Abstract

In this work, we make an effort to offer a means of support during the design phase of developing a software system. We particularly focus on the dynamic aspects, more specifically on two of the several models that are often used during design: sequence diagrams and state machine diagrams.

Sequence diagrams often represent scenarios. During the requirements specifications phase, scenarios are widely used to capture these requirements. They are often interrelated in many ways; however, they are usually treated in isolation. We take into consideration their possible relationships, we classify them, and we propose a new kind of diagrams, named dependency diagrams, which are able to represent these relationships.

State machine diagrams, on the other hand, represent a compact and elegant way of describing behaviour. They can be used for detailed design models and they can represent a solid base for the implementation. Together with sequence diagrams, they offer many advantages in the software development process.

In order to make use of the benefits of both sequence diagram and state machine diagrams, we propose a process of transformation of sequence diagrams, as representations of scenarios, into state machine diagrams, based on the relationships between the given scenarios. Through the introduction of dependency diagrams, we can ensure that the behavior of the system will be completely and precisely reflected in the state machine diagrams, according to the given specifications.

Consequently, our purpose is emphasizing the relationships existing between various scenarios obtained during requirements analysis and using these relationships for the creation of state machine diagrams that are able to provide the complete behavior of the objects involved. Our contribution supports the analysis, as well as the design phase, and facilitates the implementation process.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to express my deepest gratitude to Prof. Jiro Tanaka, who has guided me, supported me and helped me constantly in the past years. His valuable insights, comments and suggestions have made possible the completion of my thesis.

I would like to express my thanks to all the members of the doctoral commission, Prof. Hiroaki Nishikawa, Prof. Moritoshi Yasunaga, Assoc. Prof. Jie Li and Assoc. Prof. Kazuo Misue, who gave me valuable comments and constructive suggestions to complete my work.

My warmest thanks go to all IPLAB members, who have been next to me all these years. I want to thank Tohru Ogawa, Motoki Miura and Yasutaka Sakayori, who have taught me so much in the early years, Yann Jacquinot, who has given me precious ideas for my work in its early stages, Ying Xiaoping, who has been a wonderful colleague and supportive friend, as well as Iftikhar Azim Niaz and Liu Xuejun, whose help and encouragement I have felt every day.

I want to thank my mother, my father and Ady, who have raised me to be who I am today, who have taught me everything, who have supported me, encouraged me and, above all, loved me no matter what, to whom I owe my eternal gratitude and to whom I dedicate this work. I want to thank my brother, Armand, and my whole family for always being by my side.

Last, but not least, I want to thank my son, Paul, who has made me so happy in the past year, and my husband, Adrian, without whom none of this would have been possible. His everyday support, help, patience and love have made me stronger and have helped me go through with this.

Chapter 1

Introduction

1.1 Motivation and objectives

The past years have seen an increased interest in developing various software development processes, models and methodologies. Among the phases that have received an increased attention, the design phase occupies an important place. Several models to be used during design phase have been proposed. Some of them are used in the requirements specifications phase as well, while others facilitate the transition to the implementation phase. The transformation of one model type into another usually gives a better overview of a system. However, this transformation is not always clearly formalized. We try to partially fill this lack of formalization in our work.

A software system can be said to have two distinct characteristics: a structural, static part and a behavioral, dynamic part. We are mostly concerned with the dynamic part, more specifically, with the dynamic models used during the design phase of developing a software system. We focus on two of these models: sequence diagrams, because they contain important information derived from the requirements specification, and state machine diagrams, because they offer behavioral information about the objects involved in the system.

Sequence diagrams can be used to represent scenarios. Although these scenarios are generally interrelated, they are often treated individually, as if they were isolated from each other. We believe it is important to make a

classification of the possible relationships that can exist between scenarios and we take these relationships into consideration when transforming sequence diagrams, as representation of scenarios, into state machine diagrams.

Our final objective is to offer a means of support during the design phase, with an emphasis on dynamic models, focusing on sequence diagrams and state machine diagrams, and offering a solution for the transformation of one into the other.

1.2 Structure of the thesis

Our thesis is organized as follows. Chapter 2 offers a background about the software development process in general and about UML. Chapter 3 is concerned with the relationships between scenarios, their importance, classification and representation as dependency diagrams, while chapter 4 describes the transformation process of scenarios into state machine diagrams. In chapter 5 we describe our system MUSEDESK and we illustrate it with an example of the transformation process, whereas in chapter 6 we evaluate it and make a comparison with another system. Finally, chapter 7 presents concluding remarks and future work.

Chapter 2

Background

2.1 The software development process

In its most simple definition, the software development process is the process used to develop a software product. This process entails not only the actual writing of the code, but also the definition of the requirements of the product, its design, and the confirmation that what has been developed has met objectives [44].

The software development process is usually guided by some systematic software development method. Several methods for the software development process have been proposed in the past years. Referred to by a number of terms, including *process models*, *development guidelines*, and *systems development life cycle models* [44], software development methods generally describe approaches to various activities that should take place during the process. Typical such activities include: identifying the requirements of the system, designing of the system, the actual implementation, testing and maintenance of the system. Some of the most popular systems development life cycle models and processes are: the waterfall model, the prototyping model, the spiral model, rapid application development, joint application development. Other process models include iterative processes, agile methods etc.

Fig. 2.1 shows a simple schema of the classical *waterfall* model. This model takes a sequential and linear approach, in which a project is carried out

in a series of steps, and each step must be completed and verified before advancing to the next one.

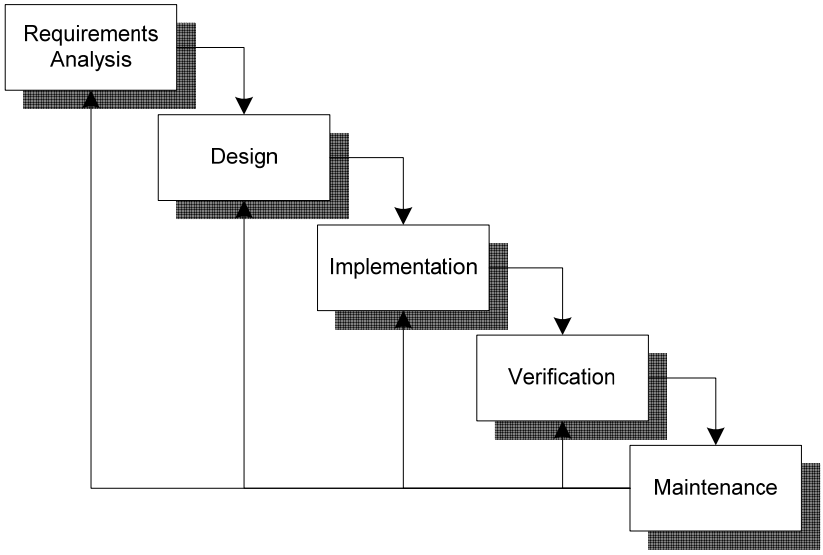


Fig. 2.1 The waterfall model

The first important step in developing an application is the one where requirements are elicited and then defined. This is where *what* is required of the system is specified. In order to be able to proceed to the implementation (writing the code itself), the design of the system needs to be completed, that is the defining of *how* to do what is required of the system. Analysis (a more general term including requirements analysis) and design have been summarized in the phrase: “do the right thing (analysis) and do the thing right (design)”.

Design is defined as the period of time in the software life cycle during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements [37]. Design offers a conceptual solution that fulfills the requirements. It represents a blueprint from which a reliable system can be built. In particular, during object-oriented design there is an emphasis on defining software objects and how they collaborate to fulfill the requirements [32]. Object-oriented design uses a variety of notations and diagrams. Some of the most commonly used such notations and diagrams appear in UML.

2.2 The Unified Modeling Language

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [45].

UML represents a notation which has become the de facto standard in object-oriented methods. In its current version, 2.0, UML makes use of 13 diagrams. Three classifications of UML diagrams exist [28]:

- **Behavior diagrams**. A type of diagrams that depict behavioral features of a system or business process. This includes activity, state machine, and use case diagrams as well as the four interaction diagrams.
- **Interaction diagrams**. A subset of behavior diagrams which emphasize object interactions. This includes communication, interaction overview, sequence, and timing diagrams.
- **Structure diagrams**. A type of diagrams that depict the elements of a specification which are irrespective of time. This includes class, composite structure, component, deployment, object, and package diagrams.

A software system can be said to have two distinct characteristics: a structural, "static" part and a behavioral, "dynamic" part [42].

- **Static:** The static characteristic of a system is essentially the structural aspect of the system. The static characteristics define what parts the system is made up of.
- **Dynamic:** The behavioral features of a system; for example, the ways a system behaves in response to certain events or actions are the dynamic characteristics of a system.

Similarly, object-oriented design can be carried out in two ways: at the static level or at the dynamic level. According to [40], at the static level, a design is formulated as a number of *classes* which are related to each other in a

number of ways. The classes may contain attributes and methods. At the dynamic level, a design is formulated as a number of *objects* that are related to each other, and which interact with each other by sending messages. Our work focuses on the dynamic level, where the designer of the system has to think in terms of objects, object relations, and object interactions.

The UML diagrams that fall under the category of “dynamic” are: state machine diagrams, activity diagrams, sequence diagrams and communication diagrams. Among these, we are going to focus on two of these diagrams that are often used during the design phase: the sequence diagram and the state machine diagram.

The current version of UML supports concurrency to a certain extent. There are concurrency considerations of some kind in most UML diagrams type, as well as in OCL (Object Constraint Language). UML supports three levels of concurrency description [55]:

1. *system level* ensured by the concurrency between objects (several objects may run concurrently at a time);
2. *object level* where an object may perform various things at a time (various operation executions, and the state machine execution may co-exist);
3. *operation level*, which can be described by a state machine with concurrent states, or by an internally concurrent action.

Although several additions and improvements have been made from one version to another, there is still a long way before UML can be considered a notation perfectly suited for concurrent, real-time systems.

Chapter 3

Relationships between Scenarios

3.1 From use cases to scenarios to sequence diagrams

Scenarios represent a concept often used during the requirements analysis phase, as well as during the design phase of developing a system. When it comes to the requirements analysis, its main task is to generate specifications that describe the behavior of a system unambiguously, consistently and completely [1]. *Use cases* are a widespread practice for capturing the requirements in numerous software processes, particularly the functional requirements. They are a means of communicating with users and other stakeholders about what the system is intended to do. Use cases capture who does what with the system, for what purpose, without dealing with system internals. A complete set of use cases specifies all the different ways to use the system, and therefore defines all that is required of the system [2]. Use cases provide a high-level view of the requirements of the system.

A *scenario* is an instance of a use case, and represents a single path through the use case. Thus, one may construct a scenario for the main flow through the use case, and other scenarios for each possible variation of flow through the use case (e.g., triggered by options, error conditions, security breaches etc.) [2]. Consequently, for one use case, we will have several different possible scenarios.

According to [6], a scenario represents a sequence of user-system interactions representing a system transaction or a system function from a user's perspective. Some of the reasons scenarios receive so much attention in requirements engineering are: they describe the externally visible behavior of a system only (avoiding premature design solutions); they describe how users work with a system, therefore are easy to validate with these users; they can be used both for elicitation and description of requirements; last, but not least, they inherit some of the advantages of natural language, without their disadvantages [11].

UML provides a graphical means of depicting scenarios using *sequence diagrams*. Sequence diagrams are commonly used for both analysis and design purposes [28]. They typically show a user or actor, and the objects and components they interact with in the execution of a use case [43]. One sequence diagram typically represents a single use case scenario or flow of events. The sequence diagram shows the interactions between objects in the sequential order that those interactions occur. During the design phase, architects and developers can use the diagram to force out the system's object interactions, thus flushing out overall system design [3].

In the following, we present a simple example of a use case, followed by one scenario for the use case and the sequence diagram that represents the scenario.

Let us consider a simple ATM system, where a user inserts an ATM card into an ATM machine; the card has to be authenticated with the bank. The user can then perform the desired transaction (withdrawal, deposit, transfer or enquiry). The bank reports that the customer's transaction is disapproved in case the user introduces an invalid PIN. Fig. 3.1 represents a use case diagram for the ATM example (based on the example in [41]). In Fig. 3.2 a textual representation (in natural language) of one scenario of the use case is described. Finally, Fig. 3.3 illustrates the corresponding sequence diagram for the scenario in Fig. 3.2.

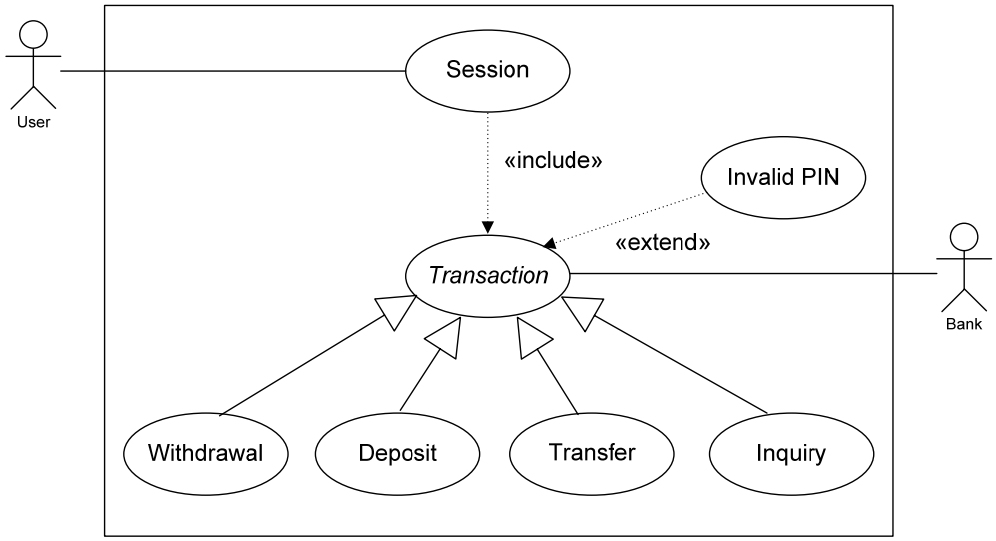


Fig. 3.1 Use case diagram for an ATM system

Scenario: Card and password verification

The user approaches the ATM, where a main screen is displayed.
The user inserts the card and then he is prompted to insert the password.
The user inserts the password.
The account is verified with the bank and the consortium of banks.
The account is authenticated.
An options menu is presented to the user.

Fig. 3.2 One scenario for the "Session" use case

Sequence diagrams are an excellent way to document usage scenarios and to both capture required objects early in analysis and to verify object usage later in design [43]. We can affirm that in the software development process, because we start from specifying the requirements of the system, sequence diagrams are the earliest well defined / formally specified concepts used. They show the requirements of a system in a formally defined way, with precise notation, as opposed to use cases, which show a high-level view of the

requirements, or scenarios, which are more or less described using natural language. This is one of the reasons why we decided to use sequence diagrams as our starting point for model transformation.

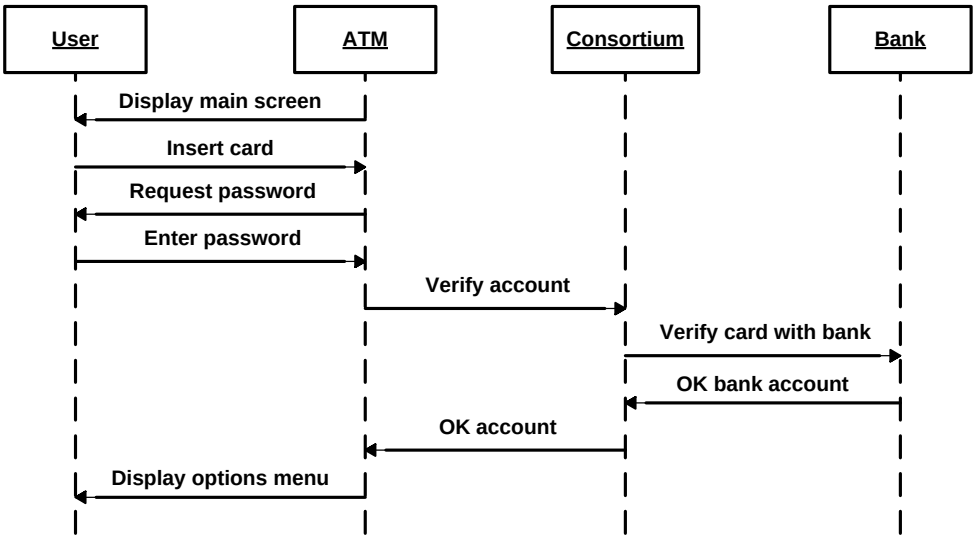
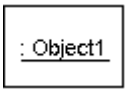
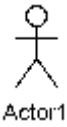
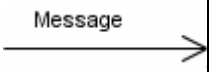


Fig. 3.3 Sequence diagram corresponding to the scenario in Fig. 3.2

3.2 Sequence diagrams as representation of scenarios

A sequence diagram shows objects communicating with other objects and the messages that trigger these communications. A sequence diagram lists objects horizontally and time vertically, and models these messages over time. In a sequence diagram, classes and actors are listed as columns, with vertical lifelines indicating the lifetime of the object over time [38]. The basic notation used in sequence diagram is summarized in Table 3.1. In its simplest form, a sequence diagram looks like in Fig. 3.4.

Table 3.1. Sequence diagram basic notation

Object	Objects are instances of classes, and are arranged horizontally. The pictorial representation for an Object is a class (a rectangle) with the name prefixed by the object name (optional) and a semi-colon.	
Actor	Actors can also communicate with objects, so they too can be listed as a column. An Actor is modeled using the ubiquitous symbol, the stick figure.	
Lifeline	The Lifeline identifies the existence of the object over time. The notation for a Lifeline is a vertical dotted line extending from an object.	
Activation	Activations, modeled as rectangular boxes on the lifeline, indicate when the object is performing an action.	
Message	Messages, modeled as horizontal arrows between Activations, indicate the communications between objects.	

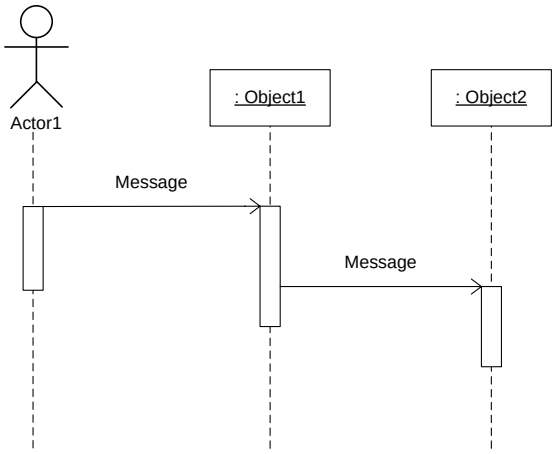


Fig. 3.4 A simple sequence diagram

In the following we give some formal definitions for a sequence diagram.

Definition1

A sequence diagram is a structure (O, M, <), where:

- O is the set of all objects appearing in the sequence diagram;

- M is the set of all messages exchanged between objects;

- < shows a partial ordering of the messages.

Definition 2

M represents the set of messages.

Each message is a tuple (M_{ijk}, N, W [,G]), where:

- M_{ijk} is the k^{th} message originating in object i and going to object j;

- N is the name attached to the message;

- W is the type of message; W ∈ {0, 1, 2} (0: simple message, 1: synchronous message, 2: asynchronous message);

- G is the guard attached to the message.

As we have shown using square brackets, not all messages necessarily have a guard attached to them. (A guard attached to a message shows that the message is sent only if the condition included in the guard is met).

A *self message*, that is a message sent by an object to itself, can be easily identified, since for it we have i=j.

Lost messages are those that are either sent but do not arrive at the intended recipient or which go to a recipient not shown in the current sequence diagram. They are denoted by setting j=0.

Found messages are those that arrive from an unknown sender or from a sender not shown in the current sequence diagram. They are denoted by setting i=0.

We have decided to focus only on the fundamental aspects of the sequence diagrams and we have introduced a few simplifications, therefore not using certain features. These simplifications however do not diminish the

expressiveness of the sequence diagrams. (For example, throughout our thesis, we are going to represent the lifeline in a simple manner, without the activation bar.)

For each sequence diagram, we will have a corresponding *scenario matrix*. A scenario matrix is obtained by representing all the interactions inside a sequence diagram, and it includes all the messages exchanged between objects. (We have decided to use the term “scenario matrix” instead of “sequence diagram matrix” for two reasons. The first one is the fact that we want to emphasize the idea that a sequence diagram is a representation of a scenario. The second reason is purely a language reason, the chosen expression is shorter.)

Definition 3

For a given sequence diagram, a scenario matrix is an ordered list of all message tuples (M_{ijk}, N, W [,G]) belonging to the sequence diagram.

The scenario matrix corresponding to the sequence diagram in Fig.. 3.3 (denoted as S_1) appears in Fig. 3.5:

$$S_1 = \left\{ \begin{array}{l} M_{211}, \text{Display\_main\_screen}, 1 \\ M_{121}, \text{Insert\_card}, 1 \\ M_{212}, \text{Request\_password}, 1 \\ M_{122}, \text{Enter\_password}, 1 \\ M_{231}, \text{Verify\_account}, 1 \\ M_{341}, \text{Verify\_card\_with\_bank}, 1 \\ M_{431}, \text{OK\_bank\_account}, 1 \\ M_{321}, \text{OK\_account}, 1 \\ M_{213}, \text{Display\_options\_menu}, 1 \end{array} \right\}$$

Fig. 3.5 Scenario matrix for the sequence diagram in Fig. 3.3

We should note here that throughout our thesis we often use the term “scenario”, along with the term “sequence diagram”. The former term suggests the actual use of the system, a non-formal concept, while the latter means the actual representation of a scenario as a sequence diagram.

3.3 Normalization of sequence diagrams

In their original form, as they are constructed from scenarios (use cases), two or more sequence diagrams can overlap. *Overlapping* denotes the fact that a common sequence of messages can be found in two (or more) sequence diagrams. We are going to *normalize* the given sequence diagrams and for this we are going to *remove the overlapping* that might exist between them.

The reason we want to remove the overlapping is that we consider that it is important to maintain the property of having distinct, individual sequence diagrams. Fig. 3.6 shows two generic sequence diagrams, A and B, containing a common part, denoted as C. We perform the normalization of sequence diagram by removing the common part and making it into a new sequence diagram, while the remaining non-common part of the initial sequence diagram will be kept unchanged. In our generic example here we separated the common part of sequence diagrams A and B into a new sequence diagram, named C, while keeping the rest of A, that is A1, unchanged. As for sequence diagram B, after the separation of the common part C, we are left with sequence diagram B1 and sequence diagram B2. They each contain the messages exchanged previous to the common messages and following the common messages, respectively. Thus, after removing the overlapping, instead of the two initial sequence diagrams, A and B, we will have 4 sequence diagrams, A1, B1, B2 and C.

Overlapping is detected by identifying common messages in the sets of messages (in the matrices) of sequence diagrams.

Definition 4

Two messages are considered common to two different sequence diagrams if they are exchanged between the same two objects, they have the same name, and they have the same type, that is: (M_{ijk}, N_1, W_1) and (M_{lmn}, N_2, W_2) are common if: $i=l$, $j=m$, $N_1=N_2$ and $W_1=W_2$.

(Note that *k* and *n* do not have to be equal, since they denote the order of each message in their respective sequence diagrams and naturally these messages can appear at different times in the sequence diagrams.)

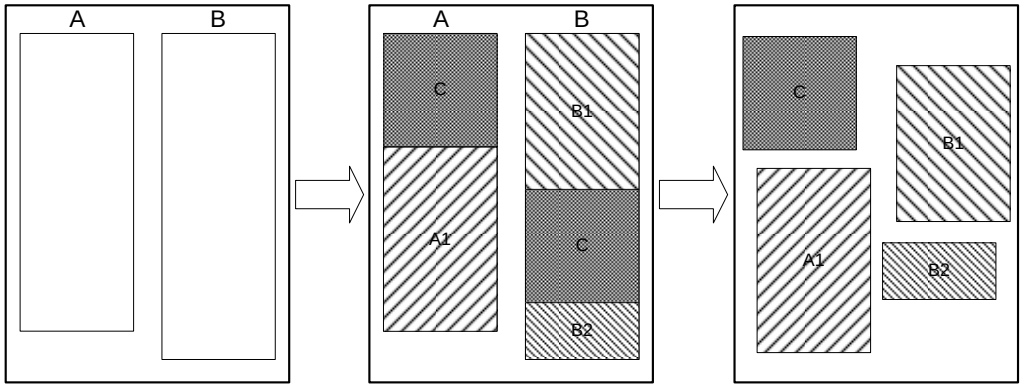


Fig. 3.6 Overlapping between sequence diagrams

Rule 1

A new sequence diagram will be created if a sequence of N consecutive common messages is found in two or more sequence diagrams. The new sequence diagram will comprise of the sequence of common messages identified and these messages are going to be eliminated from each of the sequence diagrams they belong to, thus obtaining disjoint sequence diagrams.

It is important that the messages are consecutive; only if they represent a *continuous sequence* we can say that overlapping exists and only then it is justified to separate them into a new sequence diagram.

Our purpose is to obtain *disjoint* sequence diagrams, i.e. individual, distinct sequence diagrams.

Definition 5

Two sequence diagrams are considered disjoint if they have a maximum number of N-1 consecutive common messages.

We must set up a minimal limit *N* of number of common messages that are going to be separated into a new sequence diagram. This is necessary because on one hand it is too expensive and on the other hand it is not useful to separate a too small number of common messages into a new sequence diagram. Our experience with removing overlapping in sequence diagrams shows that, for practical purposes, a reasonable minimal limit can be set to 5 common messages. Below this number, any sequence of common messages can be ignored.

Rule 2

The process of eliminating overlapping will end when among all the refined sequence diagrams no sequence of N consecutive common messages can be found between any two different sequence diagrams.

Summarizing, the normalization (performed by removing the overlapping), is performed by identifying sets of minimum *N* consecutive common messages in the scenario matrices, separating them, and creating a new sequence diagram from each such common set.

This normalization entails the increase of the number of sequence diagrams than initially existent. However, this brings with it the benefit of "disjoint" (non-overlapping) sequence diagrams. This offers a clearer overview of them and, as we will see later in this chapter, a better overview of the relationships existing between them.

After eliminating the overlapping, we are going to modify the existing scenario matrices and create new ones for the new sequence diagrams, so that each sequence diagram will have its own complete scenario matrix.

As an example, let us consider the same ATM system and two scenarios represented as sequence diagrams for the use case "Transaction" appearing in Fig. 3.1. The first scenario is the one where the user performs a withdrawal and the second one is the scenario where the user performs a deposit. The two sequence diagrams appear in Fig. 3.7.

We notice that both scenarios suppose an initial set of operations where the card is validated with the bank and consortium of banks.

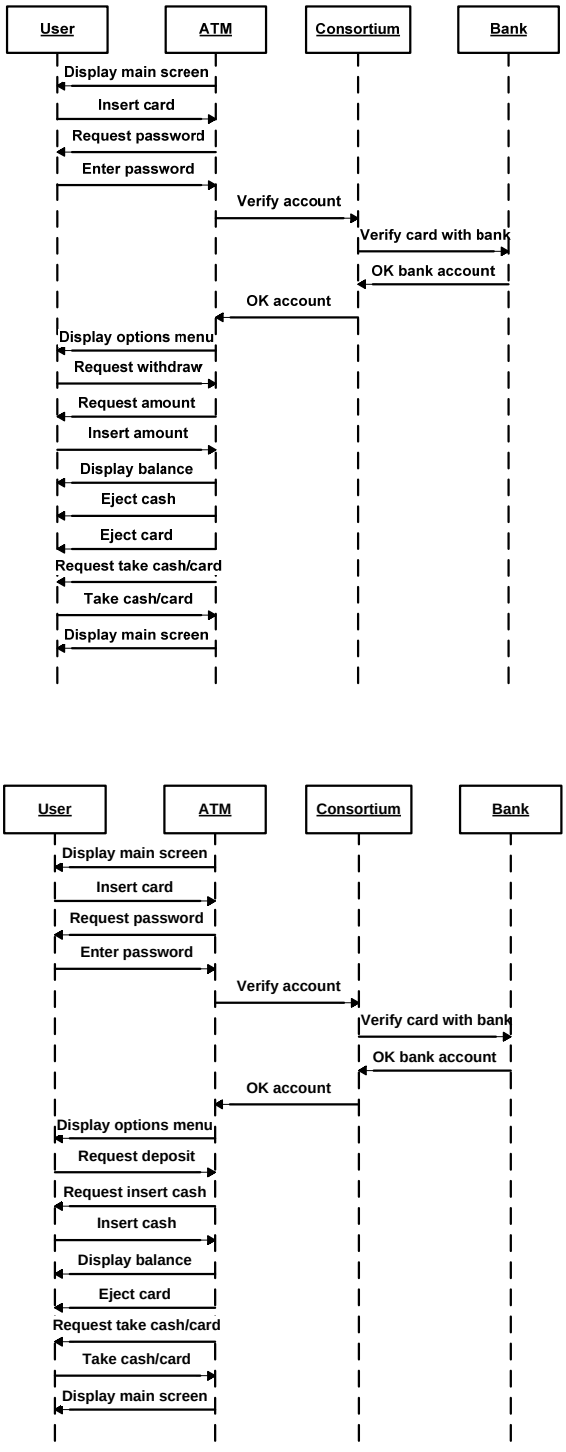


Fig. 3.7 Two sequence diagrams for withdrawal and deposit in an ATM system

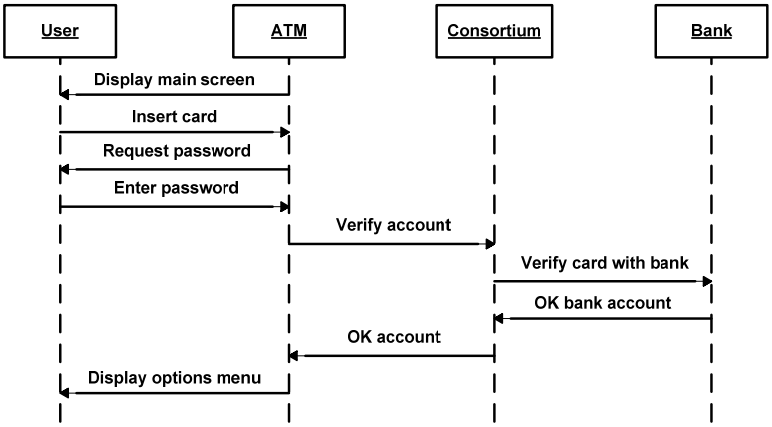


Fig. 3.8 New normalized sequence diagram

More specifically, both sequence diagrams contain a series of 9 consecutive common messages (starting with *Display main screen* and ending with *Display options menu*). This is the overlapping that we are going to eliminate. We will separate the common messages into a new sequence diagram, appearing in Fig. 3.8. The other two sequence diagrams appear in Fig. 3.9. From the original two sequence diagrams we have obtained three normalized sequence diagrams.

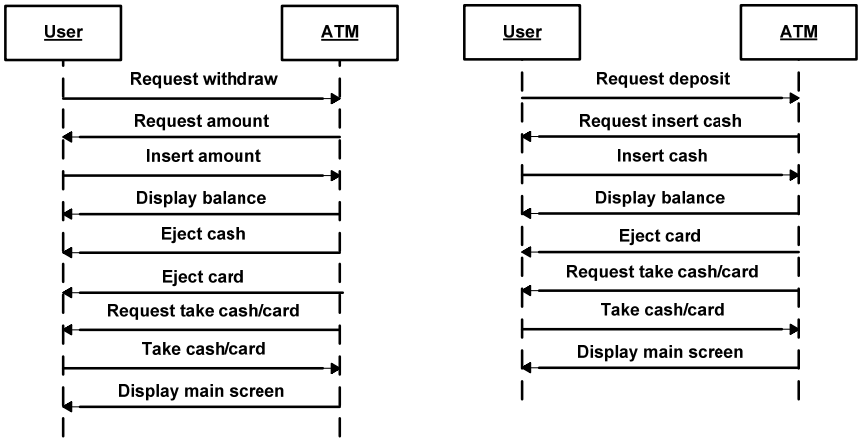


Fig. 3.9 Normalized sequence diagrams

3.4 Importance of relationships between scenarios

One scenario represents only one particular “story” of the use of a system. For the complete description of the requirements specification, a number of scenarios are needed. These scenarios are not independent of each other, but several relationships and dependencies interconnect them.

For example, if we consider the same example of an ATM system, and two scenarios, one for creating a card with a bank, and another one for using the card for ATM operations, it is obvious that the scenario of creating the card must precede the one of performing ATM operations. The other way around would not be possible. This shows that the two scenarios must follow a strict order, there is a strict and clear relationship between them which cannot be ignored.

Sequence diagrams, as representation of scenarios, show the objects and the way they communicate with each other. During the design phase, the behavior of the object must appear clear, unambiguous. The same object can be involved - and most often it is - in several sequence diagrams. Let us imagine, for instance, that we have two scenarios and two different situations: one in which it is required that the two scenarios are concurrent and the other one in which the two scenarios are consecutive. Naturally, the two situations are entirely different and the overall behavior of the objects involved should be clearly different in the two cases.

Different relationships between the scenarios where the object appears result in different overall behaviors of the respective object. It is therefore essential to know the correct relationship between scenarios, the one that reflects the requirements of the system. This is what determines us to emphasize that the relationships between scenarios should be taken into account and should be given a proper representation.

3.5 Classification of relationships between scenarios

Depending on the application, the number of scenarios varies; however small the number of all possible scenarios, relationships and dependencies exist between them. These relationships between scenarios can be of various natures.

Several scenarios can be related with respect to their *goals* [19]. For instance, in an extensive ATM system, one such goal would be “fulfill the requirements to be able to perform bank transactions”. These requirements could be fulfilled in scenarios like “Apply for a cash card”, “Verify user’s personal information”, “Create and send the cash card to the user” etc. The above scenarios are therefore related because they share the same goal.

Scenarios can also be considered to be related in terms of the *resources* they use. For example, we can look at scenarios “Verify user’s previous financial dealings with the bank” and “Verify user’s credit history“ as making use of the same resources, that is the bank files, along with other resources obtained through collaboration between banks.

Based on the criteria above, the related scenarios do not necessarily have to be equivalent, that is they can be related not only as having “the same” goals or “the same” resources, but also “complementary” goals or resources or “opposite” goals or resources.

Several other criteria can be imagined that allow a categorization of the relationships between scenarios (the use cases they belong to, the actors that are involved etc.). Our interest with the scenarios lies in their use during the design phase of the software development process, more specifically in the dynamic aspects that they can reveal. We want to know the behavior of the objects involved in the scenarios, and thus our intention is to emphasize the various dependencies and relationships that would lead us to different object behavior. Consequently, we are interested in a classification of the relationships between scenarios *from the point of view of their influence on the overall behavior of each object involved.*

One object is usually involved in more than one scenario. By definition, dynamic aspects involve a change “over time”. This change refers not only to the behavior inside one scenario, but also to the overall behavior in all the

scenarios involved. Therefore, we have to know exactly how scenarios relate to each other in time, and this translates into *the execution order of scenarios*.

After careful analysis, we have come up with the following categories of dependencies from the point of view of scenario execution order:

- **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);
- **repetition** (the scenario is executed repeatedly).

a) Two scenarios are said to have a *succession* dependency if all the messages in the preceding scenario must be sent/received before all the messages in the following scenario.

For example, the scenario of creating a card with the bank must precede the scenario of performing a transaction with the bank (any transaction involves the use of a card).

b) **Two scenarios are said to have a *disjunction* dependency if either all messages in one scenario or all messages in the other scenario are sent/received.** This establishes an OR type of relationships between the two scenarios and therefore between the two respective sets of messages belonging to the scenarios.

When the user approaches an ATM, after his/her card is verified with the bank, he can choose to perform either one of the following transactions: withdrawal, deposit, transfer or inquiry. Only one of these is possible at a certain moment in time.

c) **Two scenarios are said to have a *conjunction* dependency if their respective sets of messages are sent/received simultaneously** (we do not know how the process happens in terms of individual messages, and this is not relevant; we do not know if the exchange of messages in the two scenarios terminates at the same time, but, with

respect to how we define “conjunction”, the exchange of messages starts at the same time). This establishes an AND type of relationship between the two scenarios.

In most ATM systems, all transactions with the ATM are monitored. This means that the monitoring scenario happens simultaneously with the transaction scenario.

d) **A scenario is *repeated* if the set of messages that are part of it are sent/received repeatedly, according to the conditions that impose the repetition).** This is done either a certain number of times, or until/while a certain condition is fulfilled.

After inserting the card into the ATM machine, the user can introduce an invalid PIN a number of maximum 3 times. The scenario of introducing an invalid PIN repeats itself either until the PIN is correct, or until this is done 3 times. After this, in case his PIN is still not correct, his card is withheld by the machine.

We say that succession, disjunction and conjunction are *dual* relationships, because they involve the existence of at least two scenarios, as opposed to repetition, which refers to a single scenario, and which we call *single* relationship.

We must also take into consideration the situation where a scenario is not related in any way to the other scenarios. This implies that it can occur any time, regardless of the execution time of the others. We call such a scenario an *independent* scenario.

A scenario is *independent* of the other scenarios if no relationship can be established between the timing of sending/receiving of its set of messages and the sending/receiving of the other scenarios’ sets of messages.

For instance, relating to our ATM example, one such independent scenario could be the one updating the ATM software, “Scenario update software”. This is a scenario that can practically occur at any time. It only depends on the decision of consortium of banks that controls the banking activity to update the software; it does not depend on the other possible

scenarios of dealing with the ATM system. A representation of this scenario appears in the sequence diagram in Fig. 3.10.

Summarizing, because we are interested in the effect of the relationships and dependencies between scenarios from the point of view of *their influence on the behavior of the objects involved*, we will consider the following types of dependencies, which, as stated above, can be one of the following: *succession*, *disjunction*, *conjunction* and *repetition*.

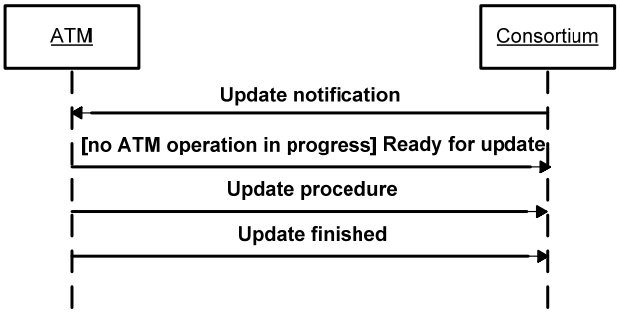


Fig. 3.10 Sequence diagram for "Scenario update software"

3.6 Dependency diagrams

In order to be able to represent and make use of the relationships existing between various scenarios, we have introduced a new type of diagrams called ***dependency diagrams***. The notation used in these diagrams is based on the notation used in Message Sequence Charts [14] and it is illustrated in Fig. 3.11, while Fig. 3.12 shows a general dependency diagram.

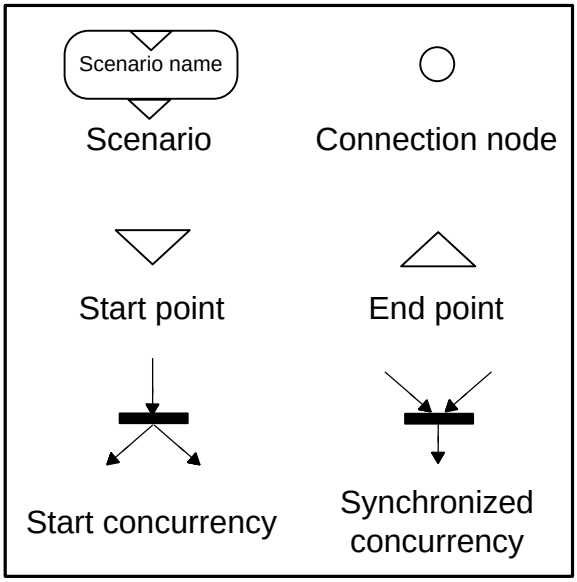


Fig. 3.11 Basic notation in dependency diagrams

Definition 6

A dependency diagram is a directed graph where nodes are scenarios and edges represent the way they are interconnected. If we consider Sc_i *and* Sc_j *two nodes in a dependency diagram, the edge connecting these two nodes represents* ($Sc_i \ \mathcal{R} \ Sc_j$)*, where* $\mathcal{R}$ *denotes the relationship between* Sc_i *and* Sc_j*.*

The relationship $\mathcal{R}$, as defined earlier, can be one of the following:

- sequence, denoted with "**;**"
- disjunction, denoted with "$\vee$"
- conjunction, denoted with "$\wedge$"
- repetition, denoted with "*****"

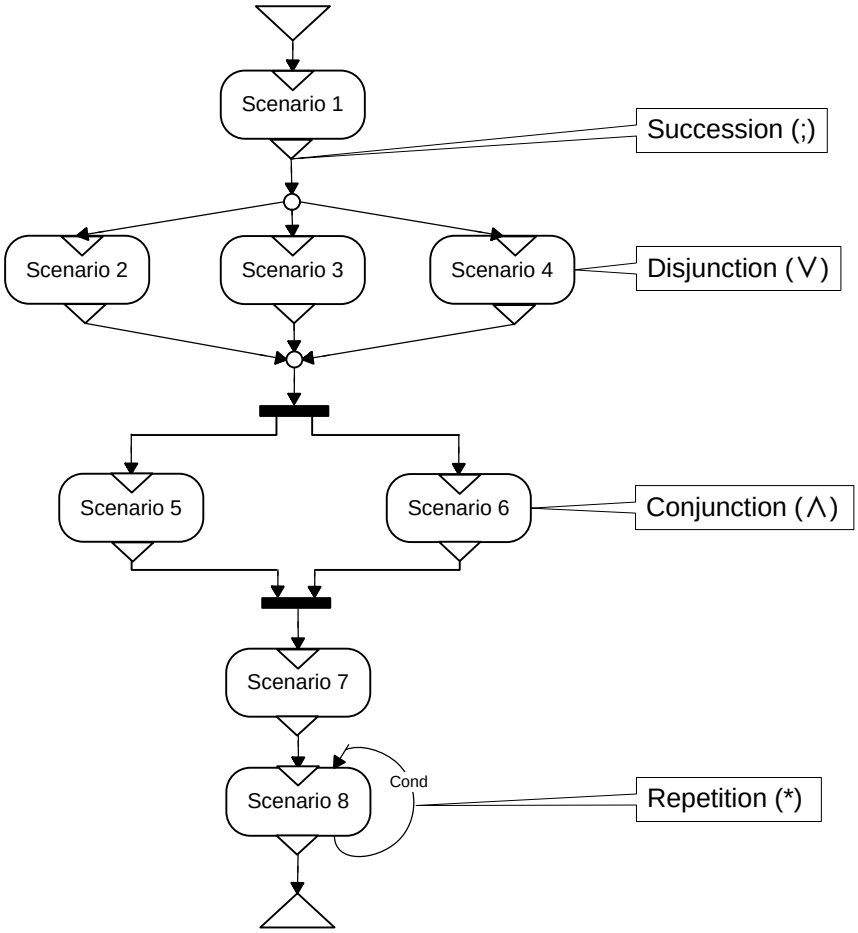


Fig. 3.12 General dependency diagrams

In Fig. 3.12, *Scenario 1* is succeeded by all the other scenarios. *Scenario 2*, *Scenario 3* and *Scenario 4* are related by disjunction, while *Scenario 5* and *Scenario 6* are related by conjunction. *Scenario 8* is repeated according to the condition *cond* appearing on top of the self-message.

A specific example of a dependency diagram is shown in Fig. 3.13. It is based on the same classical example of an ATM system. (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 three 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).

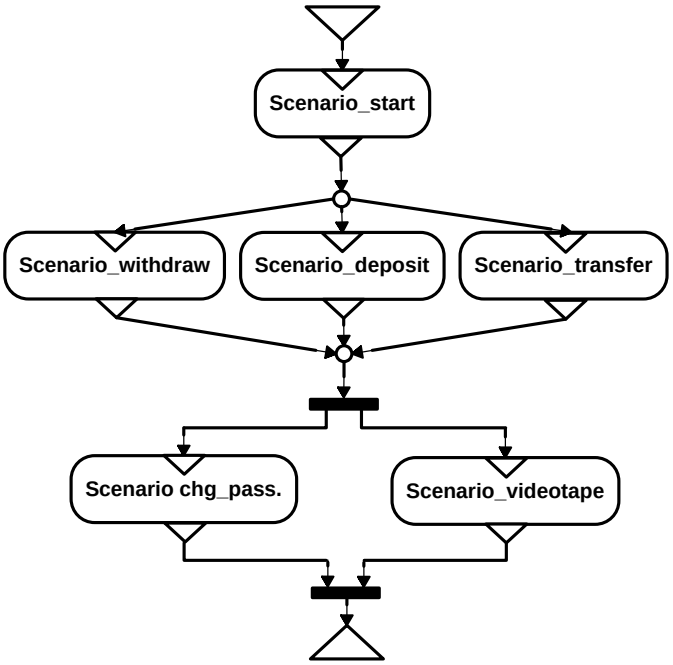


Fig. 3.13 Dependency diagram for an ATM system

Fig. 3.13 exemplifies succession (*Scenario_start* precedes the other ones), the disjunction of three scenarios, *Scenario_withdraw*, *Scenario_deposit* and *Scenario_transfer* (any of them can be executed after *Scenario_start*), as well as the conjunction of two scenarios, *Scenario_chg_ pass.* and *Scenario_videotape*. The above dependency diagram can be written as in the following (we call this textual format a “dependency formula”):

***Scenario_start* ; (*Scenario_withdraw* ∨ *Scenario_deposit* ∨ *Scenario_transfer)* ; (*Scenario_chg_ pass.* ∧ *Scenario_videotape*)**

3.7 Benefits of representing relationships between scenarios

Through the introduction of dependency diagrams as a means to represent relationships between various scenarios, we can benefit from several advantages in the process of requirements analysis and throughout the whole development process of the system. First of all, by knowing the dependencies that exist between scenarios, we can create state machines that show the complete behavior, rigorously according to the information in the scenarios. This is going to appear more clearly in the following chapter, where we will follow exactly the role of the dependency diagrams in the creation of state machine diagrams.

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.

Last, but not least, dependency diagrams could be used in the process of generating test cases; by carefully representing all the possible relationships in the dependency diagrams, we can generate a multitude of test cases by traversing various paths in these diagrams (in the same way as we create traversal paths in any graph, in general). We can easily test unwanted behavior in the dependency diagrams. For instance, if two scenarios are related with a sequence kind of relationship, we can try to execute the succeeding scenario first and the preceding scenario afterwards.

As expected, when dealing with large and complex dependency diagrams, we will have a large number of paths that we can traverse and therefore test. This will make the task of the tester more tedious, but it will increase the chances of discovering inconsistencies and errors in the system. As Dijkstra famously stated more than 30 years ago, “Program testing can be used to show the presence of bugs, but never to show their absence” [23]. We might not be able to find the errors only on the basis of testing, but, nevertheless, by providing the opportunity to derive numerous test cases, we could narrow

down the number of possible inconsistencies and errors in the intended system. An extended description of using depending diagrams in the testing process constitutes a part of our future work.

3.8 Number of dependency diagrams

Ideally, we should have one dependency diagram for a system; this would help in creating a state machine that include the entire behavior throughout the system. Naturally, this dependency diagram is quite possibly very large, since it must include all scenarios. However, this is not always easy to put into practice.

Firstly, we might (and quite often we do) have a number of independent scenarios. These scenarios are not dependent on the other scenarios and therefore we cannot integrate them easily in the dependency diagram. Being independent actually means that the respective scenario can occur at any time. From the point of view of the dependency diagram, an independent scenario could in theory appear anywhere in it (in the very beginning, in the end, concurrent with other scenarios etc.). One solution would be to include one instance of it as an alternative scenario next to each of the other scenarios in the system (obtaining this way a multitude of ORs), but for a system including many scenarios, this would clutter the dependency diagram. Another solution would be not to integrate this scenario in the dependency diagram, and obtain a separate state machine for it.

Secondly, some groups of scenarios, although related to one another, might not be related to another group of scenarios; therefore, it might be more practical to have one dependency diagram for each such group.

These considerations might lead us to being forced to construct not one, but two or more dependency diagrams. We would have several dependency diagrams that comprise of several scenarios interrelated. This would show how the scenarios depend on each other and would help in obtaining final state machines for each object. Since one final state machine corresponds to one dependency diagram, more than one dependency diagram would result into

more that one “final” state machine diagram. This is not the ideal case; however, our purpose is to obtain state machines showing the *complete* behavior of the respective objects. Since we cannot achieve this using only one such diagram, we would obtain in the end as many state machine diagrams as dependency diagrams.

3.9 Scenarios involved in more than one relationship

In practice, it is sometimes possible that a certain scenario is related to two (or more) other scenarios in two different ways. In order to keep the atomic nature of each relationship, we will consider the scenarios involved in more than one relationship *as many times as necessary*.

The scenario will keep the same name, but it will appear twice (or as many times needed) in the dependency diagrams, as we have illustrated in Fig. 3.14.

In our example, scenario S3 appears in two different relationships: $\mathcal{R}1$ with scenario S2 and $\mathcal{R}2$ with scenario S5; we have therefore considered it twice in our dependency diagram. (The relationships between the other objects are not of interest here; we have denoted them generally with a simple $\mathcal{R}$).

We believe that the benefits of showing each relationship and therefore, possibly, including certain scenarios several times, outweigh the burden of increased complexity (and implicitly increased size) of the dependency diagrams. We can have a clear view of the whole system, of all the scenarios in the system, and we persist in our purpose of emphasizing the dependencies that exist between all scenarios in the system.

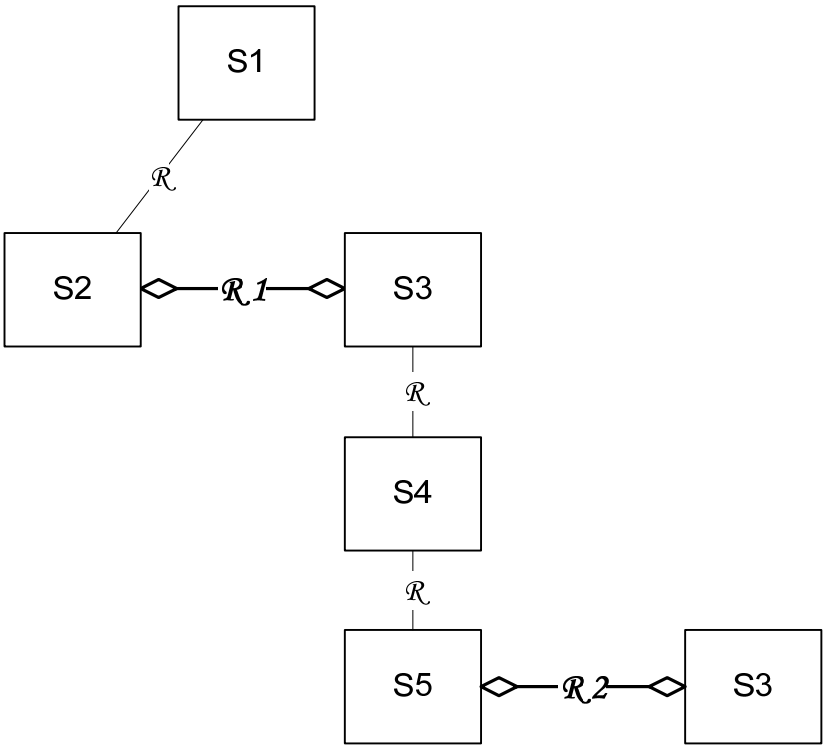


Fig. 3.14 One scenario involved in two different relationships

3.10 Related work

In [11], Ryser and Glinz have introduced a new kind of chart, dependency chart to model the dependencies between scenarios. This chart is mainly used for systematical testing of the requirements. The classification of the possible dependencies given in [11] is the following: *abstraction, temporal* and *causal*. Scenarios arranged in hierarchies and scenarios to cover variants (e.g. the same scenario with various slight differences is true for a system depending on hardware configuration) establish abstraction dependencies. Temporal dependencies establish a sequence dependency between scenarios. If one scenario may only be executed under certain conditions and another scenario establishes these conditions, then the two are related by a causal dependency.

The abstraction relationship is not significant in our purpose of

emphasizing relationships that lead to different behavior of the objects involved. As for the causal relationship, from our point of view, it can be reduced to a succession dependency (as defined in section 3.5). The scenario where the condition has to be fulfilled must precede the scenario that is depending on the condition.

The dependency chart introduced by Ryser and Glinz is mainly used for systematical testing of the requirements.

Breitman and Leite have given a different classification of scenarios relationships in [29], used in the scenario construction process. The relationships were related to the incidence of some scenario components, like goal, context, resources, episodes, actors. Depending on these, several scenario relationships have been defined: *complement*, *equivalence*, *subset, precondition, detour*, *exception, include* and *possible precedence*.

This classification, as well, does not serve our purpose of identifying the overall behavior of objects based on how the scenarios are related to each other. The relationships defined in [29] are used in a scenario evolution process proposed by the authors.

Chapter 4

Transforming Sequence Diagrams into State Machine Diagrams

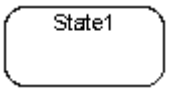
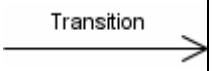
4.1 State machine diagrams in UML

Sequence diagrams (as representation of scenarios) are not only capable to capture requirements, but they can also be used in conjunction with other models, especially behavior models. State machines (particularly statecharts, originally introduced by Harel [4]), represent a compact and elegant way of describing behavior. They can be used not only for behavioral requirements specifications, but also for detailed design models close to implementation [5].

The formalism used in UML for representing state machines is inspired from Harel's statecharts [4]. A state machine diagram, as it is called in UML, models the behavior of a single object, specifying the sequence of events that an object goes through during its life cycle in response to events [43].

The basic notation used in a state machine diagram is summarized in Table 4.1 [38], while a simple example of a state machine diagram can be found in Fig. 4.1.

Table 4.1. State machine diagram basic notation

State	The State notation marks a mode of the entity, and is indicated using a rectangle with rounded corners, and the state name written inside.	
Transition	A Transition marks the changing of the object State, caused by an event. The notation for a Transition is an arrow, with the Event Name written above, below, or alongside the arrow.	
Initial State	The Initial State is the state of an object before any transitions. For objects, this could be the state when instantiated. The Initial State is marked using a solid circle. Only one initial state is allowed on a diagram.	
Final State	End States mark the destruction of the object whose state we are modeling. These states are drawn using a solid circle with a surrounding circle.	

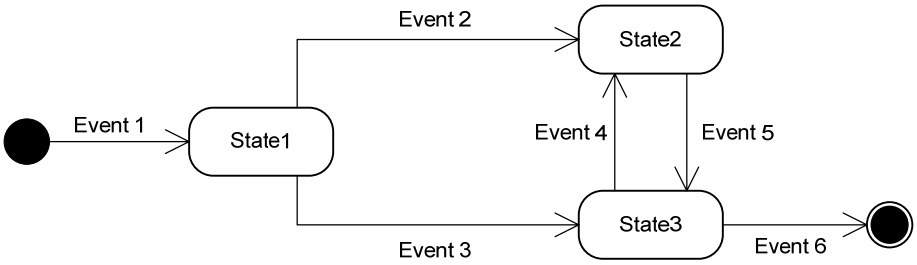


Fig. 4.1 Example of a state machine diagram

State machine diagrams have proved to be useful in the dynamic description of the behavior of a system. Together with class diagrams, they can express the design model of the system and they can be used for code generation, since each of them describes the complete behavior of one object.

4.2 Combining sequence diagrams and state machine diagrams

Sequence diagrams and state machine diagrams each present benefits on their own, especially when it comes to illustrating behavioral aspects of a system. Using them together augments their benefits; they are able to complement each other in many ways. While a sequence diagram represents a single trace of behavior of a complete set of objects, a state machine diagram represents the complete behavior of a single object. The two concepts together provide an orthogonal view of a system.

A state machine shows the behavior of one object; however, it is not always easy to construct a state machine. The information contained in sequence diagrams (as representations of scenarios) can be used for obtaining state machines that depict the specified behavior.

The information regarding the requirements is part of the scenario information. By transforming scenarios (more specifically, sequence diagrams representing scenarios) into state machines, we respect the requirements and we can also understand the behavior of each object. The information in the state machines (together with the class diagrams) can be used during detailed design; moreover, code can be generated from the state machines. Thus the use of both sequence diagrams and state machines supports a considerable part of the software development process, that is the requirements analysis, design and the implementation. As our main interest lies in the design phase, particularly in the dynamic modeling, both sequence diagrams (as representation of scenarios) and state machine diagrams are extremely useful.

4.3 Transformation process of sequence diagrams into state machine diagrams

In order to make use of the benefits of both scenarios and state machines, we propose a process of transformation of sequence diagrams (as representations of scenarios) into state machines, based on the information in the dependency diagrams.

Our analysis shows that different relationships between scenarios result in different state machine structures. Since state machines will further be used during design or for the implementation phase, we must obtain the state machine structures that reflect accurately the behavioral information contained in the scenarios. This can only be done if we know exactly how the scenarios are related to each other and it is the dependency diagrams that are capable of depicting these relationships. Therefore, it would not be possible to create state machine diagrams that accurately reflect the behavior of the objects involved in scenarios if we did not consider the information in the dependency diagrams, that is if we did not know how these scenarios are related. Consequently, in our process of transformation of sequence diagrams into state machine diagrams, we will rely extensively on the dependency diagrams.

Given a number of scenarios (obtained in the requirements specification phase of developing a system), our goal is to emphasize all the relationships existing between these scenarios and then use them in the transformation process of the sequence diagrams (as representation of scenarios) into state machine diagrams.

The *process of transformation* of sequence diagrams into state machine diagrams involves three major phases, as follows:

Phase I - Identification and representation as sequence diagrams of all single scenarios (called *initial* scenarios) and normalization of the sequence diagrams;

Phase II - Identification and representation (as dependency diagrams) of the relationships between all scenarios;

Phase III - Synthesis of the state machine diagrams (one for each object), based on the information acquired in the previous two phases.

Phase III involves the following two steps, *for each object* in the system:

III.1. Creation of *initial* state machine diagrams, one diagram corresponding to each scenario where the object appears;

III.2. Creation of the final state machine diagram by combining all the initial state machine diagrams, based on the information in the dependency diagrams.

The result of the transformation process will be *a number of state machine diagrams equal to the total number of objects* appearing in the scenarios. An overview of the process is illustrated in Fig. 4.2. We will describe each phase in detail in the following.

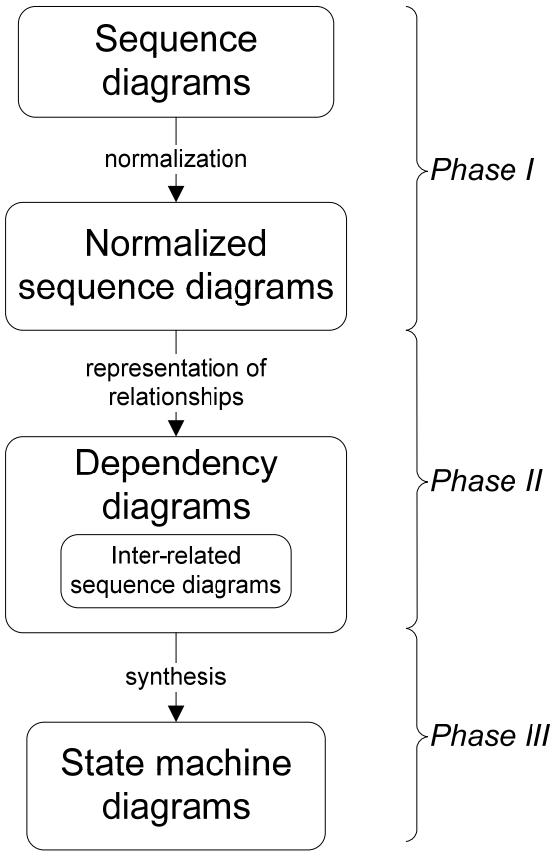


Fig. 4.2 Overview of the transformation process

4.3.1. Phase I: Identification and representation as sequence diagrams of all single scenarios and normalization of the sequence diagrams.

During this first phase, we create what we call “initial” scenarios, by writing down all the possible scenarios of using the system. These scenarios are represented as UML sequence diagrams. For each sequence diagram, we will have a corresponding scenario matrix.

In case overlapping between scenarios exist, as we have shown in section 3.3, the initial sequence diagrams have to be normalized. This is done in preparation for phase II, where the dependencies between disjoint, non-overlapping scenarios are going to be illustrated. If we consider $N_{initial}$ the number of initial scenarios, after the normalization we will have N scenarios, where $N \geq N_{initial}$ (N and $N_{initial}$ are equal only in case there was no overlapping between scenarios and therefore no newly created scenarios).

Result of phase I: a number of N scenarios (N normalized sequence diagrams and an equal number N of scenario matrices).

4.3.2. Phase II: Identification and representation (as dependency diagrams) of the relationships between all scenarios.

During this phase, we will show the relationships between scenarios by creating dependency diagrams. As noted in section 3.6, the related scenarios will be the nodes in the dependency diagram, while their relationship will be the edges. At this stage we need to identify all the possible relationships that exist between the given scenarios.

When we identify the relationships, it is important that they *do not involve a “subpart” relationship*.

Definition 7

A subpart type of relationship denotes the fact that a relationship exists between a subpart of one scenario and a subpart of another scenario.

For example, let us consider an abstract representation such as the one in Fig 4.3a, with two initial scenarios *A* and *B*, and let us consider that when we try to identify the relationships between various scenarios in our system, we discover that a relationship exists between a subpart of scenario *A* and a subpart of scenario *B* ($\mathcal{R}$ denotes this relationship, which can be any of the dual relationships described in section 3.5). This is not desirable in our methodology, because we need clear relationships that show exactly how individual scenarios, not subparts of scenarios, are related.

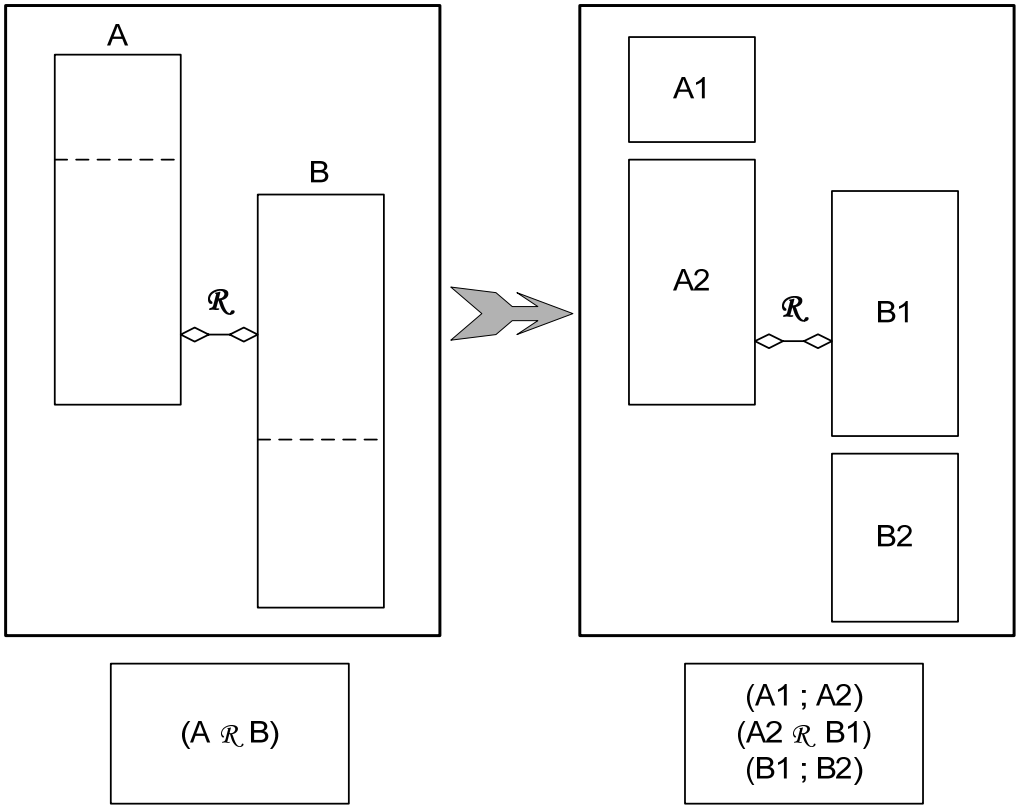


Fig. 4.3 Abstract representation of a subpart type of relationship

In case of such a subpart type of relationship, each of the involved scenarios is going to be divided into two parts, such that the relationship appears clearly as relating two individual scenarios. We will thus divide scenario A into scenarios A1 and A2, and we will divide scenario B into scenarios B1 and B2, as in Figb. We will have 4 scenarios instead of the two original ones, with relationship $\mathcal{R}$ existing between A2 and B1. We must keep in mind that A1 precedes A2 and B1 precedes B2; this preserves the original behavior existing inside scenarios A and B.

We will break the scenario matrices of each scenario into two parts, corresponding to the newly created scenarios, and thus obtain 4 new scenario matrices, one for each newly created scenario. This is a straightforward operation. By making no modifications to the message tuples and to their order in the newly created matrices, we are making sure that the behavior of the original scenarios is preserved after the division into two new scenarios.

In case the relationship $\mathcal{R}$ represents disjunction or conjunction between A2 and B1, the dependency diagram illustrating their relationship appears in Fig. 4.4b. This can be expressed as: ***(A1 ; (A2 $\mathcal{R}$ B1) ; B2)***, that is A1 precedes (A2 $\mathcal{R}$ B1), which is then followed by B2. ($\mathcal{R}$ is either $\vee$ or $\wedge$).

In case $\mathcal{R}$ represents a succession relationship, we will have the overall dependency being: ***(A1 ; A2 ; B1 ; B2)***. This appears in Fig. 4.4a. It is arguable that, in case we have a succession relationship between A2 and B1, there is actually no practical reason to divide the scenarios A and B into the above two parts. ***(A1 ; A2 ; B1 ; B2)*** is equivalent to ***(A ; B)***.

It is also possible to find out that a subpart of a scenario (not the whole scenario) involves a repetition. In this case we will separate this part and illustrate the fact that it is repeated, as in Fig. 4.5.

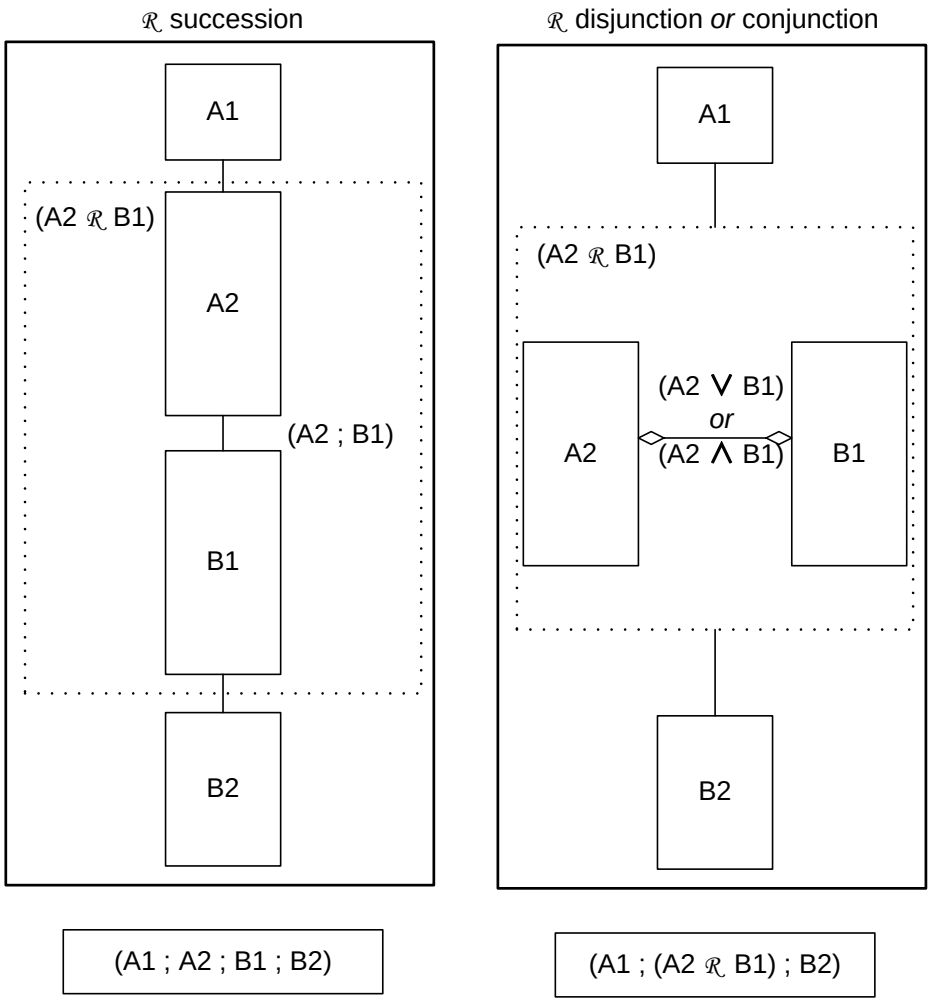


Fig. 4.4 Subpart relationships:
succession, disjunction, conjunction

The process of creating dependency diagrams can thus be described as follows:

a. *Identify dependency between scenarios*

b. While *a subpart relationship exists between scenarios*

> ***b1***. *Partition the involved scenarios (break them down into two or more new scenarios)*
>
> ***b2***. *Identify another dependency*

c. *Finish the creation of dependency diagrams.*

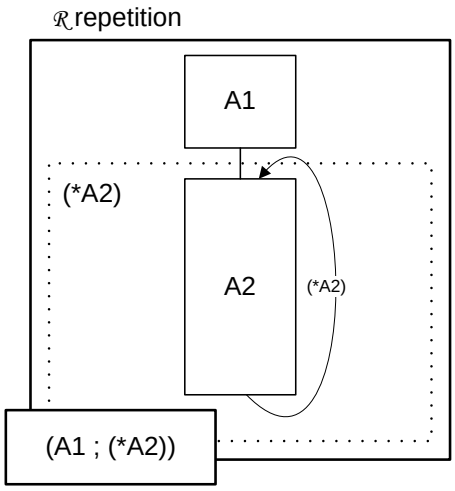


Fig. 4.5 Supbart relationship: repetition

It is important to emphasize that while eliminating subpart relationships we must preserve the behavior of the initial scenarios. The initial scenarios are those that reflect the requirements of the system and it is crucial that we do not alter these requirements.

By eliminating subpart relationships (in case there are any), the number of the scenarios resulting from phase I has increased. We will therefore have a new number of scenarios, N`, where $N` \geq N$ (the two numbers are equal only in case there was no "subpart" relationship involved and therefore no newly created scenarios).

Result of phase 2: a number $N` \geq N$ of normalized sequence diagrams (along with an equal number of scenario matrices) and a number D_d of dependency diagrams.

4.3.3 Phase III: Synthesis of the state machine diagrams

The following description applies in the same way for each object appearing in the set of given scenarios.

a) Phase III.1. Creation of initial state machines

For a given object, we will first identify all the scenarios where the object appears. For each of these scenarios, we will create one state machine diagram; these diagrams are called “initial” state machine diagrams. Therefore, for one object, we will have a number of initial state machine diagrams equal to the number of scenarios where that object appears. (Scenarios being represented as sequence diagrams, once again we are using the terms “scenarios” and “sequence diagram” interchangeably.)

Sequentially, we have to do the following:

- *create empty state machine diagrams, one for each sequence diagram where the object appears;*
- *for each of these state diagrams, do the following, extracting information from each sequence diagram, one by one:*
  - *for all transitions to the object, create events;*
  - *for all transitions from the object, there will be actions that will lead to states: create the respective states;*
  - *set the right time sequence for the transitions.*

For the sake of simplicity, we are not going to include the actions, but only the states that they lead to.

Messages in sequence diagrams can have guards attached to them. We have decided that these guards are going to appear together with the event; therefore they will appear only for the receiving object. In case there is a guard attached to an outgoing message from an object, it will be considered when creating the state machine for the object that receives this message. Fig. 4.6 shows how this is done on a simple example. Fig. 4.6a shows a sequence

diagram with objects O1 and O2, and with messages *Msg 1* and *Msg 2*, each having attached the guards *guard 1* and *guard 2* respectively. As a result, in the state machine for object O_1, [*guard2]* will be attached to the event resulting from *Msg2* (named *E_Msg2*) while *Msg1* will lead to a state (*S_Msg1*). In the state machine for object O_2, *[guard1]* will be attached to the event resulting from *Msg1* (named *E_Msg1)*, while *Msg2* will lead to a state (*S_Msg2*). The state machine illustrating the above appears in Fig. 4.6b.

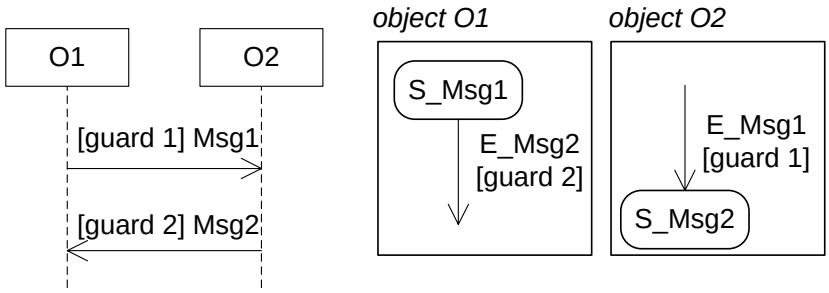


Fig. 4.6 Guards from sequence diagrams to state machine diagrams

In the following, we describe the algorithm for creating initial state machine diagrams. The same algorithm applies to each object that we want to create a state machine for. We should note that a *default state* should be created as being the state in which the object is before it receives any event.

For object O_x, the algorithm appears in Fig. 4.7. The actual output of the algorithm, for object O_x, is a number of ordered lists S_xM_l, with $l = 1,\ldots,N_xSM_{initial}$, that represent the initial state machines for object O_x (this means that O_x appears in $N_xSM_{initial}$ scenarios).

Result of phase III.1: For each object O_i, a number $N_iSM_{initial}$ of state machine diagrams (equal to the number of scenarios where object O_i appears) ($N_iSM_{initial} \leq N$`).

***Input**: a set of N scenario matrices, each representing a scenario (sequence diagram)*

***Output**: a set of $N_x SM_{initial}$ state machines (equal to the number of scenarios where O_x appears)*

1. c=0 (c counts the scenarios where O_x appears)

2. For s=1 to N:

3. find out if O_x appears in the scenario Sc_s, that is search in the scenario matrix of Sc_s: check in all tuples (M_{ijk}, N, W, [G]) if i=x or j=x; if at least one such tuple is found, it means we found a scenario containing O_x and we DO:

4. c=c+1

5. create empty state machine

5.1. create empty list of states&transitions (S_xM_c)

5.2. sc=0 (sc shows position of states/transitions in the ordered list)

5.3. create empty list of states for S_xM_c (S_xLS_c)

5.4. create empty list of transitions for S_xM_c (S_xL_c)

6. find the tuples in which O_x appears; for each tuple (M_{ijk}, N, W, [G]) in the scenario matrix of Sc_s DO:

6.1. if x=i, add one state to the state machine's list of states (S_xLS_c):

6.1.1. sc=sc+1

6.1.2. add element (state, Nijk, sc) to the list (S_xLS_c)

6.2. if x=j, add one transition to the list of transitions (S_xLT_c):

6.2.1. sc=sc+1

6.2.2. add element (trans, Nijk, [G], sc) to the list (S_xLT_c)

7. set the right time sequence of states and transitions – obtain a list S_xM_c of states and transitions, by combining the two lists S_xLS_c and S_xLT_c into one list (S_xM_c), ordering the elements by the value of sc

End

Fig. 4.7 Algorithm for creating initial state machines for object O_x

b) Phase III.2. Creation of final state machines

Based on the information in the initial state machines and the information in the dependency diagrams, we will create the final state machine diagram for the object. This involves the merging of the initial state machine diagrams, according to the relationship between the scenarios that constituted the origin of the initial state machines.

Based on the classification of relationships between scenarios, there are several rules that need to be followed.

Rule 1

If two scenarios where a particular object appears are related with a sequence kind of dependency, then the two corresponding state machines (for the considered object) will follow one another. The state machine corresponding to the scenario that has to be executed first will precede the one corresponding to the succeeding scenario.

Rule 2

The state machines corresponding to two scenarios related by disjunction are going to be unified into one single state machine.

If the initial state machines have common transitions or common states, they will be taken only once in the final state machine.

Rule 3

If two scenarios are related with a conjunction dependency, the initial state machines corresponding to the considered object involved in them will be combined with AND type substates.

Rule 4

If a scenario is repeated, the same repetition can be shown in the state machine of the object we consider. This is indicated in the form of a transition from the terminal state to the initial state of the state machine. The condition imposing the repetition is specified on a label situated on top of the arrow depicting the transition.

When combining the initial state machines, the default states in them will represent a single default state in the final state machine.

Result of phase III.2:

- For each object, one final state machine diagram;
- In total, a number of final state machine diagrams equal to the total number of objects from all scenarios.

4.4 Applications of the obtained state machine diagrams

The transformation of sequence diagrams into state machine diagrams, as a transformation between two different UML models, gives a better view of the system. The obtained state machine diagrams can be used not only for detailed design, but also during the implementation phase. 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. This way the developer is brought one step closer to the final phases of developing the system.

Several tools and research papers that deal with generating code from state machine diagrams exist (e.g. [21], [22]). By obtaining the state machines for all the objects in the system, we offer the developer more behavioral information to help him/her during the implementation phase.

4.5 Comparison: case with dependencies versus case without dependencies

In the following, we are going to make a comparison between a state machine diagram obtained with dependencies and one obtained without dependencies. As a specific example, let us consider a simple CD Player with a remote control device. We suppose that the objects involved here are *Device* (that is the remote control device) and *Controller*, which actually implements

the behavior of the CD Player when the buttons on the device are pressed. Whenever a button is pressed, the *Device* informs the *Controller*, which in response performs some action and changes the state of the player [7].

Let us consider the following two scenarios (represented as sequence diagrams and appearing in Fig. 4.8), called *Speaker* and *Player* and afterwards let us focus on the object *Controller* and try to offer a complete behavior for it; we can do this by creating the state machine diagram that corresponds to it.

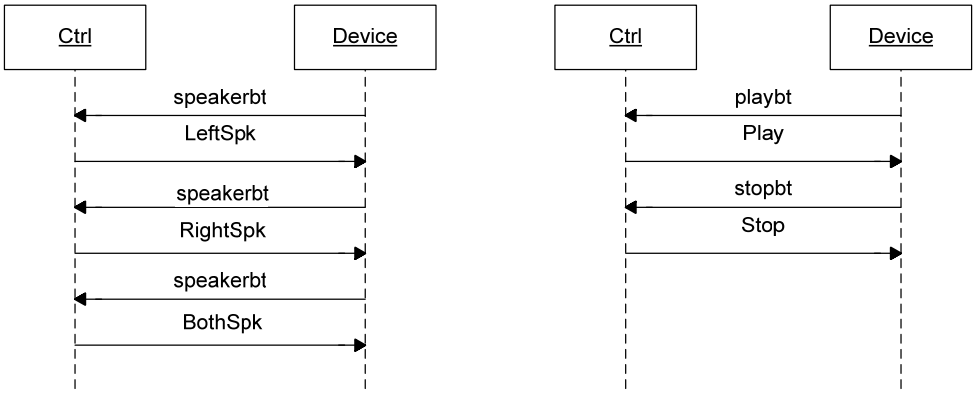


Fig. 4.8 Two scenarios, *Speaker* and *Player*

According to the algorithm we described previously, for object *Controller*, we would obtain the state machine diagrams in Fig. 4.9. a) corresponds to scenario *Speaker*, while b) corresponds to scenario *Player*. Each of these state machines shows the behavior of object *Controller* in two different cases. In these cases we do not consider that the two given scenarios are dependent on one another.

If we want to know the behavior of this object as a whole, we should consider the relationship between the two scenarios. First, let us imagine that the two scenarios follow one another, that is the Speaker scenario occurs first and the Player scenario occurs afterwards. Second, let us imagine that the two scenarios are concurrent. If the two scenarios follow one another (we have a succession dependency), we obtain the state machine diagram in Fig. 4.10. If the two scenarios are concurrent (we have a conjunction dependency), the resulting state machine diagram appears in Fig. 4.11.

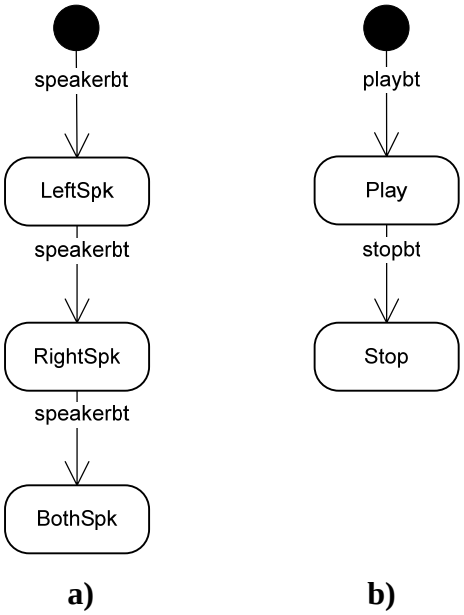


Fig. 4.9 State machine diagrams for scenarios *Speaker* and *Player*

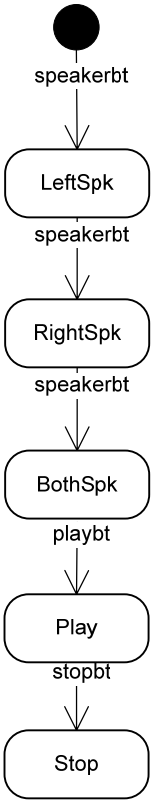


Fig. 4.10 State machine diagram for Controller in case of succession

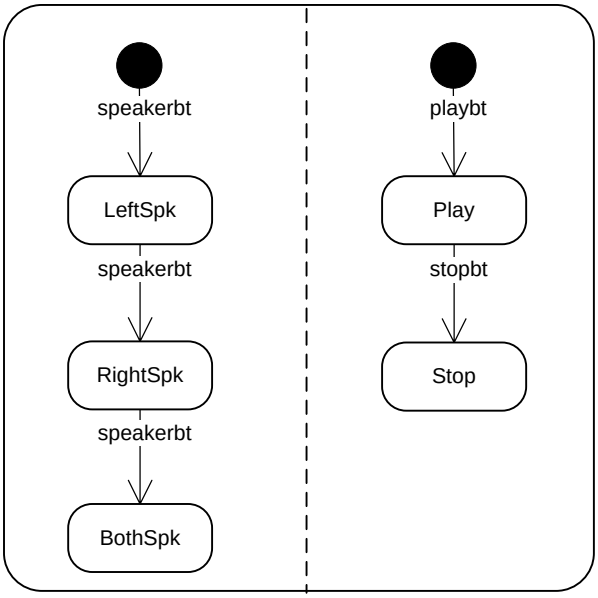


Fig. 4.11 State machine diagram for Controller in case of conjunction

We can notice that there is an important difference between the behavior of the object in the two separate state machine diagrams in Fig. 4.9 and the two state machine diagrams, showing the overall behavior of object *Controller*, in Fig. 4.10 and Fig. 4.11. Furthermore, the behavior of the object differs between Fig. 4.10 and Fig. 4.11. Since the state machine diagram should describe the behavior of the object (that we can actually use for code generation), it is important to decide which of the behaviors is the one that reflects the given requirements (in our case, the given scenarios represented as sequence diagrams).

This simple example proves the fact that, in order to be able to have a complete behavior description for an object, we need to know how to "combine" the individual state machine diagrams obtained. Moreover, because different scenario relationships result in different state machine diagrams (and thus different behaviors of the objects), it is not enough to know that the scenarios are related to each other, but it is important to represent accurately their relationship.

The above example underlines once more that **dependencies between scenarios are crucial** when we need to know the behavior of objects involved in a system. By representing these dependencies in dependency diagrams and by using the information in these diagrams, we can ensure that the requirements expressed in the given scenarios have been fulfilled and the behavior of the objects involved can be illustrated completely and accurately in the synthesized state machine diagrams.

4.6 Related work

4.6.1 Obtaining state machine diagrams from single scenarios

The problem of transforming scenario type models into behavior models like state machines has been dealt with in various research papers. The most related work can be found in [9] and [7].

One of the first papers dealing with this problem is [35], where Koskimies and Makinen developed an algorithm which converts sequence of event trace diagrams into state machines. Event trace diagrams are the representation of scenarios in OMT (Object Modeling Technique), and they have evolved into UML's sequence diagrams. The result of their algorithm is simple state machines, without state hierarchy and concurrent states. Later, a tool called SCED is proposed by Koskimies and others for automatic generation of state machines from single scenarios [9]. They apply the Biermann and Krishnaswami algorithm (used for synthesizing programs from sample executing traces) to the synthesis of state machines.

In [7], Ali et. al. propose a set of rules of constructing statecharts from event trace diagrams; class hierarchy and concurrency are introduced here.

Similarly, Maier and Zundorf consider single scenarios in [27] and offer an algorithm of synthesis of statecharts from sequence diagrams. More recently, in [8], an algorithm for generating UML statecharts from sequence diagrams is given; merging multiple sequence diagrams is discussed, along with how to merge different branches of the resulting state machines, without any regard for

the nature of relationships between the sequence diagrams.

The above research papers deal with transforming scenarios (represented either as event trace diagrams or sequence diagrams) into state machine structures. However, only single scenarios are considered, their relationship and the influence of their relationships on the resulting state machine is not taken into consideration.

4.6.2 Others

Schonberger et al. [10] describe an algorithm for model transformation, more precisely an algorithm for transforming collaboration diagrams into state diagrams; however, they consider only single collaboration diagrams as input. Collaboration diagrams describe the interaction among objects, with the focus on space. (Their name has changed since UML 2.0, they are now called “communication diagrams”). 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 how objects interact and communicate with each other, focus on time. Although the two kinds of diagrams are similar in their purpose of describing dynamic aspects, we favor the use of sequence diagrams in the analysis phase, as they allow a representation of the requirements by focusing on the time flow in the development of events.

Systa and Makinen [46] have developed MAS (Minimally Adequate Synthesizer), as an interactive algorithm that synthesizes UML state diagrams from sequence diagrams. It follows Angluin’s framework of minimally adequate teacher to infer the desired state diagram by consulting the user, with the help of membership and equivalence queries. The algorithm can conclude the correct answer to most of the membership queries without consulting the teacher, i.e., the designer. The UML sequence diagrams represent example cases that can be treated in any order.

Whittle and Schumann [8] introduce a method for automatically generating UML statecharts from a collection of UML sequence diagrams. Since the set of sequence diagrams is usually incomplete, containing insufficient and imperfect information, additional information is often needed

for synthesizing appropriate statechart diagrams. This additional information about the semantics of the messages in the form of pre and post conditions is expressed in Object Constraint Language (OCL) and can be added to the sequence diagrams to guide the synthesis process. Also, the information included in a class diagram can be used to structure a statechart diagram with composite states.

In [26], Some proposes a formalization of use cases, a natural language based syntax for use cases description, and an algorithm that incrementally composes a set of use cases as a finite state transition machine.

In [30], Uchitel provides a synthesis algorithm that translates scenarios into a behavioral specification in the form of Finite State Processes (FSP). He uses Message Sequence Charts (MSCs) as representation of scenarios and high-level MSCs (hMSCs), each representing a set of scenarios into a single graph.

We can actually observe that there exists much work on introducing methods that describe how to specify various models; however, the methods described do not sufficiently guide the developer in the task of transforming one model type into another. More specifically, although several research papers exist that show how to transform scenario type models into behavior models like state machines diagrams, they tend to neglect the relationships that can exist between various scenarios. As we have shown before, our conviction is that these relationships have to be emphasized and taken into consideration, since they influence the behavior in the state machine diagrams that can be obtained based on the information in the considered scenarios. Only by taking into consideration these relationships we can be sure that we obtain the state machine diagrams that reflect accurately and entirely the behavior resulting from the requirements analysis.

Chapter 5

Semiautomatic Synthesis of State Machine Diagrams: MUSEDESK

5.1. Description of the system

Our approach has been implemented into a system named MUSEDESK ("from MUltiple ScEnarios with DEpendencies to State machine diagrams"). Given a set of sequence diagrams (that represent scenarios) and a dependency diagram (that shows the inter-scenario relationships), MUSEDESK is able to generate one state machine for the desired object.

The system has the following input and output:

- *Input*: a number of sequence diagrams, along with a (number of) dependency diagram(s)
- *Output*: a state machine diagram

A number of sequence diagrams are created first; after that, a dependency diagram showing all the relationships that exist between them is created. An object is then selected and afterwards MUSEDESK synthesizes its state machine diagram.

Our system first generates the initial state machine diagrams (as described in the previous chapter) for a selected object. The combined list of

states and transitions is obtained automatically, as well as their order. However, the output on the screen is not done automatically. Only simple sequence diagrams are supported in the current version, as well as simple state machine diagram structures.

The final state machine diagram (as defined in section 4.3) is obtained based on the information in the given dependency diagram. The meta-list of states and transitions is obtained automatically, but the appearance and the layout on the state machine diagram pane are set manually.

5.1.1. Overview

Fig. 5.1 shows the overall structure of the MUSEDESK system. Conceptually, there are 3 main modules that make the system function: *Scenario Manager, Transformation Engine* and *Graphical Editor.*

a) Scenario Manager

This module receives its name from managing the given scenarios (when we say scenarios, we mean their representation as *sequence diagrams*). Here are its main functions:

- It creates the scenario matrices for the given sequence diagrams. As we have shown in section 3.2, for each message exchanged between two objects, a tuple is created, having the form $(M_{ijk}, N, W[,G])$; the ordered set of all tuples represents the scenario matrix.

- Through the process of normalization of the given scenarios, it creates the scenario matrices for the normalized scenarios.

- It creates the dependency formula, that is a textual representation of the given dependency diagram.
  A dependency formula is given as $(S_1 \mathcal{R}_1 S_2 \mathcal{R}_2 \dots \mathcal{R}_{n-1} S_n)$.

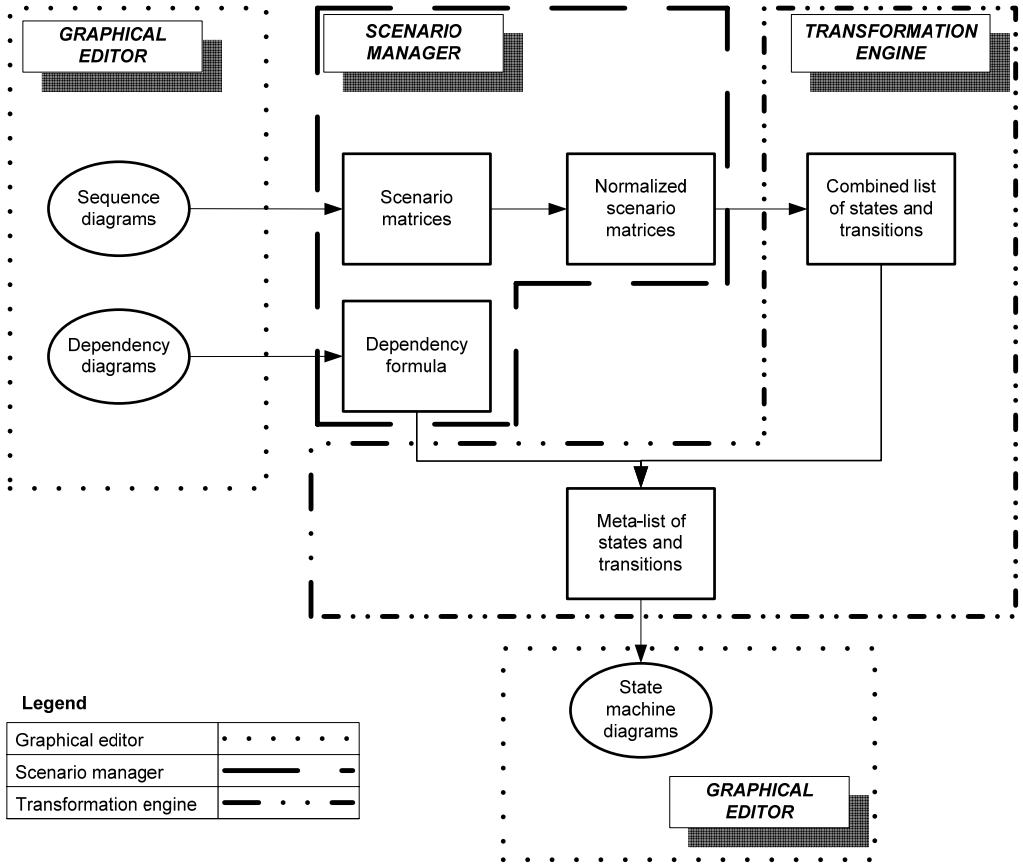


Fig. 5.1 Overview of MUSEDESK

b) Transformation Engine

The transformation engine uses information given by the *Scenario Manager* and creates the state machine diagram.

- It creates the initial state machine diagrams, using the algorithm described in section 4.3.3. It first creates a list of states and a list of transitions and then, according to their order, a set of combined lists of states and transitions: $S_{object}LS_1$, $S_{object}LT_1$…, …, $S_{object}LS_n$, $S_{object}LS_n$
  (n represents the number of sequence diagrams where *object* appears)

- It uses the set of combined lists and transitions and the dependency formula to create a meta-list of states and transitions.

c) Graphical Editor

The graphical editor is the one concerned with the graphical editing of each diagram: sequence diagram, dependency diagram and state machine diagram.

5.1.2. Description of the main window

The main window of MUSEDESK has 3 tabbed panes, one tab for each type of diagram: *Sequence diagram* pane, *Dependency diagram* pane *and State machine diagram* pane.

a) The *Sequence diagram* pane allows the creation and editing of sequence diagrams.

This pane creates and processes the typical elements of the sequence diagram: *objects* and *messages*. One element is created by clicking its corresponding button in the toolbar. The location of the element is decided by the position of the mouse. In case of a message, it will be created as outgoing from the object whose lifeline has been clicked on first and incoming to the object whose lifeline has been clicked on second. By right-clicking the selected element, an input window is shown, where the name can be written. This name will appear inside the rectangle for an object and on the arrow for a message (centered on the message).

This pane contains also typical editing menus, like selecting an element, deleting an element and clearing the whole pane. When selecting an element, it can be moved or its name can be added/changed.

Fig. 5.2 shows a snapshot of this pane being active, with *Scenario_0* created in it. In this snapshot, the name of the message exchanged between *Consortium* and *Bank* is just being inputted.

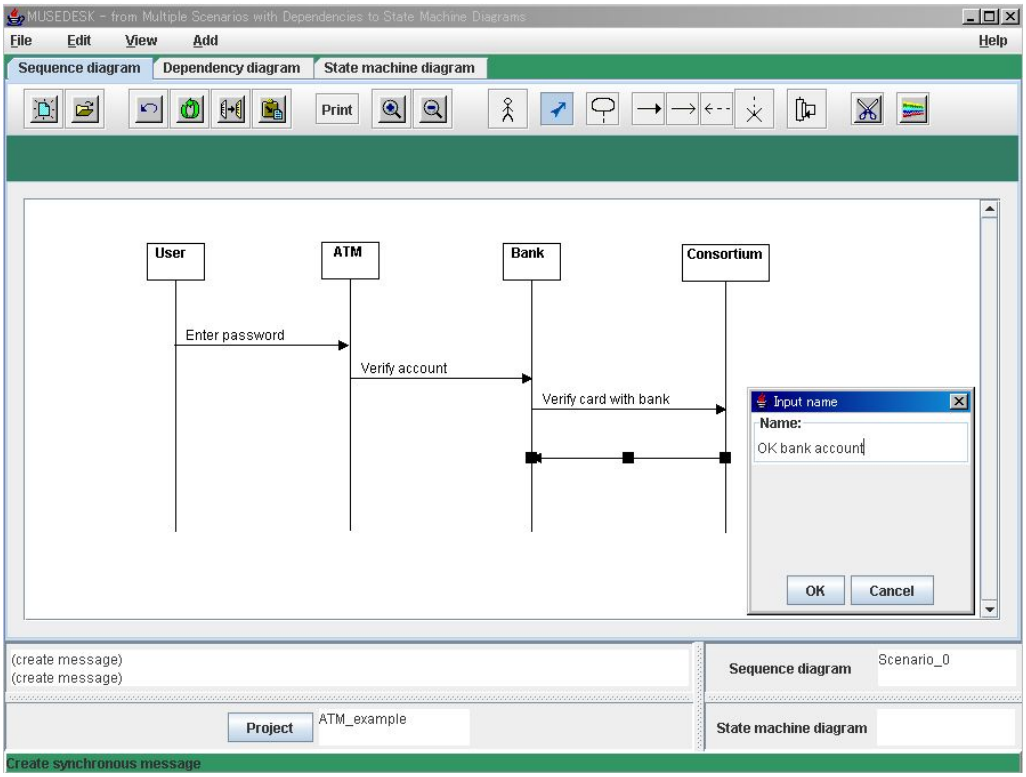


Fig. 5.2 Snapshot of MUSEDESK: sequence diagram

b) The *Dependency diagram* pane allows the creation and editing of the dependency diagrams.

There are two graphical elements here: the nodes, which are the *scenarios*, and the edges, which represent their *relationship*.

The toolbar (and the menus) also allow typical editing actions, like selecting an element, deleting an element and clearing the whole pane. When selecting an element, it can be moved or its name can be added/changed.

c) The *State machine diagram* pane is the one that displays the state machine diagrams. They can also be edited in this pane.

The graphical elements used here are *initial state*, *final state*, *state* and *transition*. The menus, here as well, allow typical editing actions, like selecting an element, deleting an element and clearing the whole pane. When selecting

an element, it can be moved or its name can be added/changed by right-clicking on it.

Fig. 5.3 shows a snapshot of this pane, where a state machine diagram for object *Bank* has been created. Setting the initial state has been done manually.

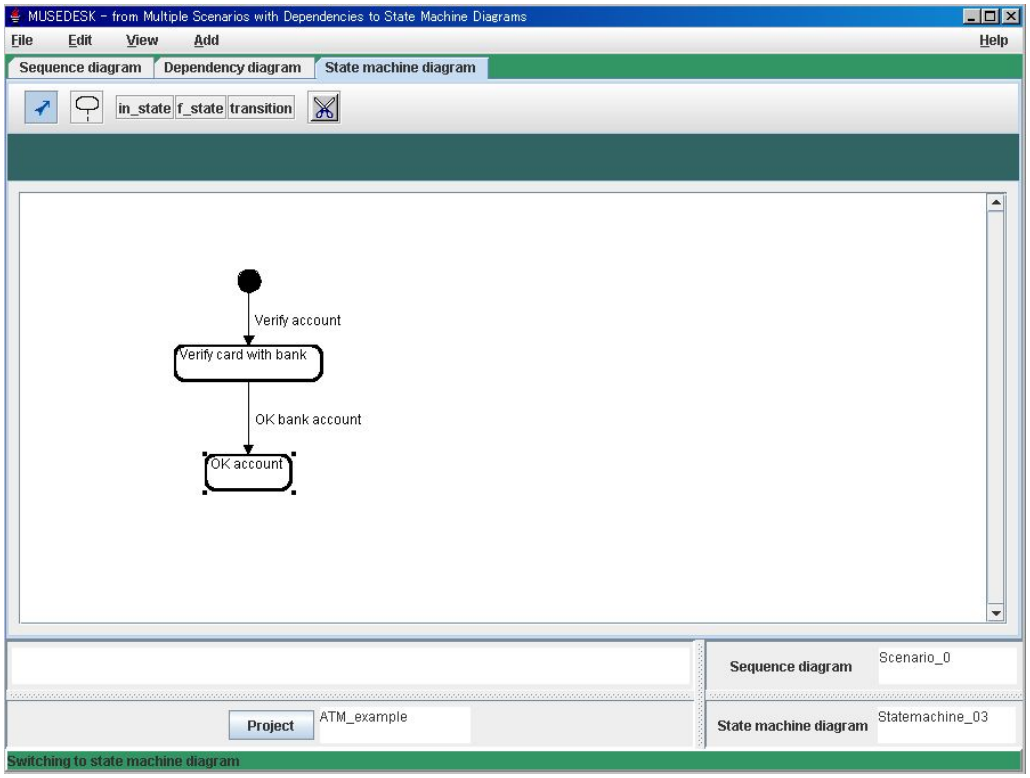


Fig. 5.3 Snapshot of MUSEDESK: state machine diagram

Along with the 3 main panes, for each type of diagram that we use, the main window contains additional tabbed panes, where the name of the project, the names of the diagrams and a description of the currently performed action appear.

In section 5.4, we are going to illustrate step by step, on a specific example, the functionality of MUSEDESK.

We should mention here that the purpose of creating our system is the illustration of our proposed method of transformation of sequence diagrams into state machine diagrams, with an emphasis on the relationships between the sequence diagrams. Our intention is not the creation of a sequence diagram and/or state machine diagram editor. Therefore, the functionality of our system is focused mainly on the method of transformation, on showing that different relationships result in different state machine diagrams, and is less concerned with the graphical elements of the diagrams, with ways of saving and retrieving various diagrams. Several formats that allow the exchange of diagrammatic information between tools exist, and in our future work we are considering using XML, for instance, which would allow the export of our generated state machine diagrams directly into other tools that could make use of them.

5.1.3. Description of the main classes

The system has been developed using Java [53], with the structure of the classes strongly relying on design patterns [51].

The hierarchy of the *MuseApplication* package appears in Fig. 5.4. (the structure is separated into two different captions because of its size; the captions are made from the Eclipse [54] hierarchy perspective). In the following, we will describe briefly the main classes that allow the functioning of the system.

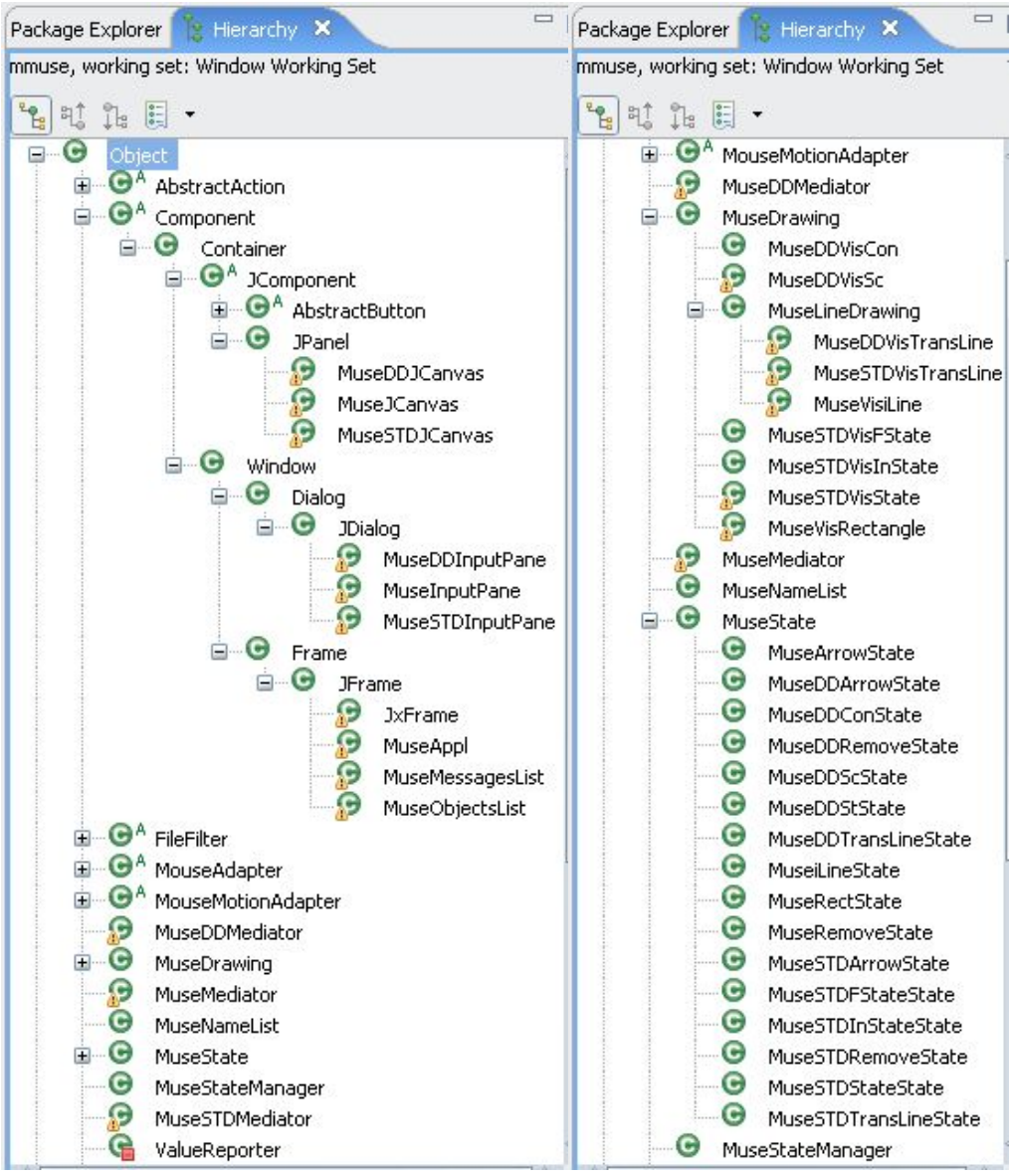


Fig. 5.4 Hierarchy for *Muse_application* package

1) MuseAppl

The main application is *MuseAppl.java*. This is the class that creates the main window, the three main panes, the tabbed panes and the menus. The application is an event based application, and therefore it responds to actions performed on its main window.

2) MuseMediator, MuseDDMediator and MuseSTDMediator

We have relied strongly on design patterns in the developing of our application. One of the most useful patterns that we have used is the mediator pattern, as described in [52], which allows a loose coupling between a large number of classes. There are 3 mediator classes, one for each type of diagram: *MuseMediator.java*, *MuseDDMediator.java* and *MuseSTDMediator.java*. Each of these mediators has an important role for each type of diagram, respectively, and each class is the only class that has detailed knowledge of the methods of the other classes that relate to the specific diagram. Mediators implement the commands, that is the pressing of the buttons, which each generate the performing of a certain operation.

3) MuseCommand and MuseXXXButton

The command pattern is used to keep the program and user interface objects completely separate from the actions they initiate [52]. Specifically, the *Command* class has an *execute* method that is called when an action occurs on that object. The *execute* method is then provided for each object that carries out the desired action; in our case, it is provided in each button class, corresponding to all the buttons we have defined: *MusePickButton.java*, *MuseRectButton.java*, *MuseRemoveButton.java*, *MuseDDScButton.java* etc. (a total number of 16 such classes, each corresponding to a certain button).

Fig. 5.5 shows how the mediator class for the state machine diagram editor connects the buttons and the interface command.

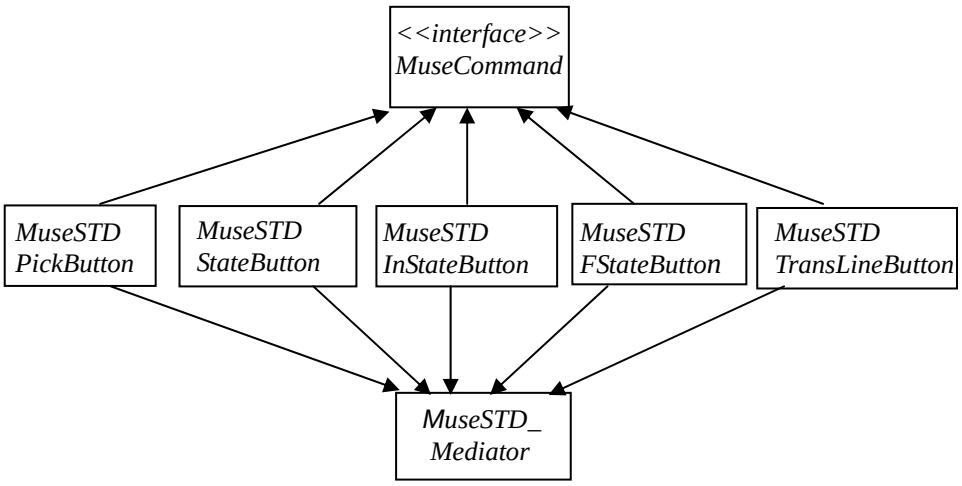


Fig. 5.5 Interaction between *MuseSTD_Mediator* and *Command*

4) MuseStateManager

Each of the buttons defined in our toolbars does something different when it is selected and the mouse is clicked; thus the state of the application affects the behavior. To illustrate this, we have found the state pattern helpful; this pattern is used to have one object represent the state of the application and to switch application states by switching objects [52].

A state manager class sets the current state and executes methods on that state object. *MuseStateManager.java* class is the one that sets our current state. This state can be one of creating a certain element, any of the ones involved in the totality of diagrams, performing an action on a specific element, like removing it, a certain button being pressed, or performing an operation on a given object, like displaying the individual state machine or the final state machine for it.

The state manager calls the methods of whichever state object is current. For instance, when a mouse is pressed, according to the current state, the corresponding method is called:

```
public void mouseDown (int x, int y)
        {
        currentState.mouseDown(x,y)
        }
```

The mediator class is the one that tells the state manager when the current program state changes. Each button click calls a specific method and changes the state. There are *startXXX* methods, called from the execute method of each button, that set the appropriate state (and turn off the other buttons), like we have in the following, for the button creating a state in the state machine diagram editor:

```
public void startState()
  {
    inStateButton.setSelected(false);
    fStateButton.setSelected(false);
    arrowButton.setSelected(false);
    tLineButton.setSelected(false);
    removeButton.setSelected(false);
  }
```

The *execute* method for the button that creates an object looks like this:

```
public void execute()
   {
      stateMgr.setState(new MuseSTDStateState(med));
      med.startState();
    }
```

The corresponding state classes show what has to be done in case of a mouse click and so on. We have state classes like: *MuseArrowState.java*, *MuseRectState.java*, *MuseRemoveState.java*, *MuseSTDStateState.java*, *MuseSTDInStateState.java* etc. Their number is equal to the number of button classes.

The interaction between the mediator and the state manager is shown in Fig. 5.6.

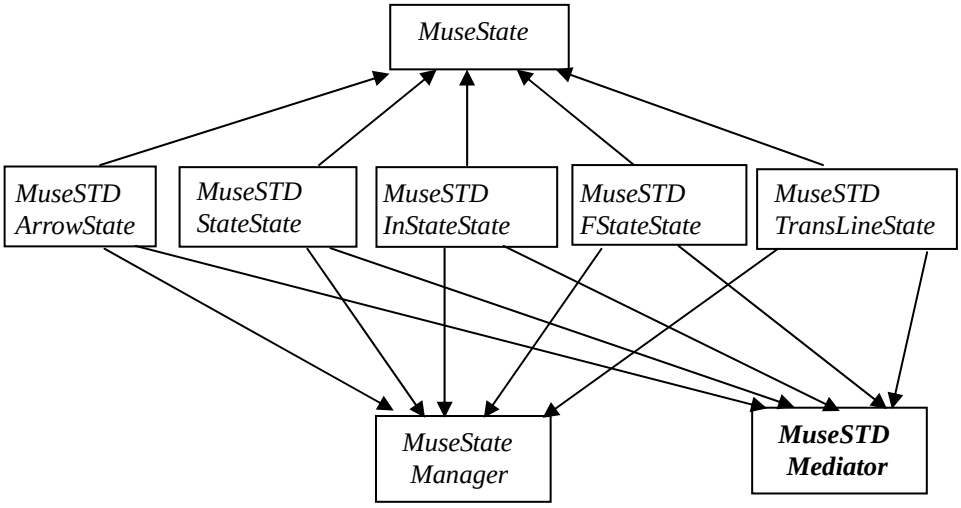


Fig. 5.6 Interaction between *MuseSTDMediator* and *MuseStateManager*

5) Classes for graphical elements

The graphical elements are created in corresponding classes, one for each such element. Among these classes we find: *MuseVisRect.java* (for an object), *MuseVisInState.java*, *MuseVisFState.java* and *MuseVisCircle.java* (for initial, final and regular states) and so on. They are all subclasses of class *MuseDrawing.java*, which has defined all the graphical elements and the operations on them. We have also defined *MuseLineDrawing.java* as a subclass of *MuseDrawing.java*, responsible for drawing line-type graphical elements (like the messages in the sequence diagram, the relationship edges in the dependency diagram and the transitions in the state machine diagrams). This class has *MuseVisiLine*, *MuseDDVisTransLine* and *MuseSTDVisTransLine.java* as its subclasses (they create a message, a relationship edge and a transition respectively).

The mediator classes are the classes which contain most of the methods responsible for the transformation of sequence diagram elements into state machine diagram elements. The method *setElement* is the one in charge with remembering the type of diagram elements that have been created and producing the list of these elements. Depending on the type of drawing element, this method remembers the element type (object or message, in the case of sequence diagrams, scenario or relationship in the case of dependency diagrams, state or transition in the case of state machine diagrams) and adds it to the corresponding list (the lists are vectors like *objectlist*, *messagelist*, *scenariolist* etc.) .

```
public void setElement(MuseDrawing d)
  {

     if(d instanceof MuseVisRectangle)
     {
     …
     objectlist.addElement(((MuseVisRectangle)d).name)
     }

     if(d instance of MuseVisiLine)
     {
     …
     messagelist.addElement(((MuseVisiLine)d).name)
     }
      …
  }
```

5.2. Automated Teller Machine example

5.2.1 Description of ATM system

We are going to illustrate the process of transformation of sequence diagrams into state machine diagrams using our system on an example, based on the one used by Rumbaugh et. al in [31], that is an *ATM system*. Let us consider a banking network including automated teller machines to be shared by a consortium of banks. ATMs communicate with a central computer which clears transactions with the appropriate bank. An 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) and so on.

5.2.2 The process of transformation in MUSEDESK

Phase I

In the first phase, the sequence diagrams corresponding to the given scenarios are created.

In our ATM example, let us assume that the user is first presented with the main screen, then (s)he inserts the card and the password, and, after the validation with the bank and the consortium of banks, (s)he can choose among the three possible transactions: withdrawal, deposit or transfer. We therefore suppose that we have the following three possible initial scenarios: one which deals with withdrawal of cash, one which allows depositing cash and a third one which deals with transferring a certain amount of money into a different account. These initial scenarios are *Scenario_withdraw_initial* (Sc_{10}), *Scenario_deposit_initial* (Sc_{20}), *Scenario_transfer_initial* (Sc_{30}).

The three sequence diagrams that represent the three scenarios are created in the sequence diagram pane; this is done by choosing various graphical elements, like objects and messages; their names are inputted by

right-clicking on the corresponding element. A list of objects and a list of messages are produced for each sequence diagram. After finishing the editing, the name for the created sequence diagram is inputted. Fig. 5.7 shows a caption of the sequence diagram pane, where *Scenario_withdraw_initial* has been created. Fig. 5.8 and Fig. 5.9 show the other two scenarios, *Scenario_deposit_initial*, *Scenario_transfer_initial*.

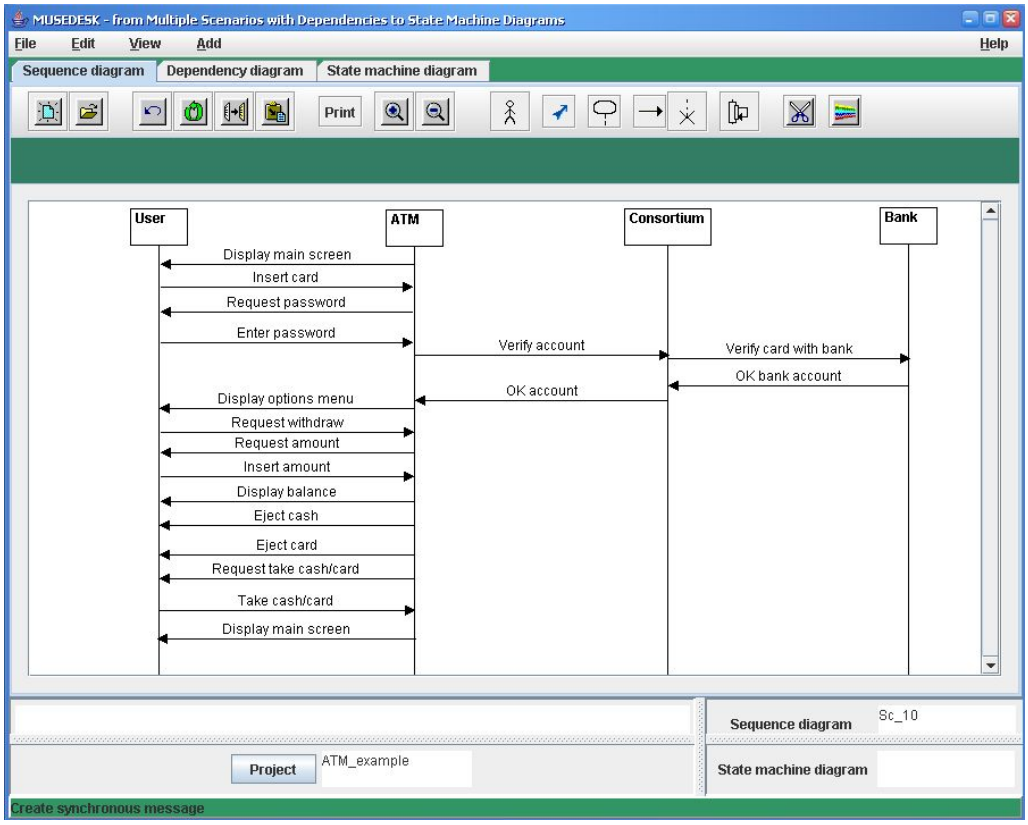


Fig. 5.7 Scenario *Scenario_withdraw_initial* for an ATM

The system saves the names of the objects created in a vector of names, "*objectlist*". The messages are also remembered; the list of messages for this partially created scenario, Sc_{10}, maintained by the mediator class, is the following:

(M_10, Display main screen), (M_01, Insert card), (M_10, Request password), (M_01, Enter password), (M_12, Verify account), (M_23 Verify card with bank), (M_32, OK bank account), (M_21, OK account), (M_10, Display options menu), (M_01, Request withdraw)

(Similar lists are created for the other two scenarios.)

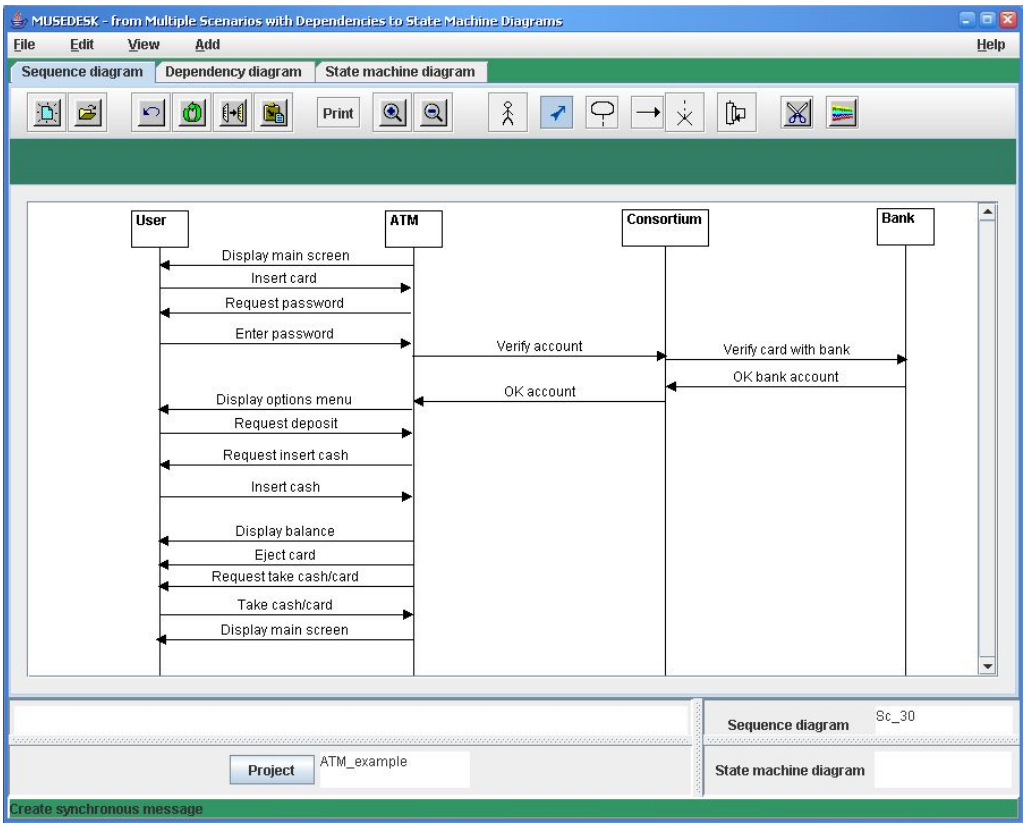


Fig. 5.8 Scenario *Scenario_deposit_initial* for an ATM

After the representation of the initial sequence diagrams, we need to perform their normalization (involving the removal of overlapping). The system checks the 3 lists it created and it looks for a number of N or more consecutive common elements. (We have previously decided, as shown in section 3.3, that N = 5 is the minimum number of common messages which we will consider for the creation of a new sequence diagram.)

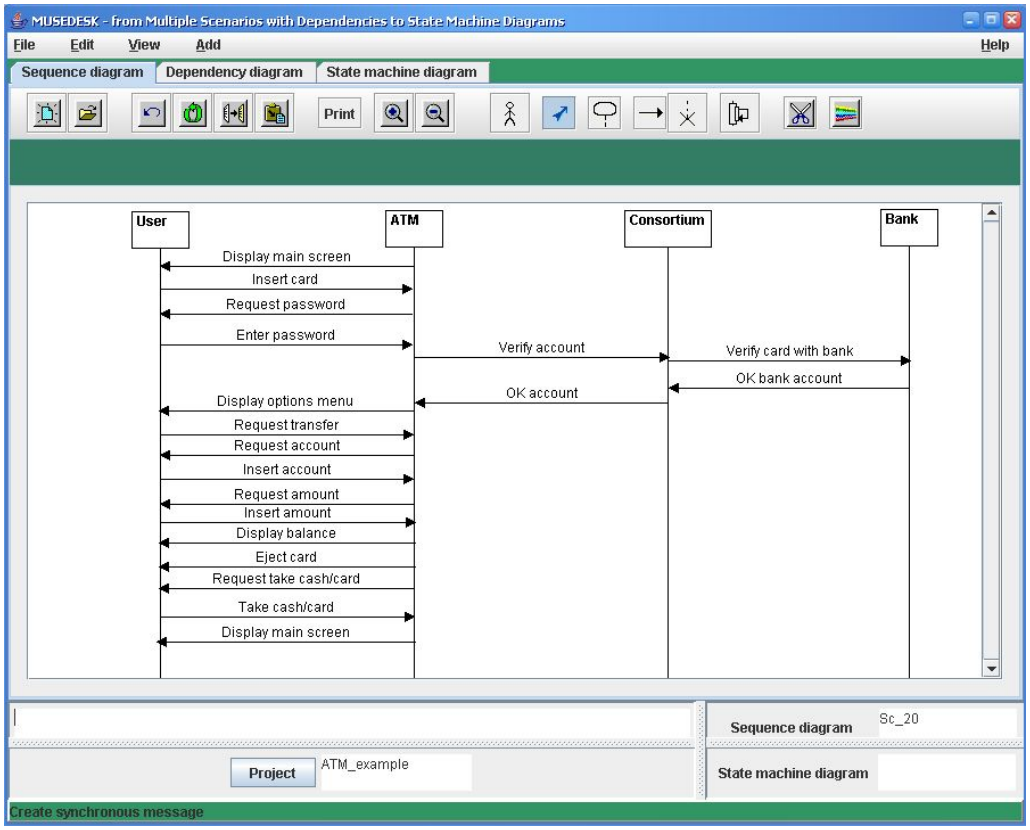


Fig. 5.9 Scenario *Scenario_transfer_initial* for an ATM

The system finds 9 common consecutive messages; they correspond to the part where the user inserts the card and this card is authenticated with the bank. The messages that are found as being common to the three given scenarios are:

(M_10, Display main screen), (M_01, Insert card), (M_10, Request password), (M_01, Enter password), (M_12, Verify account), (M_23 Verify card with bank), (M_32, OK bank account), (M_21, OK account), (M_10, Display options menu)

These messages will form a new scenario, which we will call *Scenario_start*, and which appears in Fig. 5.10. After removing the overlapping, the other three newly obtained scenarios are *Scenario_withdraw*, *Scenario_deposit*, *Scenario_transfer*.

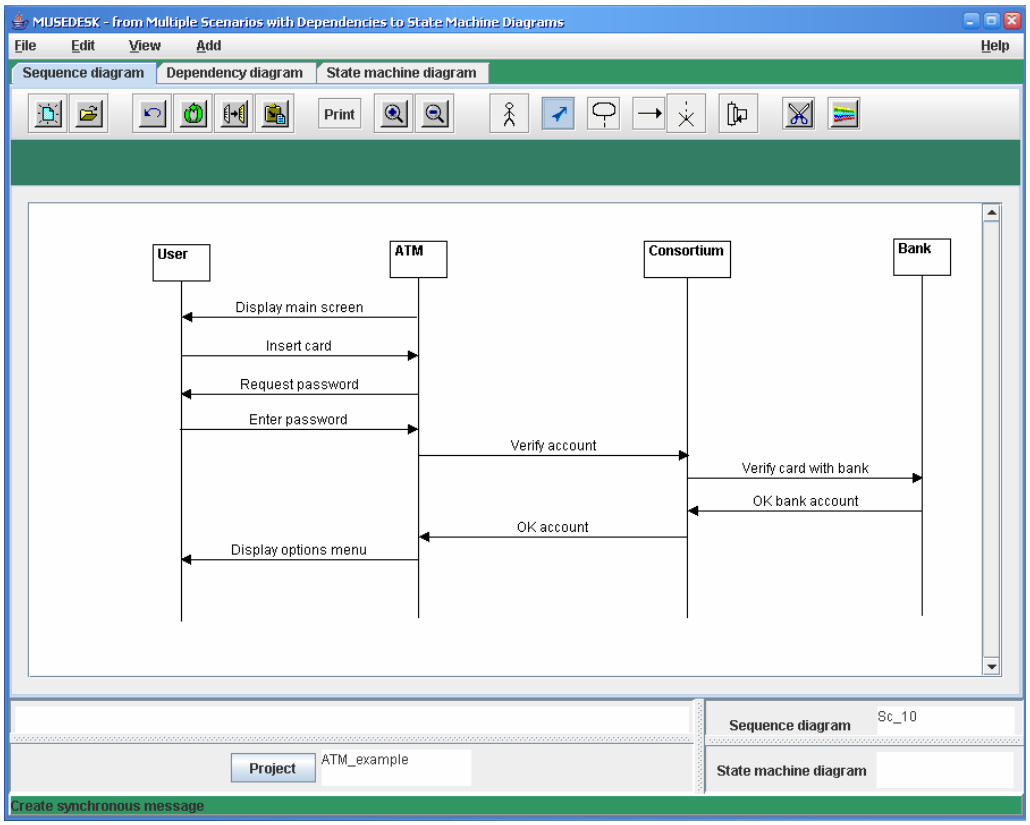


Fig. 5.10 Scenario *Scenario_start* for an ATM

The system separates the common messages and those that are not common into different lists. However, it does not automatically draw the new sequence diagram on the screen. In the current version, this is done manually by the user.

As we have mentioned in section 5.1, the purpose of our actual system is to show the importance of dependencies between various scenarios in the creation of state machine diagrams. Our system focuses on this feature, but it does not have the capabilities of a diagram editor. Consequently, in its current version, the information contained in the created diagrams is remembered only in terms of semantic elements, like messages, states, transitions etc., not graphical information. Therefore the diagrams cannot be recreated graphically in their corresponding pane, once they have been erased.

Result of phase I: 4 normalized sequence diagrams.

Phase II

After the normalization of the sequence diagrams, we will represent the relationships existing between them. These relationships are reflected in the dependency diagram in Fig. 5.11. We cannot find any “sub-part” type of relationship in our example, and therefore the normalized sequence diagrams obtained in phase I will remain unchanged.

The dependency formula that represents the dependency diagram has the form:

Scenario_start ; (Scenario_withdraw ∨ Scenario_deposit ∨ Scenario_transfer)

Result of phase II: (the same) 4 normalized sequence diagrams, along with one dependency diagram.

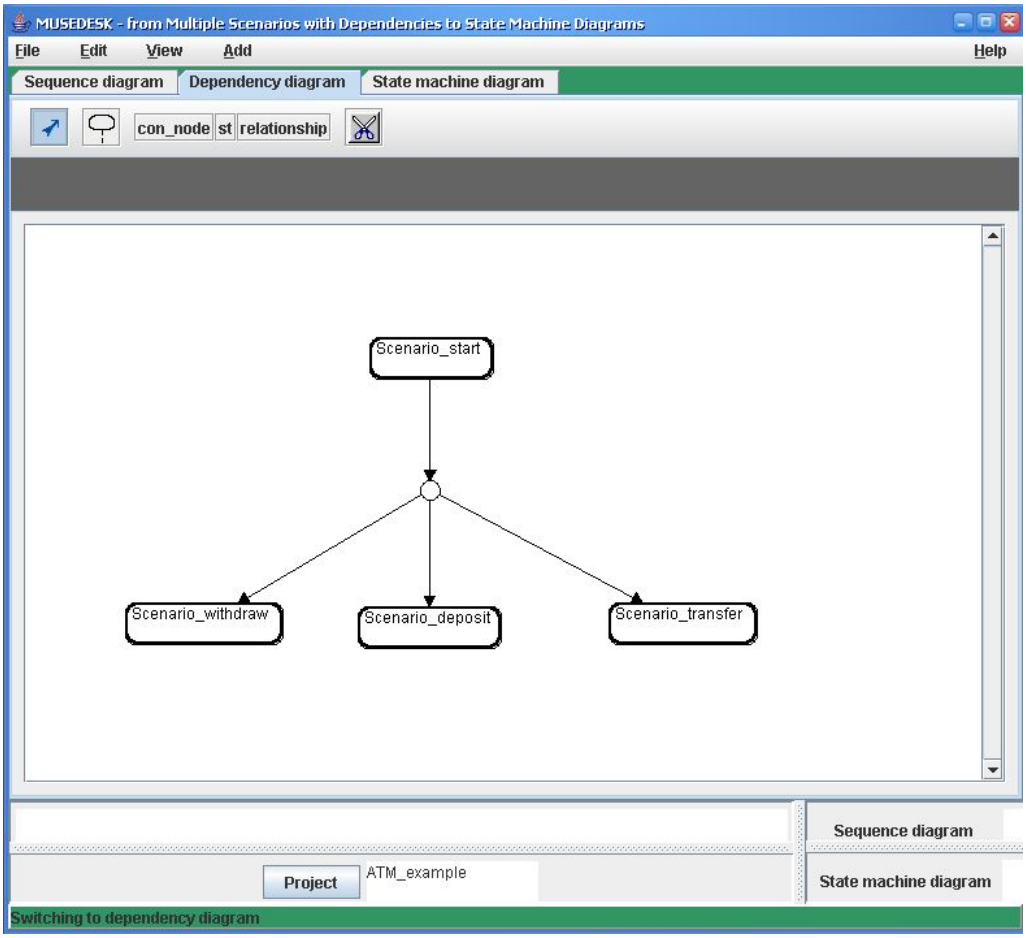


Fig. 5.11 Dependency diagram for the ATM example

Phase III

In this phase we focus on a certain object and create its state machine diagram, including its complete behavior, as reflected in the given scenarios. The system has already created a list including all the objects appearing in the sequence diagram ("*objectlist*"). This list can be displayed on the main screen and the desired object can be selected. In the snapshot in Fig. 5.12, we can see the list containing the 4 objects created ("*List of created objects*"). Currently, object *ATM* is selected.

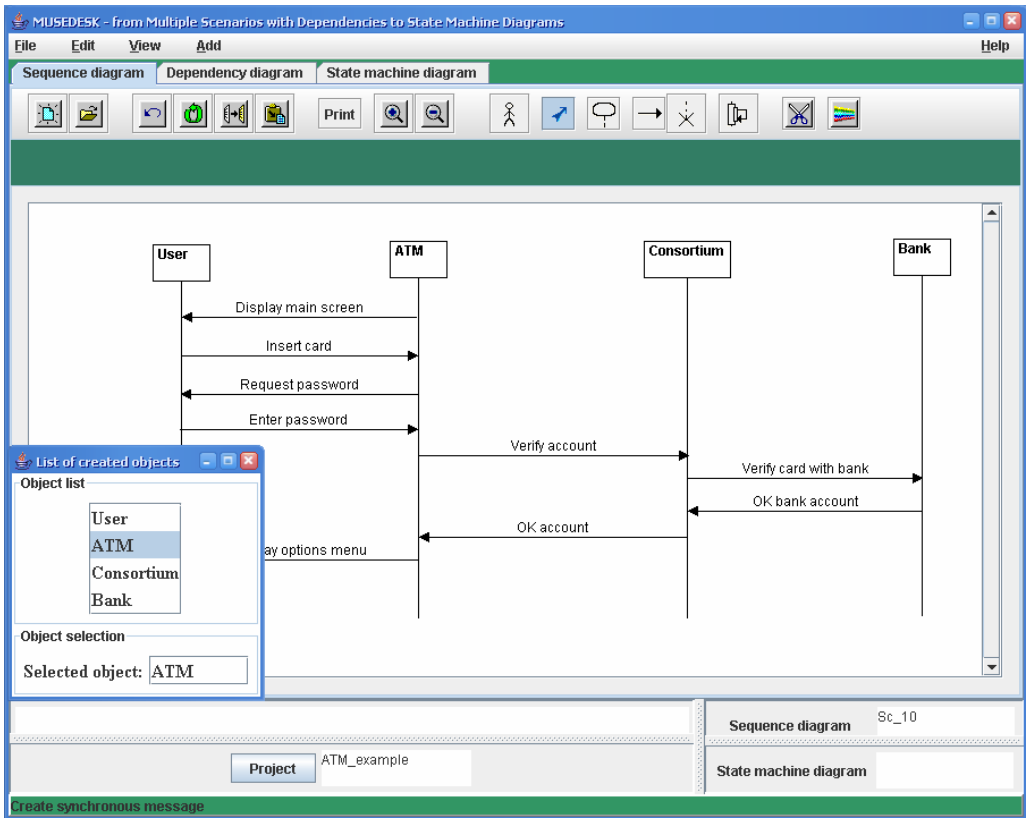


Fig. 5.12 Selecting the desired object from the list of created objects

In our example, let us focus on object *ATM* and create the state machine diagram for this object.

Phase III.1

First, we will find the sequence diagrams in which our object appears. This is done by searching if the object name *ATM* appears in the list of messages of the sequence diagrams. The result is that it appears in all 4 scenarios; this means that we will have 4 initial state machines for object *ATM*.

As described in the algorithm in section 4.3, for each sequence diagram where the specified object appears, we will search the message tuples and do the following:

- if the object is the originator of the message, we will create a state with the same name;
- if the object is the receiver of the message, we will create a transition with the same name.

This results into two different lists, one for states, one for transitions. Again, the mediator class is the one where these lists are created.

The following represent the lists of states and transitions corresponding to the first scenario where object *ATM* appears, that is *Scenario_ start*:

- list of states: $S_{ATM}LS_1$ = *{(state, Display main screen, 1), (state, Request password, 3), (state, Verify account, 5), (state, Display options menu, 7)}*
- list of transitions: $S_{ATM}LT_1$ = *{(trans, Insert card, 2), (trans, Enter password, 4), (trans, OK account, 6)}*

After checking the order of the elements in the list, we create a combined list of states and transitions; our list for the first sequence diagram is the following:

$S_{ATM}M_1$ = *{(state, Display main screen, 1), (trans, Insert card, 2), (state, Request password, 3), (trans, Enter password, 4), (state, Verify account, 5), (trans, OK account, 6), (state, Display options menu, 7)}*.

For our system, this means that we have all the semantic information necessary to create an initial state machine diagram. However, if we are interested in the overall behavior of object *ATM*, resulting from all scenarios, we actually need the complete information showing the behavior of the object

from all scenarios. We could choose to represent this initial state machine diagram graphically, but it would not serve our ultimate purpose.

Similarly, we will obtain lists of states and transitions for the other 3 initial state machines for object ATM:

- combined list of states and transitions obtained from *Scenario_withdraw*:
$S_{ATM}M_2$ = *{(trans, Request withdraw, 1), (state, Request amount, 2), (trans, Insert amount, 3), (state, Display balance, 4), (state, Eject cash, 5), (state, Eject card, 6), (state, Request take card/cash, 7), (trans, Take cash/card, 8), (state, Display main screen, 9)}*

- combined list of states and transitions obtained from *Scenario_deposit*:
$S_{ATM}M_3$ = *{(trans, Request deposit, 1), (state, Request insert cash, 2), (trans, Insert cash, 3), (state, Display balance, 4), (state, Eject card, 5), (state, Request take card/cash, 6), (trans, Take cash/card, 7), (state, Display main screen, 8)}*

- combined list of states and transitions obtained from *Scenario_ transfer*:
$S_{ATM}M_4$ = *{(trans, Request transfer, 1), (state, Request account, 2), (trans, Insert account, 3), (state, Request amount, 4), (trans, Insert amount, 5), (state, Display balance, 6), (state, Eject card, 7), (state, Request take card/cash, 8), (trans, Take cash/card, 9), (state, Display main screen, 10)}*

Result of phase III.1: 4 initial state machine diagrams (for object ATM).

Phase III.2

According to the information in the dependency diagram in Fig. 5.11, the state machine resulting from *Scenario_initial* precedes the other state machines. The rest of the 3 state machines are combined with OR. (A single default state will exist.) This allows us to obtain the meta-list of states and transitions, following the dependency formula written in the previous phase. Our meta-list for object *ATM* will look like in the following:

{((state, Display main screen), (trans, Insert card), (state, Request password), (trans, Enter password), (state, Verify account), (trans, OK account), (state, Display options menu)) ; (((trans, Request withdraw), (state, Request amount), (trans, Insert amount), (state, Display balance), (state, Eject cash), (state, Eject card), (state, Request take card/cash), (trans, Take cash/card), (state, Display main screen)) ∨ ((trans, Request deposit), (state, Request insert cash), (trans, Insert cash), (state, Display balance), (state, Eject card), (state, Request take card/cash), (trans, Take cash/card), (state, Display main screen)) ∨ ((trans, Request transfer), (state, Request account), (trans, Insert account), (state, Request amount), (trans, Insert amount), (state, Display balance), (state, Eject card), (state, Request take card/cash), (trans, Take cash/card), (state, Display main screen))))}

Again, this list contains all the information regarding the states and transitions in the state machine diagram for object *ATM*. The graphical information is not contained here and this is why the graphical visualization of the diagram cannot be obtained automatically. If we want to construct this diagram using our system, we use the list of states and transitions provided in the meta-list; they are ordered and they contain all the necessary semantic information. (The only exception is the setting of the default state, which is not automatically generated in our approach.)

We can use the state machine diagram editor to graphically create the resulting state machine diagram; part of this diagram is visible in Fig. 5.13.

Result of phase III.2: the final state machine diagram for object ATM.

In our example we chose to generate the state machine diagram for one object only, *ATM*. If we were to generate the state machine diagrams for the other objects, the result of phase III.2 would be a number of 4 final state machine diagrams, one for each existing object. The state machine diagram for object *User*, which appears in 4 sequence diagrams, can be obtained in a

similar manner. In the case of objects *Bank* and *Consortium*, they appear in one scenario only; we would therefore have one initial state machine diagram for each, and this would be the same as the final state machine diagram.

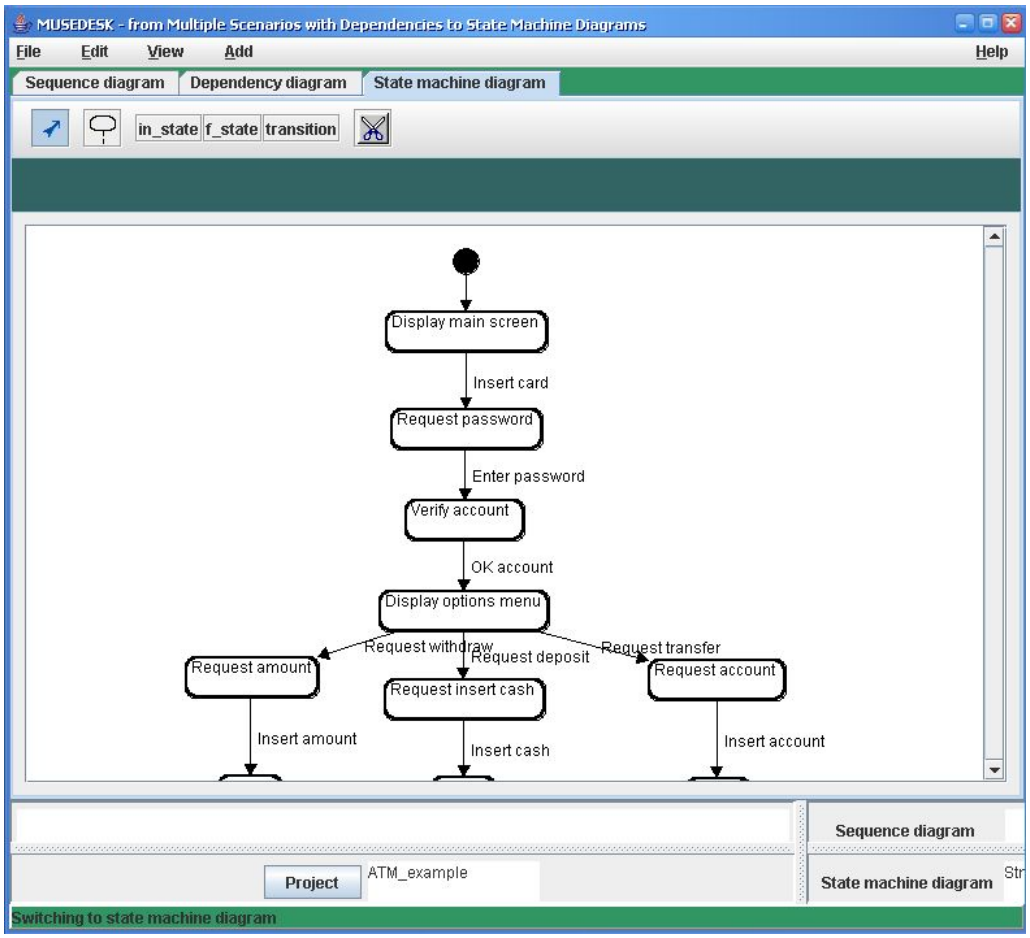


Fig. 5.13 Final state machine diagram for object ATM

5.2.3 Other features and limitations

- **Guards**

We should note that in our example's sequence diagrams we did not have any messages with guards attached to them. As we have shown in section 4.3, in case guards are involved, these guards are going to appear together with the event; thus they will appear only for the receiving object. In case there is a guard attached to an outgoing message from an object, it will be considered when creating the state machine for the object that receives this message.

- **Type of message**

We considered all our messages to be synchronous by default and therefore we have not included information related to their type at all. As we have shown in section 3.2, the message tuple includes, as a third element, the type of message; in our case it was considered "1" by default, for all messages (1 is the value for synchronous message).

- **Lost and found messages**

These features can be included in the sequence diagram, as graphical elements. However, their influence on the obtained state machine diagram is not significant from our point of view and we have therefore not included them in our example. In theory, for a lost message, according to its definition, we will have only a sending object. The state machine diagram for this object will therefore include a state with the same name (there will not exist any object with a transition resulting from this message). For a found message, we will have only the receiving object, and therefore a transition for this object will exist (but no object having a state with the same name as the found message).

- **"Correctness" of sequence diagrams**

One important issue that we need to emphasize is that our approach is not concerned with the elicitation of requirements itself. Therefore, in our system, the creation of sequence diagrams practically suggests that this is equivalent to writing the requirements. This should be done on the basis of certain assumed scenarios (given, for instance, in natural language). For the purpose of proving

our point, which is the transformation of scenarios (sequence diagrams) into state machines (with an emphasis on relationships between scenarios), we have to consider that the user knows exactly the interaction between various objects in the sequence diagram and thus the creation of sequence diagrams is only a matter of graphical editing in our system. The process of identifying exactly the objects and the messages exchanged between them, on the basis of given scenarios, is a different matter, which is outside our current concern boundaries. It is arguable that the creation of sequence diagrams from requirements given as scenarios written in natural language (or any other form) is a straightforward operation. In the same way, ensuring that requirements are correct or, at least, that the created sequence diagrams reflect the given requirements, is a matter which constitutes in itself an entire area of research. Our intention is to show how the state machine diagrams depend on these sequence diagrams and on the relationships between them, and, in order to do this, we assume that the sequence diagrams are "correct", i.e. they reflect the requirements of the system.

- **Scalability and maintainability of our system**

Our system can be easily applied to large systems, where the requirements are very extensive, as long as they can be expressed in scenarios represented as sequence diagrams. Actually, the experiments we have performed showed that our system behaves well when the input is a large number of sequence diagrams, with a large number of messages.

As for maintainability, our system is small enough not to pose special problems when it comes to its maintainability. We believe that if new issues arise, new requirements for it, for its improved performance, our system can be easily modified to be able to fulfil these new requirements.

Chapter 6

MUSEDESK System Evaluation

6.1. Experiment description and results

The final purpose of our proposed approach is the obtaining of state machine diagrams. We are going to compare the process of obtaining these diagrams manually with that of obtaining the diagrams by using our system MUSEDESK. For this, we have conducted an experiment in which a number of 6 participants were given simple requirements for a software application, given in natural language (as scenarios are often written) and they were asked to obtain the state machine diagram for a certain object in two ways: first, manually (using pencil and paper) and then automatically, using our system.

All 6 users were males between the ages of 25 and 36 and all had a computer science background. Half of the users considered their knowledge about UML concepts as average, while the other half considered their UML knowledge below average.

The average time per participant needed for performing one complete set of tasks (including training and general explanations) was around 30 minutes. We have performed all the measurements on a 1.3GHz computer, with a Celeron M microprocessor and 512Mb of RAM. Our system was developed in Java 1.5 in a Windows XP environment.

The requirements specifications and the required final task appear in Fig. 6.1.

Requirements specifications

Let us consider that we have a CD Player system, which contains a remote controller (object *Ctrl*) and the CD device itself (object *Device*).
The remote controller has 3 buttons: one for the speakers (*speakerbt*), one which makes the device play (*playbt*) and one which stops the device (*stopbt*).

We assume the following two scenarios:

Scenario 1
1.1 The user presses the "*speakerbt*" button, which activates the left speaker of the device.
1.2 Then, the user presses the "*speakerbt*" button again, which activates the right speaker of the device.
1.3 After the user presses the "*speakerbt*" button for the third time, both speakers of the device are activated.

Scenario 2
2.1 The user presses the "*playbt*" button and the device starts playing.
2.2 The user presses the "*stopbt*" button and the device stops playing.

Consider that scenario 1 takes place first, and it is followed by scenario 2.

TASK:
Create the state machine diagram for the remote controller object (*Ctrl*).

Fig. 6.1 Requirements specifications for a CD Player system

We are going to call the two terms of our comparison "*manual process*" and "*MUSEDESK process*" and we are going to describe them in the following.

Case I: Manual process

In this case, the users perform the final task directly from the given requirements, so they have to:

- *create the state machine diagram for the given object.*

The users were given the requirements in Fig. 6.1. After studying them carefully, they created the state machine diagram. The time necessary to create the state machine diagrams for each participant appears in Table 6.1. (We have considered this as "Task A".)

Table 6.1 Time needed to create the state machine diagram for object *Ctrl* in the manual process, directly from the given requirements specification

Participant / Task	1	2	3	4	5	6	average
Task A	3`29``	4`18``	3`32``	4`23``	3`51``	4`07``	3`58``

We have noticed the following general tendencies when creating the state machine diagrams manually, directly from the given requirements:

a) introduction of additional, unnecessary states

This was the case with regard to the states arising from pressing the buttons "*Play*" and "*Stop*". Two of the users introduced the following states: "*Play*", "*devicePlay*", as well as "*Stop*" and "*deviceStop*". In each pair, one of the states is redundant. (At the end of the experiment, when asked to explain their use, the users admitted that they were not in fact necessary.)

b) missing transitions

This was the case of "*speakerbt*"; although 3 transitions with this name result from scenario 1, the participants were tempted to write its name only once, afterwards ignoring it.

Also, "*speakerbt*" was missing as a first transition in the state machine diagram, in the case of one user.

c) "over-interpretation" of the requirements

The users included in the state machine diagram behavior which was not expressed in the requirements. For example, although it might seem natural to consider that after reaching state "*Stop*", the device will be again in the state it started from, where "*speakerbt*" is pressed, this kind of behavior is not part of the requirements. Therefore, the resulting state machine diagram does not reflect the given requirements, but the subjective addition of requirements resulting from the user's personal opinion.

d) separation of state machine diagrams into several parts

Instead of obtaining one single state machine diagram for the required object, one of the participants has created 3 diagrams, one corresponding to each function of the device being modeled. Interestingly, the user did not create one diagram for each scenario, but one diagram for each function of the system (one for the "*speakerbt*" related behavior, one for "*playbt*" and one for "*stopbt*").

e) necessity to familiarize with state machine diagram concepts

For the participants less familiar with UML concepts, additional time was necessary to familiarize with the significance of states and transitions in state machine diagrams. This operation was actually the most time consuming. (For this half of the participants, not familiar with UML, the average time needed to understand simple states and transitions was around 6 minutes.)

Case II: MUSEDESK process

When using MUSEDESK, the user has to perform the following activity only:
- *create the sequence diagrams corresponding to the given scenarios from the requirements specifications (and specify their relationship).*
After choosing the desired object, the system obtains the information necessary for creating state machine diagrams *automatically*, without the intervention of the user.

The participants studied the given requirements and then used our system to create the sequence diagrams. Editing the diagrams was straightforward, although in the beginning the users took longer time in finding the appropriate buttons. (We have considered this as "Task B"). The time needed to perform this operation for the 6 participants appears in Table 6.2.

As for the time needed for MUSEDESK to perform the automatic transformation into state machine diagrams, according to our measurements, its value is a few tens of milliseconds (even for large number of messages exchanged between various objects in sequence diagrams, like 1000 messages, the values remain in the same range, of tens of milliseconds).

Table 6.2 Times (in minutes and seconds) for completing task B for the 6 participants

Participant / Task	1	2	3	4	5	6	average
Task B	4`31``	4`40``	3`49``	5`02``	2`55``	4`18``	4`12``

The result of task B looks like in Fig. 6.2. This is a snapshot of the sequence diagrams created by one user; the other 5 look very similar.

The final result, the state machine diagram itself, as it can be created based on the automatically created meta-list of states and transitions, looks like in Fig. 6.3.

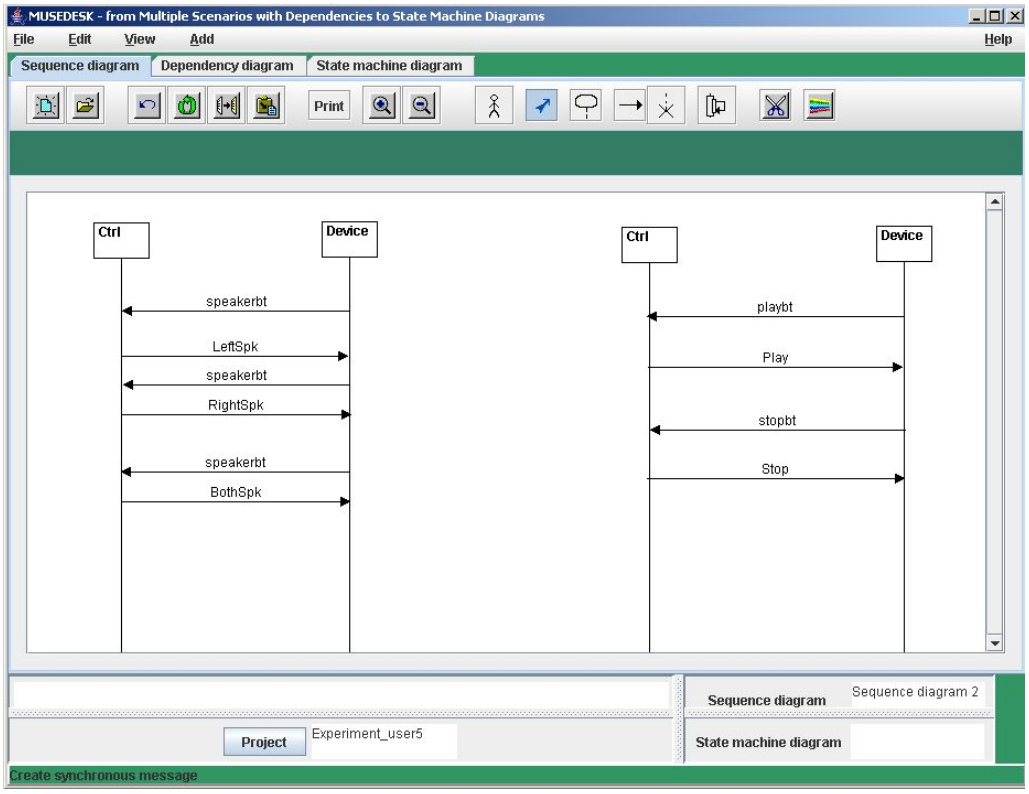


Fig. 6.2 Result of task B for one participant

As compared to the manual process, the MUSEDESK process, being an automatic one, cannot result in missing transitions, addition of extra states or unintended specifications. The process of creating the state machine diagrams, as described in chapter 4, states clearly how each transition and each state is born. This being an automatic process, it does not introduce errors like the ones appearing in the manual process.

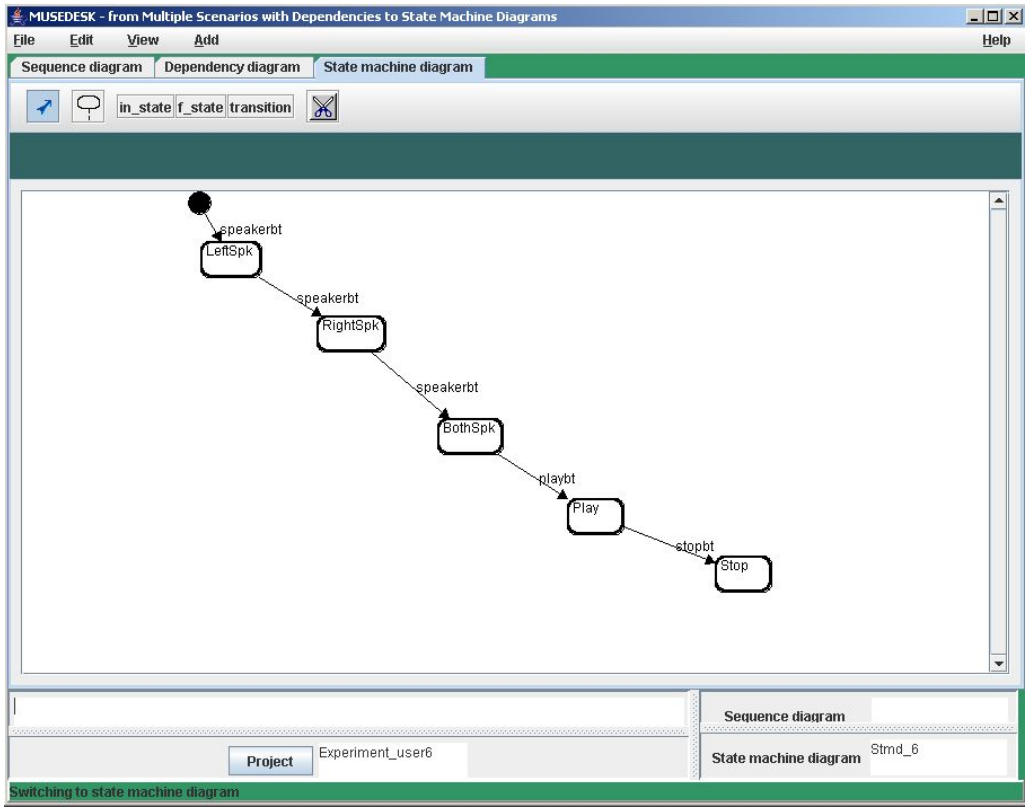


Fig. 6.3 Result of using MUSEDESK in the creation of the state machine diagram

Time comparison

Let us now compare the time needed to achieve our goal (of creating state machine diagrams) in the two given cases.

I. In case I, corresponding to a manual process, directly from the requirements specifications, the time is:

T1 = time_TaskA

II. In case II, when using MUSEDESK, the time is:

$$T2 = time\_taskB + theta \qquad \text{(theta is in the range of tens of ms)}$$

Since time_taskB represents values in the range of several minutes, as compared to *theta* which is in the order of milliseconds, we can approximate the total time T2 needed by MUSEDESK to create a state machine diagram as being:

$$\boldsymbol{T2 = time\_TaskB}$$

Table 6.3 summarizes the values for the time needed to perform the whole process in the two cases.

Now that we have obtained the amounts of time needed for each situation, let us compare them.

Table 6.3 Time values (in minutes and seconds) for T1 and T2

Participant / Time values	**1**	**2**	**3**	**4**	**5**	**6**	**average**
T1	3`29``	4`18``	3`32``	4`23``	3`51``	4`07``	3`58``
T2	4`31``	4`40``	3`49``	5`02``	2`55``	4`18``	4`12``

a) We can notice from Table 6.3 that T2 is larger than T1 for 5 out of the 6 participants; for these users, the time needed for creating state machine diagrams is slightly longer when using MUSEDESK as compared to the manual process without creation of sequence diagrams.

For one participant, T2 is smaller than T1 (with 25%).

On average, the time needed by using our system is about 5% longer than the one needed for the manual process.

b) The lists created automatically by MUSEDESK contain additional information related to the position of the element in the state machine diagram, allowing the system to automatically generate the elements of the diagram in the right order.

These lists of states and transitions (equivalent to the requirements in task B), using the automatic generation mechanism offered by MUSEDESK, for object "*Ctrl*", look like in the following:

- "sequence diagram 1:
- list of states: *{(state, LeftSpk, 2), (state, RightSpk, 4), (state, Bothpk, 6)}*
- list of transitions: *{(trans, speakerbt, 1), (trans, speakerbt, 3), (trans, speakerbt, 5)}*

- sequence diagram 2:
- list of states: *{(state, Play, 2), (state, Stop, 4)}*
- list of transitions: *{(trans, playbt, 1), (trans, stopbt, 3)}*

However, when creating the lists manually, the users define the positions of various elements mentally, by looking at the position in the scenarios (or sequence diagrams), without the need to write them down.

This works well in case of simple scenarios. As requirements become more and more complex, in the manual approach, the user will find it increasingly difficult to remember the position of states and transition and therefore he will have to write this down, as well. The time taken for task B would increase and this would lead to an increase in the time taken to achieve the goal. The time taken by MUSEDESK remains practically the same, thus, for larger and more complex systems, the time needed by MUSEDESK to achieve the goal can decrease even more, compared to the manual approach.

Observation

A general observation refers to the fact that familiarity with UML concepts plays an important role in achieving the goal. Half of the users were acquainted with UML diagrams, while the other half had knowledge of the diagrams mostly by name. For instance, the users familiar with UML realized that an initial state might be needed in a state machine diagram. More significantly, for users not familiar with UML, the time needed to understand the concepts related to state machine diagrams (and sequence diagrams, if needed) was about twice larger than the time to create the state machine diagrams themselves. This slowed down even more the manual process of achieving the final goal.

When using MUSEDESK, the users need to understand the requirements well, but they do not necessarily have to be very knowledgeable about state machine diagram concepts, about the flow of states and transitions. The diagrams are obtained automatically by the system and this can be very helpful.

Elicited comments from the participants in the experiment

We have gathered several comments and suggestions from the participants in the experiment. They are summarized in the following:

- The system is helpful in obtaining the information necessary to create state machine diagrams for the desired objects.
- Editing of sequence diagrams could be improved, made more ”user-friendly” (the only specifics given here were related to the way of selecting objects).
- The messages in the sequence diagram should be tightly linked to the objects they involve.
- Performing of illegal editing operations should not be allowed; corrections should be suggested for them.
- The state machine diagrams and the dependency diagrams are easier to input than the sequence diagrams.
- Selection of graphical elements in all diagrams should be possible any time

(not only when in selection mode).

- The icons and the buttons in the sequence diagram toolbar should be more suggestive.
- It would be helpful to the user to see what the system realizes in the background (for instance, to have easier access to the resulted lists of states and transitions).

In conclusion, the time taken to obtain state machine diagrams is slightly longer when using MUSEDESK compared to the time needed for a manual creation of these diagrams from given requirements. However, this process can be speeded up using our system in case of larger sequence diagrams, that is, in case of larger scenarios.

The more important contribution resides in the fact that the automatic process using MUSEDESK is less prone to errors. As the observations from “case I” have shown, in the manual process several errors can make place in the state machine diagram. When using MUSEDESK, there are no missing transitions or states, no over-generalizations of the requirements and no unnecessary states.

6.2. Comparison with SCED

One of the closest related approaches to our approach is represented by SCED [9] and therefore we are going to make a comparison between some of the results using our approach and those using SCED.

The SCED project has been carried out at the University of Tampere, Finland, by Kai Koskimies and his team. The central idea of SCED is to support a design-by-example approach for object-oriented analysis and design. The OMT method has been used as a guideline and notational basis, but in principle the approach used in SCED is not tied to any particular design methodology. SCED consists of two conventional CASE components, a scenario editor and a state diagram editor, and a more intelligent component, called a generator, integrating scenarios and state machines with various mechanisms. The generator can be asked to synthesize a state diagram for a selected object or an operation appearing in a desired set of scenarios. It can also be asked to check consistencies between scenarios and state diagrams, to construct scenario diagrams with the support of existing state machines and to generate various layout choices for state diagrams.

SCED does not take into consideration the relationships between scenarios; it often gives a number of state machines equal to the number of scenarios where the object appears; it does not necessarily obtain, for one object, one state machine diagram. Our approach, on the other hand, creates a unified state machine diagram that considers the relationships between scenarios. It is therefore difficult to make a quantitative comparison between the two approaches. However, we are going to make a comparison showing the differences between the results in the two approaches.

In order to illustrate our comparison, let us make use of part of the ATM example appearing in the previous chapter. Let us start from two scenarios: an authentication scenario (where the user inserts the card and this is verified with the bank) and a scenario of withdrawing cash. They appear in Fig. 6.4. We decided to focus on object *ATM* and create the state machine diagram that reflects its behavior.

Example 1

Let us first consider the scenario *Scenario authenticate* and let us see the state machine diagram resulting from it. In SCED, the generated state machine diagram for object ATM resulting from this scenario appears in Fig. 6.5; this is an actual snapshot from the SCED system (we have changed the layout in order to be able to view a more clear state machine diagram; we have done the same for the other state machine diagrams obtained in SCED that will appear later). The state machine obtained using our approach appears in Fig. 6.6 and we can notice that the two state machines are equivalent.

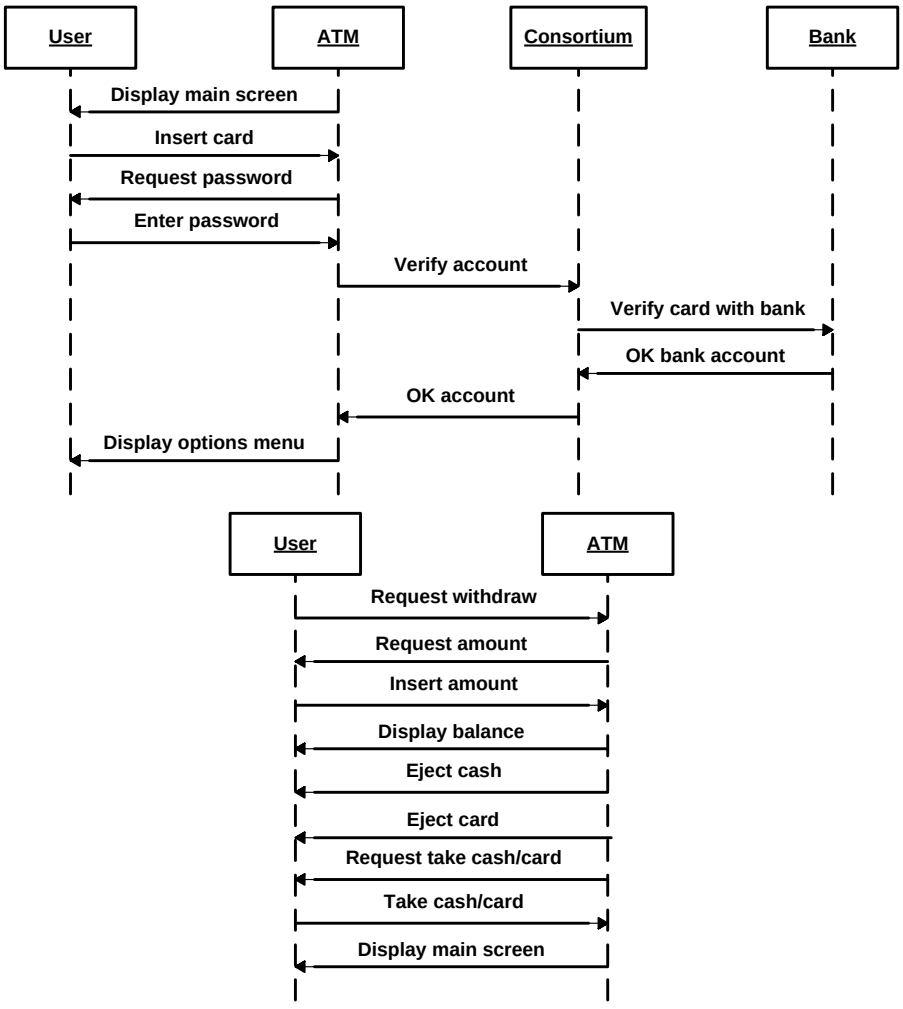


Fig. 6.4 Two scenarios for ATM example:
Scenario authenticate* and *Scenario withdraw

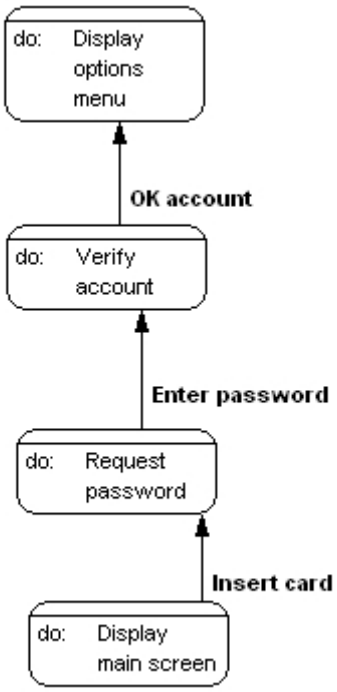


Fig. 6.5 State machine diagram for object ATM resulting from *Scenario authenticate* in SCED

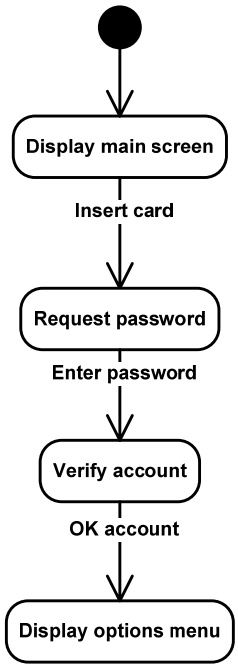


Fig. 6.6 State machine diagram for object ATM resulting from *Scenario authenticate* in our approach

Next, let us consider the other scenario, *Scenario withdraw*. The state machine diagram obtained in SCED for this scenario appears in Fig. 6.7, while the one obtained using our approach appears in Fig. 6.8. We can notice that, again, the sequence of states and transitions is basically the same in both cases. Thus, for this individual scenario, as well, we get similar results using SCED and using our approach. The question is how do the generated state machine diagrams differ in the two approaches when we consider both scenarios at the same time, not one by one. In SCED, after adding this second scenario, the generated state machine diagram for object *ATM* appears in Fig. 6.9.

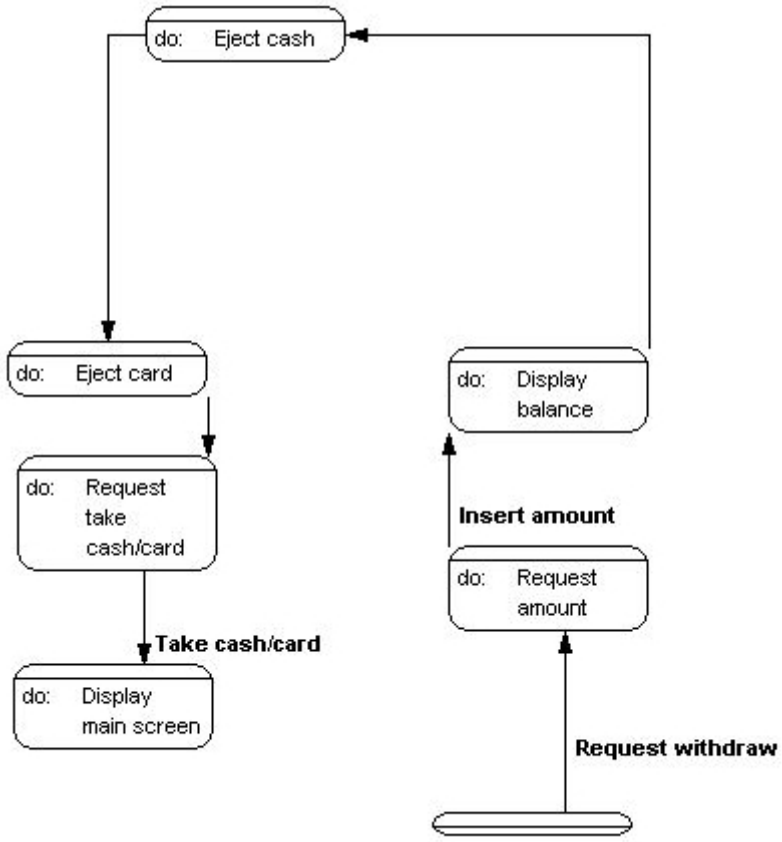


Fig. 6.7 State machine diagram for object ATM resulting from *Scenario withdraw* in SCED

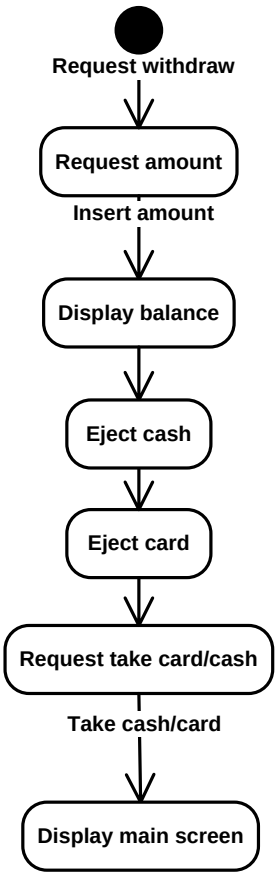


Fig. 6.8 State machine diagram for object ATM resulting from *Scenario withdraw* in our approach

We can notice that SCED considered the states resulting from *Scenario withdraw* as preceding those resulting from *Scenario authenticate*. This undesirable behavior has resulted from the fact that the message *Display main screen* (generating a state with the same name), exchanged between objects *ATM* and *User*, appears as the final message in *Scenario withdraw* and as an initial message in *Scenario authenticate*. SCED has considered that *Scenario withdraw* is immediately followed by *Scenario authenticate* and they are linked by this *Display main screen* message. We call this behavior undesirable, because the state machine shows that the behavior for object *ATM* should be such that the transitions and states involved in performing the withdrawal

operation come before those involved in authentication of the user's card. In other words, the user withdraws an amount of money first and his card is verified with the bank afterwards.

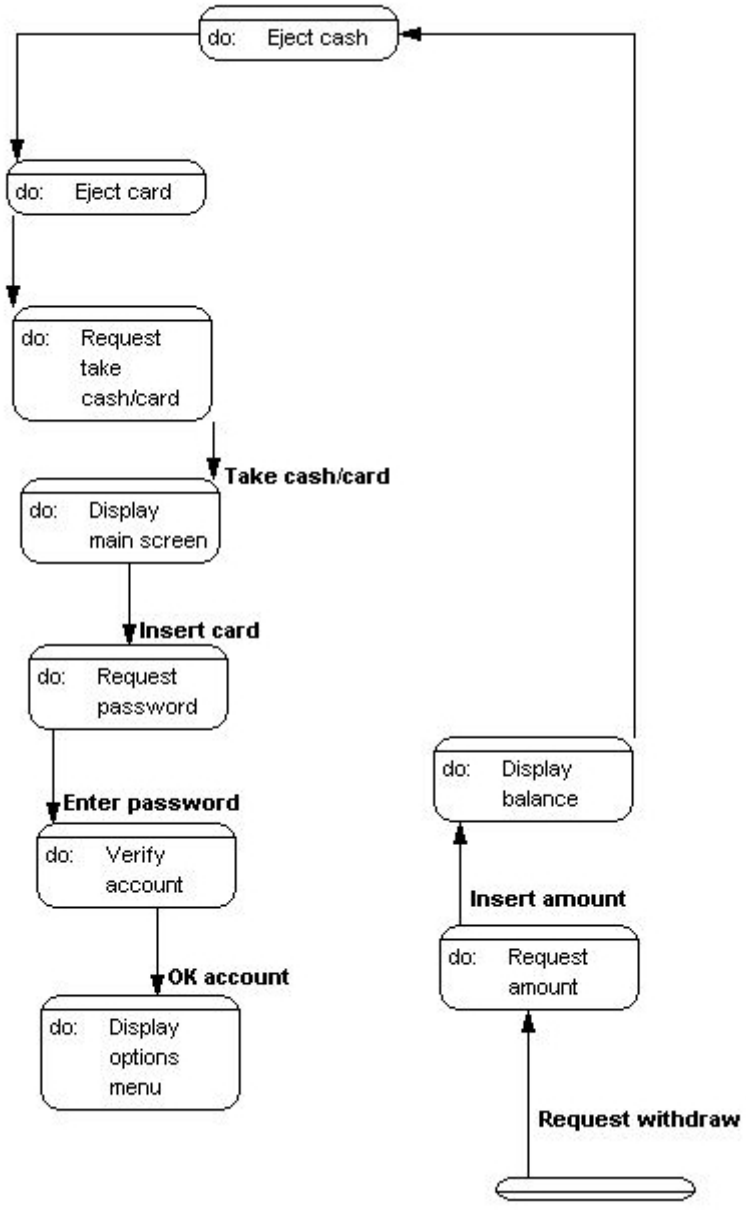


Fig. 6.9 State machine diagram for object ATM resulting from both *Scenario authenticate* and *Scenario withdraw* in SCED

In our approach, we cannot create the state machine for object ATM unless we know exactly how the two given scenarios are related. The individual state machines corresponding to each scenario are not randomly merged, but they are merged according to the relationship existing between them, relationship that needs to be specified beforehand (and which appears in the dependency diagram). In this case, it is only natural to consider that the two scenarios are related with a *succession* dependency, more exactly *Scenario withdraw* succeeds *Scenario authenticate*. The resulting state machine diagram for object *ATM* using our approach appears in Fig. 6.10.

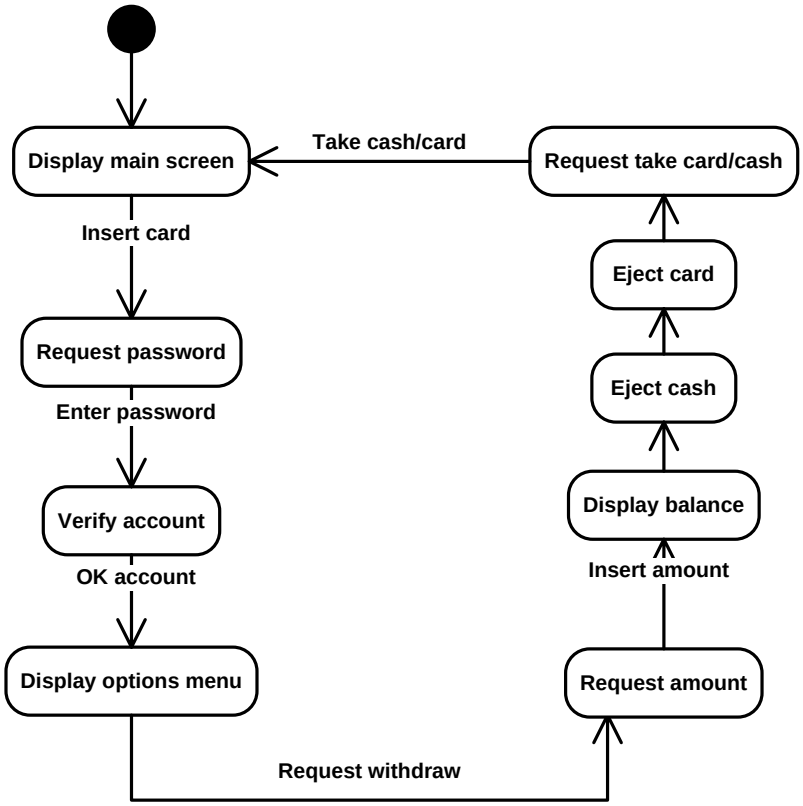


Fig. 6.10 State machine diagram for object ATM resulting from both *Scenario authenticate* and *Scenario withdraw* in our approach

We can notice that, as opposed to the state machine diagram obtained in SCED, our state machine diagram reflects clearly the fact the card has to be

authenticated first and only then it can be used to withdraw cash form an ATM system. Even though it is possible to go through the whole process of authentication after the withdrawal, it is clear that a withdrawal is not possible without the authentication talking place first. This is the desired behavior in the use of the system and our approach reflects it in the obtained state machine diagram for the object *ATM*. We have obtained this result because we have explicitly pointed out that *Scenario authenticate* is succeeded by *Scenario withdraw*; in other words, we have taken into consideration how the two scenarios are related to each other and used this information when synthesizing our state machine diagram. Unwanted behavior as the one in Fig. 6.9 could not result by using our approach (unless, of course, we specifically wanted *Scenario withdraw* to come before *Scenario authenticate* and we considered such a succession relationship).

Example 2

For a new comparison, let us consider a slightly modified example, with two scenarios as well. We will use *Scenario authenticate* as it is, as it appears in Fig. 6.4, and we will create a new scenario, named *Scenario withdraw transaction*, looking very similar to the original *Scenario withdraw*. The only difference is that we replace the last message in *Scenario withdraw*, that is *Display main screen*, with the message *Transaction completed*. We do this so that we do not have a message with the same name in the two scenarios. The new scenario appears in Fig. 6.11.

Let us focus on object *ATM* again and create the state machine diagram that reflects its behavior. The individual state machine diagrams, obtained by considering only *Scenario authenticate*, are the same (that is, Fig. 6.5 in SCED and Fig. 6.6 in our approach). As for *Scenario withdraw transaction*, the individual state machine diagrams which consider only this scenario appear in Fig. 6.12 (using SCED) and Fig. 6.13 (using our approach).

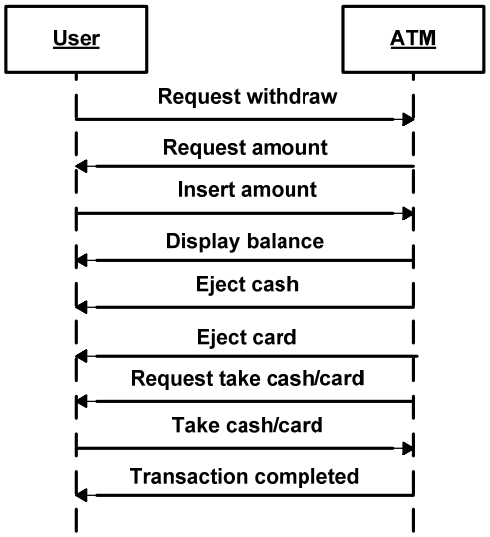


Fig. 6.11 Scenario *Scenario withdraw transaction*

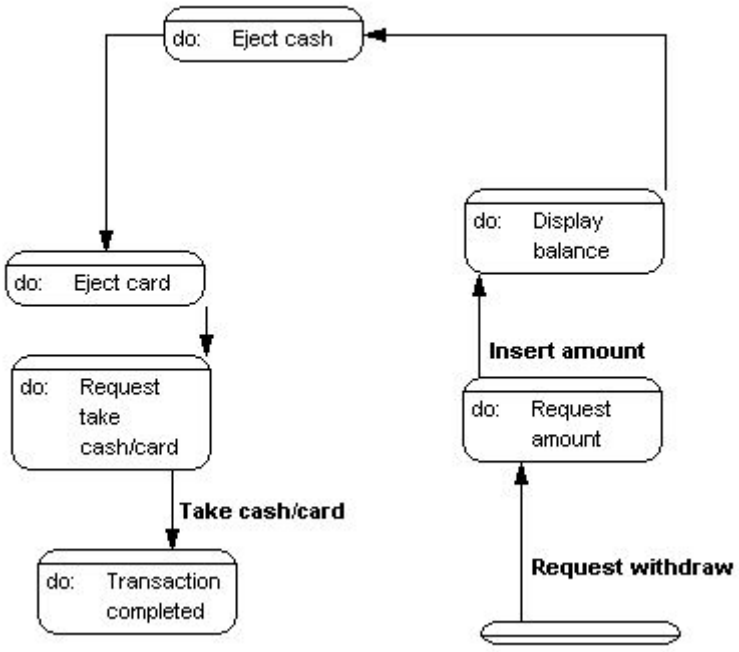


Fig. 6.12 State machine diagram for object ATM resulting from *Scenario withdraw transaction* in SCED

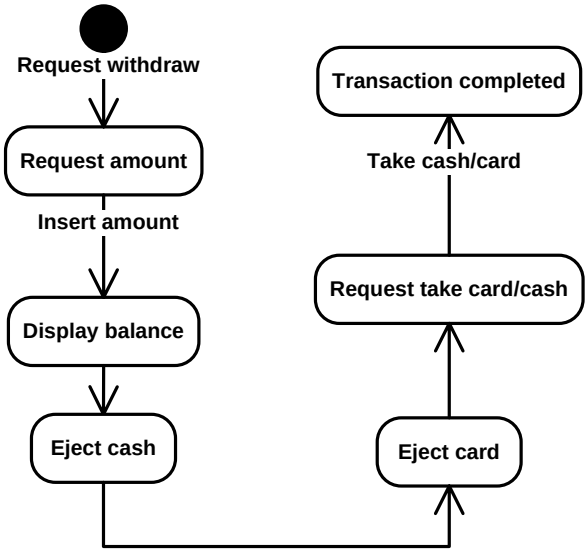


Fig. 6.13 State machine diagram for object ATM resulting from *Scenario withdraw transaction* in our approach

Now, let us consider both scenarios and create the state machine diagram for object ATM that reflects the overall behavior in these two scenarios. Using SCED, we obtain two separate state machines, as in Fig. 6.14. From SCED's point of view, there is no apparent relationship between the two scenarios and therefore the result will be two separate, unrelated, state machine diagrams.

In our approach, again, we have to know first the relationship between the scenarios. We will consider this time as well that we have a succession relationship, that is *Scenario authenticate* is succeeded by *Scenario withdraw transaction*. The result appears in Fig. 6.15.

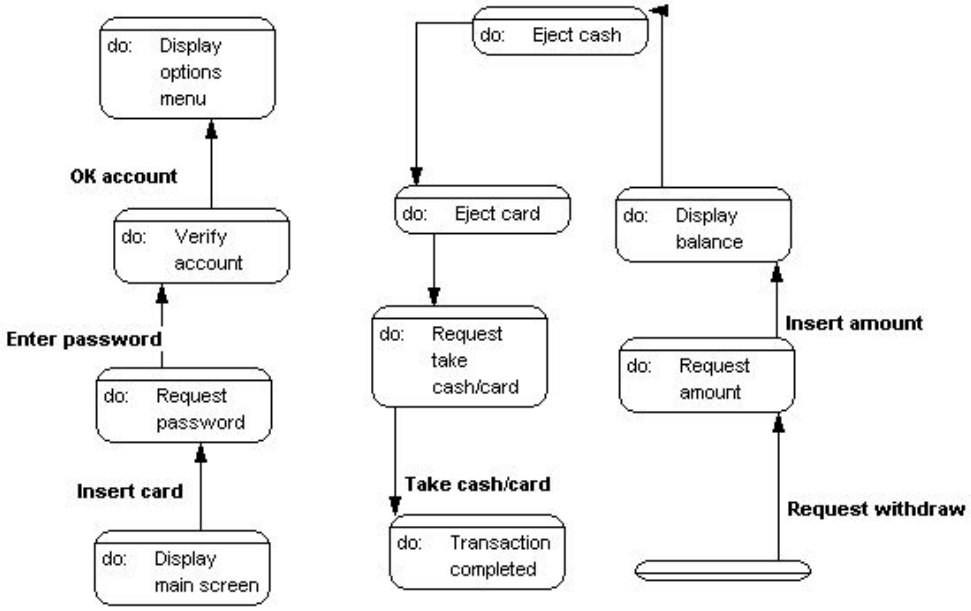


Fig. 6.14 State machine diagram for object ATM resulting from both *Scenario authenticate* and *Scenario withdraw transaction* in SCED

We observe that in case the individual scenarios have no common messages, SCED will have no way of merging the resulting state machines and therefore it will have separate state machines, one for each scenario where the object appears. This is the same in phase III.1 in our approach, but the considerable difference is that we do not stop at this point, we are not content with a number of isolated, individual state machines, each one reflecting the behavior in one scenario. We want to illustrate the overall behavior of the object and we do so by merging the individual state machine diagrams (*initial* state machine diagrams, as we call them) during phase III.2 of our synthesis process. Therefore, while in the case of SCED there are two resulting state machine diagrams and no knowledge about how they are connected, in our approach there is one unified state machine diagram, reflecting the object's behavior as a whole. This state machine diagram can be used further in the development process, for generating code for the given object.

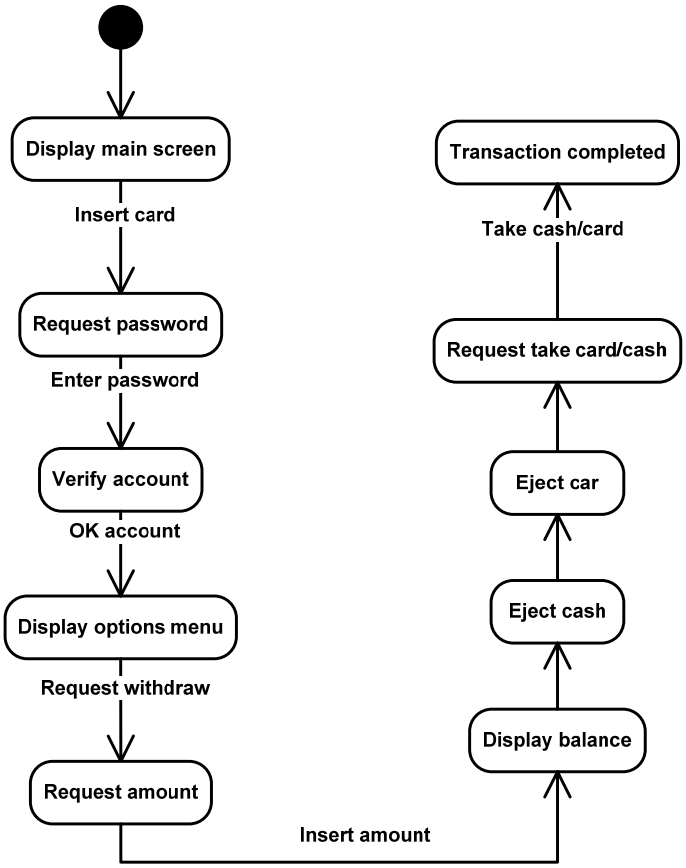


Fig. 6.15 State machine diagram for object ATM resulting from both *Scenario authenticate* and *Scenario withdraw transaction* in our approach

We have chosen these examples that involve a succession relationship to prove some important points, as summarized below. We should mention here that SCED does not support concurrent state machines, while our system allows their use.

We therefore conclude that without clear guidance as to the relationship between scenarios, there are two drawbacks:

1. we might obtain undesirable behavior (like the one in Fig. 6.9) or
2. we might obtain several seemingly unrelated state machine diagrams, not the complete behavior of the desired object in one unified state machine diagram.

Comparison summary

Summarizing, here are the main features of SCED, from the point of view of the generated state machine diagrams:

- for one object, one scenario results in one state machine diagram, while *n* scenarios might result (in most cases) in *n* state machine diagrams;
- the system considers the union of the state machines;
- if common messages exist between them, they are merged;
- there is no regard for inter-scenario relationship; actually, there cannot be any such regard, since the system has no idea about the relationship between scenarios;
- does not support concurrency;
- given a number of scenarios, the generated state machine diagram is unique.

In contrast, our approach's main features, from the same point of view, are:

- for one object, one scenario results in one state machine diagram, while *n* scenarios also result (in most cases) in one state machine diagram;
- when creating the state machine diagram for an object involved in two or more scenarios:
  - the system creates one state machine diagram for each scenario where the object appears (the state machine diagrams are not merged by default, no matter how many common messages they have);
  - the system checks the relationship between the scenarios and THEN does the merging;
  - concurrency in the state machine is possible, if concurrent scenarios were involved;
  - given a number of scenarios, usually there is one state machine diagram describing the whole behavior of an object;
  - depending on the relationship between the scenarios involved, we have different state machine diagrams.

This last feature is of outmost importance when we try to find out the complete behavior of an object. Different relationships between the given scenarios result in different behaviors of the object and thus it is crucial to know exactly which is the relationship that expresses the requirements given in the scenarios. This is what makes the dependency diagrams so useful and important; they help in obtaining the state machine diagrams that express the complete and accurate behavior of the objects involved.

We therefore consider that by using our approach we can obtain state machine diagrams that reflect the behavior expressed in the given scenarios. By merging the information in all the individual state machine diagrams, obtained for all the scenarios where the desired object appears, we can deliver a complete and unambiguous behavior of the object and this can further be helpful in generating its implementation.

Moreover, our approach can be integrated into other systems which transform scenario models (like sequence diagrams) into state machine diagrams. The construction of the dependency diagrams has to be added; as for the behavior illustrated in these diagrams, that is the relationships between scenarios, this has to be integrated in the system’s mechanism that transforms the scenario models into state machine diagrams.

Chapter 7

Concluding Remarks

We have proposed a means of support for the dynamic modelling during the design phase of developing a software system. Our proposal is based on the use of a new type of diagrams, named dependency diagrams, which are able to represent the various kinds of relationships existing between scenarios. They present the advantage of an enhanced traceability and a better overview of the system. We have classified the possible relationships between various scenarios and created the possibility to represent these relationships in the dependency diagrams.

Our focus is on dynamic models, more specifically on sequence diagrams and state machine diagrams. In order to make use of the benefits of both models, we propose a process of transformation of sequence diagrams (as representations of scenarios) into state machine diagrams. This transformation is performed by taking into consideration the relationships between the given scenarios.

The obtained state machine diagrams can be used for detailed design models and code can further be generated from them. Using our approach, we can support both the analysis and design phases and we can bring the developer one step closer to the implementation.

Moreover, dependency diagrams could be used during the testing phase; they can facilitate the generation of new test cases by traversing paths through them. We intend to explore this possibility in the future. Two more possible future work directions are described in the following.

7.1. Using our approach in real-time systems

When modeling a real-time system, not only structural and dynamic behavior have to be considered, but also time constraints are mandatory for correctness [47]. Time constraints appear in the category of non-functional requirements, along with scalability, cost etc.

Sequence diagrams allow the expressing of some time constraints, at a very basic level. Sequence diagrams often represent scenarios, while scenarios are paths through use cases. It is well known that use cases are mainly used for functional requirements, therefore not very well suited for showing non-functional requirements. Information about non-functional requirements can be added in a supplementary specification, which is different from the one appearing in use-cases [48]. Actually, for real-time systems, it is quite possible that a greater percentage of the system requirements reside in the supplementary specification.

The most widely used way to express non-functional requirements in general and time constraints in special is through annotations.

When applying our approach to real-time systems, the sequence diagrams can be extended with specific notation for real-time situations. The question is if the dependency diagrams would also require an extended notation. As for how this information appears in state machine diagrams, one possibility would be the already used annotations. This is an issue that needs further and thorough exploration.

7.2. Consistency check between scenarios and state machines

Our final purpose is obtaining state machine diagrams that reflect the complete behavior of the objects involved. One important issue for obtaining a correct and complete final state machine diagram for each object addresses the consistency between the state machines and the scenarios. We have to make sure that the behavior 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 behavior appearing in the state machines, and the possible conflicts that might arise. Throughout the whole transformation process, starting from the normalization of sequence diagrams and ending with synthesizing the final state machine diagram, we should always make sure that the behavior expressed in scenarios has not been altered.

In our current approach, the consistency check is done by verifying the states and transitions in the state machine diagram against the messages in the sequence diagrams. For now, it is a process performed manually, but as part of our future work we intend to develop automated methods that verify whether consistency is accomplished.

Bibliography

[1] 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.

[2] Functional Requirements and Use Cases: http://www.bredemeyer.com.

[3] UML’s sequence diagram – Donald Bell - http://www-128.ibm.com/developerworks/rational/library/3101.html.

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

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

[6] M. Glinz, “Improving the quality of requirements with scenarios”, in P*roc. 2nd World Congress on Software Quality*, Yokohama, 2000; pp. 55-60.

[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”, in *Proc. 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. Ryser and M. Glinz, “Using dependency charts to improve scenario-based testing”, in *Proc. of the 17th International Conference on Testing Computer Software (TCS2000)*, Washington D.C., 2000.

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

[15] S. Vasilache and J. Tanaka, “Using dependency diagrams in dynamic modelling of object-oriented systems”, in *Proc. of the 7th IASTED Conference on Software Engineering and Applications (SEA2003)*, Marina del Rey, USA, 2003, pp. 277-283.

[16] S. Vasilache and J. Tanaka, “Synthesis of state machines from multiple interrelated scenarios using dependency diagrams”, in *Proc. 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004)*, Orlando, Florida, USA, 2004, pp. 49-54.

[17] S. Vasilache and J. Tanaka, “Bridging the gap between analysis and design using dependency diagrams", in *Proc. 3rd International Conference on Software Engineering, Research, Management and Applications (SERA2005)*, Mt. Pleasant, Michigan, USA, Aug. 11-13, 2005, pp. 407-414.

[18] M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, M. Veanes, “Validating Use-Cases with the AsmL Test Tool”, in *Proc. 3rd International Conference on Quality Software (QSIC2003)*, Dallas, USA, 2003.

[19] K. K. Breitman, JCSP Leite, D. M. Berry, “Supporting scenario evolution”, *Requirements Engineering*, (2005) 10, pp. 112-131.

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

[21] A. Knapp and S. Merz, “Model Checking and Code Generation for UML State Machines and Collaborations”, in *Proc. 5th Workshop on Tools for System Design and Verification*, Reisenburg, Germany, 2002, pp. 59-64.

[22] I. A. Niaz and J. Tanaka: "An Object-Oriented Approach To Generate Java Code From UML Statecharts", *International Journal of Computer & Information Science (IJCIS)*, Vol.6, No.2, 2005, pp. 83-98.

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

[24] F. Bordeleau, J. P. Corriveau, "On the need for "state machine implementation" design patterns", *Scenarios and state machines: models, algorithms, and tools, International Conference on Software Engineering ICSE 2002 Workshop*, Orlando, Florida, USA, 2002.

[25] Scott W. Ambler, *"The Elements of UML 2.0 Style"*, Cambridge University Press, 2005.

[26] Stephane S. Some, "Beyond scenarios: generating state models from use cases", in *Proc. of International Conference on Software Engineering (ICSE2002)*, Orlando, Florida, USA, 2002.

[27] T. Maier, A. Zundorf, "The Fujaba statechart synthesis approach". in *Proc. 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE 2003*, Portland, Oregon, USA, 2003.

[28] Agile modeling: http://www.agilemodeling.com.

[29] K. Breitman and JCSP Leite, "Scenario evolution: a closer view on relationships", in *Proc. 4th International Conference on Requirements Engineering (ICRE'00)*, pp. 102-111.

[30] Sebastian Uchitel, "Synthesis of behavioral models from scenarios", *Transactions on Software Engineering*, Vol. 29, No.2, Febr. 2003..

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

[32] Craig Larman, *"Applying UML and Patterns"*, Prentice Hall, 2002.

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

[34] Kai Koskimies and Erkki Makinen, Automatic synthesis of state machines from trace diagrams, Software – Practice and Experience, 24(7), 1994, pp. 643-658.

[35] S. Vasilache and J. Tanaka, "Synthesizing statecharts from multiple interrelated scenarios", in *Proceedings of the International Symposium on Future Software Technology (ISFST2001)*, Zheng Zhou, China, Nov. 5-8, 2001, pp. 158-163.

[36] S. Vasilache and J. Tanaka, "Support in the Software Development Process Using Dependency Diagrams," *International Journal of Computer & Information Science (IJCIS)*, Vol.7, No.1, 2006 (to appear).

[37] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.

[38] DotnetCoders: Guide to UML diagrams http://www.dotnetcoders.com/web/learning/uml/default.aspx.

[39] H. E. Eriksson, M. Penker, B. Lyons, D. Fado, ” UML Toolkit”, Wiley, 2003.

[40] The DYNAMO Home Page: http://www.cs.aau.dk/~normark/dynamo.html

[41] http://www.math-cs.gordon.edu/courses/cs211/ATMExample/ - An Example of Object-Oriented Design: An ATM simulation.

[42] Develper.com: UML Overview, By Mandar Chitnis, Pravin Tiwari, & Lakshmi Ananthamurthy, http://www.developer.com/design/article.php/1553851

[43] Sparx Systems UML Tutorial: http://www.sparxsystems.com/resources/tutorial/.

[44] SearchWinIT.com: http://searchwinit.techtarget.com/sDefinition/0,,sid1_gci936454,00.html

[45] OMG (Object Management Group): http://www.omg.org/

[46] E. Makinen, T. Systa, “MAS – an interactive synthesizer to support behavioral modeling in UML”, in *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, 2001, pp. 15-24.

[47] J. Seemann, J. Wolff v. Gudenberg, “Extension of UML sequence diagrams for real-time systems”, *Lecture Notes in Computer Science 1618*, June 1998, pp. 240-252.

[48] D. Hanslip, “Practical application of use cases to a real-time system”, DeveloperWorks/Rational: http://www-128.ibm.com/developerworks/rational/library/5272.html.

[49] D. Amyot, “Introduction to the user requirements notation: learning by example”, Computer Networks (42), 2003, pp. 285-301.

[50] I. Jacobson, “The Unified Software Development Process”, Addison Wesley Professional, 1999.

[51] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design patterns: elements of reusable object-oriented software”, Addison Wesley Professional, 1995.

[52] James W. Cooper “Java design patterns”, Addison-Wesley, 2000

[53] Sun Microsystems, Java Technology: http://java.sun.com/

[54] Eclipse.org home: http://www.eclipse.org/

[55] S. Gerard, I. Ober, “Parallelism/Concurrency specification within the UML”, White paper, UML Conference, Toronto, Canada, 2001.

Author’s Publication List

1. S. Vasilache and J. Tanaka, “Translating OMT state diagrams with concurrency into SDL diagrams", *Proceedings of the International Symposium on Future Software Technology (ISFST2000)*, Guiyang, China, Aug. 28-31, 2000, pp. 21-26.

2. S. Vasilache and J. Tanaka, "Synthesizing statecharts from multiple interrelated scenarios", *Proceedings of the International Symposium on Future Software Technology (ISFST2001)*, Zheng Zhou, China, Nov. 5-8, 2001, pp. 158-163.

3. S. Vasilache and J. Tanaka, "Using dependency diagrams in dynamic modeling of object-oriented systems", *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003)*, Marina del Rey, USA, Nov. 3-5, 2003, pp. 277-283.

4. S. Vasilache and J. Tanaka, "Bridging the gap between analysis and design using dependency diagrams", Proceedings of the 3rd International Conference on Software Engineering, Research, Management and Applications (SERA2005), Mt. Pleasant, Michigan, USA, Aug. 11-13, 2005, pp. 407-414.

5. S. Vasilache and J. Tanaka, "Synthesis of state machines from multiple interrelated scenarios using dependency diagrams", *Journal of Systemics, Cybernetics and Informatics*, Vol. 3, No. 3, 2006 (8 pages).

6. S. Vasilache and J. Tanaka, "Support in the Software Development Process Using Dependency Diagrams," *International Journal of Computer & Information Science (IJCIS)*, Vol.7, No.1, 2006 (to appear).