrixes (one of them appearing in Fig. 6 and the other two similar to it, although not represented here), and following the proposed algorithm, we will obtain a state machine diagram as the one in Fig. 8. This state machine is already refined and we will explain how we achieved this in the following. The event *OK account* takes the system into one of the states where cash can be withdrawn, cash can be deposited or money can be transferred. Because they have the same entry and the same exit actions, they can be combined into a superstate. We represent a single superstate as a rectangle, and inside it we figure the corresponding states resulting from each scenario. We thus obtain multilevel state machines. As introduced by Harel in [4], we use the concept of *cluster*. We can cluster, therefore, the 3 sets of states (corresponding to each possible action: withdraw, deposit or transfer) into a new superstate, named "transaction". The semantics of the superstate "transaction" is the OR of the 3 states mentioned above.

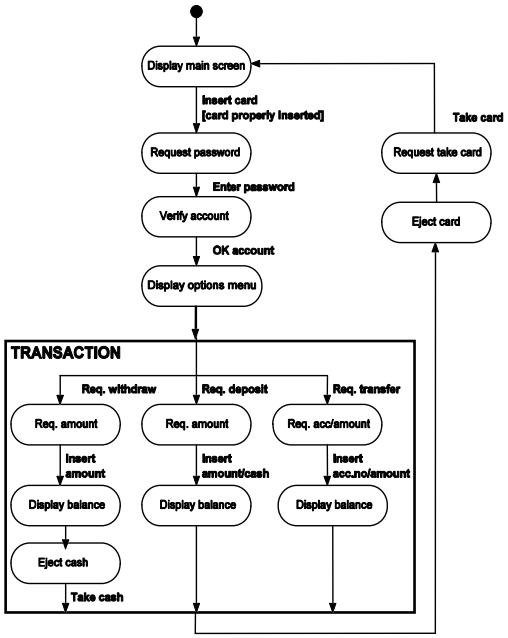

Fig.8. State machine for object ATM

The rectangle (which we figured on purpose with a thicker black line) is the one that clusters the OR type substates. The 3 possible actions that lead to transitions are *Request withdraw, Request deposit* and *Request transfer*. The user can decide upon only one of them at a certain moment in time; according to the user's decision (the action), the corresponding transition takes place (*Request deposit, Request withdraw* or *Request transfer*).

In our example we only constructed the state machine for one object, *ATM*. In the same manner, we can obtain the state machine diagrams for the other 3 objects that can be found in the scenarios (*User*, *Consortium* and *Bank*).

4.4. Various issues

When creating the scenario matrices, it is important to know what kinds of messages are involved, that is simple, synchronous or asynchronous messages. All the messages in our example are synchronous, so the operation that handles the message is completed before the caller resumes execution. This happens in the case of numerous scenario messages, but it is not always the case. Dealing with asynchronous messages raises the problem of setting the right time sequence (the last step in creating the initial state machines), making the issue more complex.

Another important issue appears after drawing the dependency diagrams, when we might have several messages, external to the scenarios, but which appear as a result of the relationships between them. These messages appear explicitly in the dependency diagrams. After we identify the tuple for each of these messages, (Oi, Oj, Mijk, W), we can apply the same rules as the ones above and integrate the corresponding states and/or transitions into the final state machine. However, we will have to carefully consider the exact timing of occurrence of the corresponding messages and, since this is an inter-scenario message, we will have to identify its place inside the originating scenarios, as well as inside the receiving scenario.

The process of synthesis does not end with applying the algorithm and the rules defined. Before we can say that we obtained a correct and complete final state machine diagram for each object, we need to address the issue of consistency between the state machines and the scenarios. We have to make sure that the behaviour of the final state machine diagrams reflects the information contained in the scenarios, so that we respect the requirements specifications. This is a task that involves the detection of implied scenarios, the unwanted behaviour appearing in the state machines and the possible conflicts that might arise. Only after solving these problems the process of synthesis can reach its end.

The state machine diagrams obtained can be used for the implementation. They offer a dynamic view of the system, whereas a static view can be found in the class diagram of the system. Attached to this class diagram, the state machine diagrams can express the design model of the system and can facilitate the code generation. There are several tools and research papers (e.g. [20], [21], [22]) that deal with generating code from state machine diagrams.

5. Related work

The problem of transforming scenario type models into behaviour models is dealt with in several papers. In [8], an algorithm for generating UML statecharts from sequence diagrams is given, but the relationships between the sequence diagrams (as representations of scenarios) are limited to the introduction of hierarchy. SCED [9] is a tool for automatic generation of statecharts from single scenarios. Schonberger et al. [10] describe an algorithm for model transformation, more precisely an algorithm for transforming collaboration diagrams into state diagrams. Collaboration diagrams describe the interaction among objects, with the focus on space. This means that the links among objects in space are of particular interest and explicitly shown in the diagram. Sequence diagrams (as representation of scenarios) on the other hand, although they also describe the way objects interact and communicate with each other, focus on time. Although the two kinds of diagrams are similar (and called together interaction diagrams), we favour the main use of

sequence diagrams during the analysis phase, as they allow an easier representation of the requirements (when we think of the usage of a system, i.e. of scenarios, we mainly focus on the time flow in the development of events).

In [6], Ryser and Glinz introduced a new kind of chart, dependency chart, and a new notation to model the dependencies between scenarios. However, the charts only show the dependencies between various scenarios, without giving directions about the way they could be used for translation into state machine diagrams.

We can actually observe that there is much work going on introducing methods that describe how to specify models; however, these methods do not sufficiently guide the developer in the task of transforming one model type into another, leaving, therefore, a gap between various phases of the application development, especially between analysis and design.

6. Conclusions

In this paper we have offered an instrument addressed to helping analysts and designers with the dynamic aspects of developing an application, based on scenarios and state machines. We have introduced dependency diagrams illustrating the various dependencies and relationships that exist between scenarios. They present the advantage of an enhanced traceability, a better overview and improved testing. We have also introduced a scenario matrix as a representation of all the messages exchanged between all objects in a scenario. Based on the information in the scenario matrices and the dependency diagrams, we have proposed a method of synthesizing state machines form multiple interrelated scenarios. Our approach bridges the gap between analysis and design and brings the developer one step closer to the actual implementation of the desired system.

References

[1] D. Harel, "From Play-In Scenarios to Code: An Achievable Dream", Fundamental Approaches to Software Engineering (FASE2000), LNCS 1783, Springer-Verlag, 2000, pp. 22-34.

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

[3] UML Resource Page, http://www.uml.org/.

[4] D. Harel, "Statecharts: A visual formalism for complex systems", Science of Computer Programming, 8(3), 1987, pp. 231-274.

[5] L. Helouet, C. Jard, "La manipulation formelle de scenarios", Modelisation des systemes reactifs, Vol. 0, 2001.

[6] J. Ryser and M. Glinz, "Using dependency charts to improve scenario-based testing", Proceedings of the 17th International Conference on Testing Computer Software (TCS2000), Washington D.C., 2000.

[7] J. Ali and J. Tanaka, "Constructing statecharts from event trace diagrams", Technical report of IEICE, KBSE98-33, 1998, pp. 41-47.

[8] J. Whittle and J. Schumann, "Generating statechart designs from scenarios", Proceedings of International Conference on Software Engineering (ICSE2000), Limerick, Ireland, 2000, pp. 314-323.

[9] K. Koskimies, T. Mannisto, T. Systa, J. Tuomi, "Automatic support for dynamic modeling of object-oriented software", IEEE Software, 15(1), 1998, pp. 87-94.

[10] S. Schonberger, R. K. Keller, I. Khriss, "Algorithmic support for model transformation in object-oriented software development", Concurrency and Computation: Practice and Experience, 13(5), 2001, pp. 351-383.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

[12] J. C. S. P. Leite, G. D. S. Hadad, J. H. Doorn, G. N. Kaplan, "A scenario construction process", Requirements Engineering, 5, 2000, pp. 38-61.

[13] H. Muccini, "An approach for detecting implied scenarios", Scenarios and state machines: models, algorithms, and tools, ICSE2002 Workshop, Orlando, Florida, USA, 2002.

[14] S. Vasilache and J. Tanaka, "Using dependency diagrams in dynamic modelling of object-oriented systems", Proceedings of the 7th IASTED Conference on Software Engineering and Applications SEA2003, Marina del Rey, USA, 2003, pp. 277-283.

[15] Craig Larman, Applying UML and patterns, Prentice Hall, 2002.

[16] F. Bordeleau, J. P. Corriveau, "On the need for "state machine implementation" design patterns", Scenarios and state machines: models, algorithms, and tools, ICSE2002 Workshop, Orlando, Florida, USA, 2002.

[17] I. Jacobson, *Object-oriented software engineering: A use case driven approach*, Addison Wesley, Reading, Massachusetts, 1992.

[18] M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, M. Veanes, "Validating Use-Cases with the AsmL Test Tool", Proceedings of the Third International Conference on Quality Software (QSIC2003), Dallas, USA, 2003.

[19] M. Mutz, M. Huhn, "Automated Statechart Analysis for User-defined Design Rules", Informatik-Bericht Nr. 2003-10, 2003.

[20] Fujaba Case Tool, http://www.fujaba.de/

[21] A. Knapp and S. Merz, "Model Checking and Code Generation for UML State Machines and Collaborations", Proceedings of the 5th Workshop on Tools for System Design and Verification,Reisenburg, Germany, 2002, pp. 59-64.

[22] I.A. Niaz and J. Tanaka, "Mapping UML Statecharts to Java Code", Proceedings of IASTED International Conference on Software Engineering (SE 2004), Innsbruck, Austria, Feb. 2004, pp.111-116.

[23] E. W. Dijkstra, "Notes on structured programming", T.H. – Report 70-WSK-03, 1970.