Using dependency diagrams in dynamic modelling of object-oriented systems

Simona Vasilache
Institute of Information Sciences and Electronics
University of Tsukuba, Japan
simona@iplab.is.tsukuba.ac.jp

Jiro Tanaka
Institute of Information Sciences and Electronics
University of Tsukuba, Japan
jiro@is.tsukuba.ac.jp

ABSTRACT

In defining the behaviour of a system, requirement specifications make use of a number of scenarios that are interrelated in many ways. Most of the current approaches, even though giving directions on how to translate them into state machines, treat each scenario separately. In this paper we propose a method of synthesizing state machines from multiple scenarios, with respect to the relationships among them.

We propose a new type of diagrams that are able to illustrate the relationships and dependencies among scenarios. We call these diagrams *dependency diagrams*. We have noticed that, when trying to synthesize state machines from scenarios, different relationships between scenarios result in different state machine structures. By emphasizing these relationships, representing them and using them directly in the synthesis process, we manage to overcome this problem. We also propose a set of rules and steps for the synthesis of state machines from multiple interrelated scenarios, based on the initial scenarios and on the newly introduced dependency diagrams, as a means to properly describe the requirements specifications of a system.

KEY WORDS

object-oriented analysis and design, dynamic modelling, scenarios, state machines

1 Introduction

One of the most important phases in software development is represented by the requirements analysis. The main task of the requirements analysis is to generate specifications that describe system behaviour unambiguously, consistently and completely [1]. Several object-oriented methodologies, like OMT and UML, [2], [3] make use of scenarios as a means of capturing requirements specifications, as well as a means of communication between clients and software developers. Together with use cases, scenarios have gained considerable popularity during the recent years. A scenario is a sequence of events that occurs during one particular execution of a system [2].

Although popular, scenarios have not received the attention they actually deserve, more exactly, they have not been used up to their entire potential. Their usefulness lies not only in the ability to capture requirements, but in their applicability when used in conjunction with other models. We specifically refer to what is called ”behaviour models”, that is models that describe the behaviour of a system.

When it comes to these dynamic aspects of a system, state machines (particularly statecharts, originally introduced by D. Harel [4]), represent a compact way of describing these aspects. Statecharts are finite state machines extended with hierarchy and orthogonality, allowing the representation of a system in a compact and elegant manner. It is because of this feature that they have been preferred for representing scenarios.

While scenarios represent a single trace of behaviour of a complete set of objects, state machines represent the complete behaviour of a single object. The two concepts together provide an orthogonal view of a system.

Our intention is to provide a method of synthesizing state machines from multiple scenarios, with regard to the relationships among them. For this purpose, we will introduce *dependency diagrams*, which show all the relationships between the various scenarios described in the early phase of software development. Based on these dependency diagrams and on the initial scenarios, we will give rules and steps of synthesis of state machines from multiple interrelated scenarios. We will describe in this paper the newly introduced diagrams and our method of synthesis.

2 Sequence diagrams and state machines

2.1 Scenarios as sequence diagrams

Scenarios are represented as sequence diagrams in UML[3]. Sequence diagrams represent interactions between objects from a temporal point of view. An object is represented by a rectangle and a vertical bar called the object's lifeline. Objects communicate by exchanging messages, represented by horizontal arrows drawn from the message sender to the message recipient. The message sending order is indicated by the position of the message on the vertical axis.

2.2 State machine diagrams

State machine diagrams represent state machines from the perspective of states and transitions. The representation used in UML is inspired from Harel's statecharts [4]. State diagrams describe which states an object can have during its life-cycle and the behaviour in those states, along with what events cause the state to change.

All objects have a state; the state is a result of previous activities performed by the object. An object changes state when something happens, which is called an event. State diagrams may have a starting point and several end points. A state is represented as a rounded rectangle; between states there are state transitions, shown as a line with an arrow from one state to another. The state transitions may be labelled with the event causing the transition. When the event happens, the transition from one state to another is performed (the transition is "triggered"). This means that the system leaves its current state, initiates the actions specified for the transition and enters a new state.

State machines have proved their usefulness in the dynamic description of the behaviour of a system. Moreover, they can be used for generating code directly from them, since each of them describes the complete behaviour of one object.

3 State machines from single scenarios

Synthesizing state machines from single scenarios represents the basis for the synthesis from multiple scenarios. In the following, we are going to describe concisely how to obtain state machines from single scenarios.

A state machine diagram is a graph whose nodes are states and whose directed arcs are transitions (labelled by event names).

One state machine diagram describes the behaviour of a single class of objects. The sequence of events in a sequence diagram corresponds to paths through the state machine diagrams of the corresponding objects [5]. In order to construct a state machine for a class of objects, we have to consider the vertical line that corresponds to the objects of that class. We can define in the following the basic rules for generating state machine diagrams from single scenarios.

For an object in a sequence diagram, incoming arrows represent events received by the object and they become *transitions*. Outgoing arrows are actions and they become *actions* of the transitions leading to the states. The intervals between events become *states*. A state reflects the response of an object to input events. This response may include an action or a change of state by the object. While events represent certain points in time, states represent intervals of time. Before receiving any event, the object is in the *default state*.

Fig.1 depicts a simple example of a scenario in a classical ATM (Automated Teller Machine) system. The ATMs are shared by a consortium of banks. Each ATM accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash and prints receipts. Throughout the paper, we will describe typical simple 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, transfer) and others. Four objects are involved in our scenario: *user*, *ATM*, *consortium* and *bank*. The events in our example are: displaying the main screen (from the ATM to the user), inserting a card (from the user to the ATM), requesting password (from the ATM to the user), entering password (from the user to the ATM), and so on.

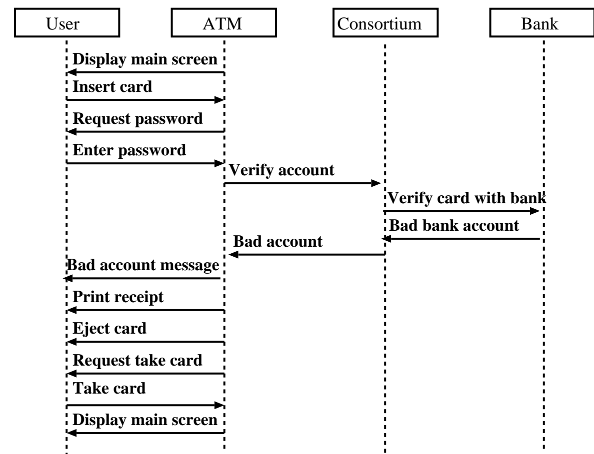

Figure 1. Sequence diagram (scenario) in an ATM system

In this scenario, after the user enters the card and then the password, the ATM verifies the card with the consortium, which, in turn, verifies it with the bank. The bank sends a bad bank account event to the consortium, and the consortium sends a bad account event to the ATM. The ATM, in turn, sends a bad account message event to the user. In the end, a receipt is issued, the card is ejected and the user is requested to take the card back. Fig.2 illustrates the state machine diagram corresponding to the ATM object in the scenario above. (*Display main screen* is considered to be the default state.) *Insert card*, *Enter password*, *Bad account*, *Take card* have become transitions, while *Display main screen*, *Request password*, *Verify account*, *Bad account message*, *Print receipt*, *Eject card* and *Request take card* are the actions that led to the states with the same names respectively.

4 State machines from multiple scenarios

4.1 Classification of relationships

According to their definition, scenarios represent partial descriptions of system behaviour [2]. In order to describe a system completely, a number of scenarios is needed. Understanding the dependencies helps in understanding the

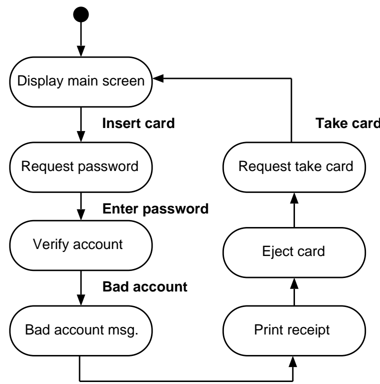

Figure 2. State machine diagram for the ATM class

requirements better, as well as obtaining an accurate design. Traceability is also enhanced, since knowing the dependencies makes it easy to see what other scenarios would be affected if one scenario is changed [8].
Sometimes, one scenario follows another scenario, or is conditioned by another one. Many times the order and the timing of their execution are not arbitrary.
For example, if we consider the scenario depicting the action of withdrawing cash, this can be executed only in the situation of the user possessing a valid card and inputting the correct password. The scenario of the user applying for a card with a bank must precede the scenarios involving transactions with the bank in the user's name. Therefore, we consider that in order to be able to understand and describe the whole system, we need to take into account not only the scenarios themselves, but also the relationships between them.
Two or more scenarios can be related in many ways: the execution of a scenario can depend on the execution time of another one (e.g. it can only be executed after/before another scenario), the necessary conditions for a scenario to be executed are described (and have to be fulfilled) in a different one, one scenario represents a part of another, a set of scenarios are very similar with each other, representing a variant of a basic scenario and so on.
We classify the relationships/dependencies between scenarios as follows:

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

A time dependency signifies the fact that one scenario has to be executed at an earlier/later moment in time than another scenario. Only after the scenario that has to be executed first has finished its transitions, the second one can start its execution. It can also mean that two (or more) scenarios must be executed at the same time. For instance, as described above, a user must first prepare a card and only then (s)he can perform transactions through the ATM. Therefore, the scenario of creating a card precedes the scenario of withdrawing cash and the two scenarios are in a time dependency.
A cause-effect dependency reflects the fact that the execution of a scenario can take place only the moment certain conditions (established in another scenario) become valid. For example, an ATM can satisfy the user's request for withdrawing cash only if it has been previously provided with a number of bills/coins. The scenario of withdrawing cash depends on the scenario of the ATM machine being "loaded" with a sufficient amount of cash (considered to cover the maximum amount that could be withdrawn during a whole day). The condition of "being able to provide enough cash" is established in a different scenario from the one where the transaction itself takes place.
A generalization dependency emerges when one scenario is a constituent part of another one or a variant of it. As a rough example, we can consider that the scenario of withdrawing cash is very similar to the scenario of depositing money. They can be generalized under one scenario, "cash operations", for example, where we have 2 variants with slight differences between them (in the case of deposit: the user selects "deposit" and inserts the money in the special slot in the ATM; in the case of withdrawal: the user selects "withdrawal" and the money is ejected through the same slot).
One could argue that time dependencies and cause-effect dependencies are equivalent, but we believe that it is important to emphasize when the dependency arises from a specific time sequence (like having to insert the card and password first, and only after that being able to perform a transaction) and when a dependency arises from certain conditions that are not explicitly time-related (at least, not necessarily). As we described above, a user could withdraw cash only if the ATM has been provided with bills and coins. Here, it is not so important to emphasize the time sequence (supplying bills first and then being able to satisfy the user's request for cash), as it is important to emphasize that having the bills is a necessary condition, which if it is not met, the operation cannot take place.
When we deal with time (as well as cause-effect) interdependent scenarios, the execution order of the scenarios defines, in most cases, these dependencies. The execution order of a number of scenarios falls into one of the following categories:

- succession;
- disjunction;
- conjunction;
- recurrence.

Succession refers to the fact that one scenario follows another one. Disjunction indicates that at a certain moment in time either one or another scenario is executed. Conjunction shows that two (or more) scenarios are executed at the same time, while recurrence denotes that a scenario is executed iteratively a certain number of times.

4.2 Introducing dependency diagrams

In view of the fact that the purpose of our work is obtaining state machine diagrams, the fact that different relationships between scenarios result in different state machine structures is of considerable importance. This is why we believe that the relationships between scenarios should not be ignored. In order to represent these relationships, we will introduce *dependency diagrams*. The notation used in these diagrams is based on the notation used in Message Sequence Charts [12].
One scenario is represented as a rounded rectangle, with connectors for *start point* and *end point* (corresponding to entry and exit points). The positioning in space of different scenarios shows the order of execution.
The basic notation is illustrated in Fig.3.

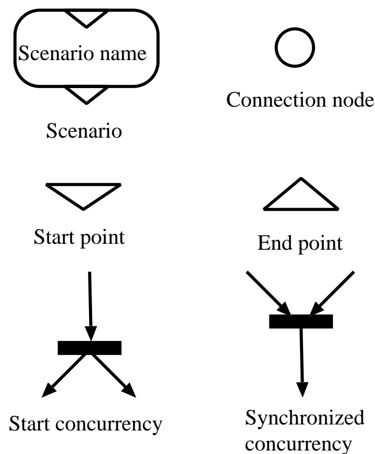

Figure 3. Basic notation for dependency diagrams

A simple example of a dependency diagram is shown in Fig.4. It is based on the same example of ATM, where we consider *Scenario start* the initial scenario (where 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 either of the 3 operations of withdrawing cash, depositing cash or transferring cash, that is either of *Sc. withdraw*, *Sc. deposit* and *Sc. transfer* scenarios respectively. We also suppose that when the user changes his(her) password (*Sc. chg. pass*), the scenario *Sc. videotape* takes place simultaneously (that is, the user is videotaped during the operation of changing the password). Fig.4 illustrates 3 alternative scenarios (any of them can be executed after *Scenario start*), as well as the concurrency of 2 scenarios, *Scenario chg. pass* and *Scenario videotape*.
Several constraints must be kept in mind when representing the dependency diagrams. Some of them are mentioned in the following. The dependency diagram must have a single start point (but can have several end points). The return of a loop can only be linked to a connection node. The end of synchronization point forces the flow of control to

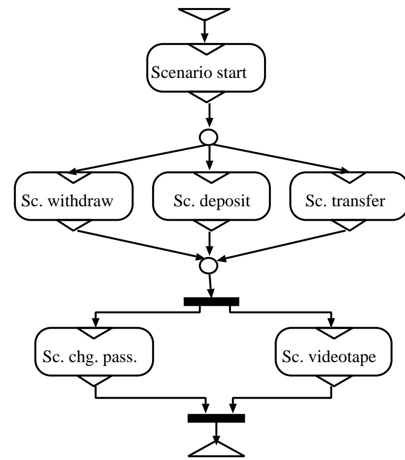

Figure 4. Simple dependency diagram for ATM

wait until the end of each of the concurrent scenarios before continuing. One block containing these concurrent scenarios is considered as one entity, so no derivation and loops are possible before the resynchronization point.
The different internal elements of scenarios are not represented, but if an arrow links the border of two scenarios, this means that there is a connection between two internal scenario messages and (above the linking arrow) the name of this linking message must be written.

4.3 Algorithm of transformation

Our methodology proposes the following major phases:

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

The third phase involves two steps, *for each object* in the system:
- creating one state machine diagram for each scenario;
- synthesizing the final state machine diagram from all the state machine diagrams, based on the information in the dependency diagrams.
The number of final state machine diagrams will be equal to the number of objects in the system.
We present in the following an overview of the algorithm used for synthesizing the initial state machine diagrams. The algorithm is applied in the same way to all the existing objects. Thus, for each object, it consists of the following steps:

1. create empty state machine diagrams, one for each scenario where the object appears;

2. for each state diagram, create all events (corresponding to transitions *to* the object);
3. for all transitions *from* the object, create actions that will lead to states and create the respective states;
4. set the right time sequence for the transitions.

The algorithm takes as input all scenarios in which the object is involved and gives as output an equal number of state machine diagrams.
Step 1 creates a state diagram for every distinct scenario involving our object. Considering that we focus on the object *ATM* and since in our example we presented only one scenario, the one in Fig.1, step 1 will create only one empty state machine diagram. (After obtaining the final state machine diagram for *ATM*, we will proceed in the same way for the other objects, like *User*, *Consortium* and *Bank*).
Step 2 creates all events corresponding to transitions to the object. In our example, it creates *Insert card*, *Enter password*, *Bad account*, *Take card*.
In step 3 the actions that lead to states are created, that is *Display main screen*, *Request password*, *Verify account*, *Bad account msg.*, *Print receipt*, *Eject card* and *Request take card*. States with the same names are created as well.
During this step the default state has to be specified; in our case, it is *Display main screen*.
At this point, the transitions are not set into the right time sequence. This is the task of step 4, where - for all transitions - the source and the destination are identified, that is all transitions will be associated a starting point and an end point.
We can notice here that, for example, there is no event received by the ATM object in between the states *Print receipt*, *Eject card* and *Request take card*. This means that we could merge the 3 states into a single one, since there is nothing that could alter this succession of states.

4.4 Synthesizing the final state machine

After creating one state machine diagram for each scenario where the object appears, we continue with the synthesis of the final state machine diagram for that object. We will combine all the initial state machines, making use of the information in the dependency diagrams.
Based on the classification of relationships between scenarios, there are several rules that need to be applied in the synthesis process:

- In a succession of two scenarios, the resulting state machine diagram merges the two basic corresponding state machine diagrams.
- For two scenarios related with a disjunction relationship, their corresponding state machines should be combined with OR.
- If two scenarios are executed at the same time, their corresponding state machines must be combined with AND.
- In the final phase, the state machine diagrams should be refined, with respect to aggregation of states and generalization of states.

We feel the need to specify here that a complete state machine diagram does not have to be extremely complex. At any level, details can be omitted and can be modelled in separate lower level diagrams. The concept of *state hierarchy* is very useful and can be used to decrease the number of transitions in a state machine diagram.
The steps and rules above apply to disjoint scenarios only, because the states of the component scenarios must be disjoint for proper composition. However, it is possible that some scenarios overlap. Most of the times this happens when scenarios describe variants of the same portion of the process. The overlapping must be treated before the composition.
There are two choices for this: the scenarios that overlap can be decomposed into mutually disjoint scenarios (subscenarios) or they can be merged into a single, more complex scenario. We consider the first option more appropriate, since it allows an easier synthesis of the state machines. Subscenarios have proved to be helpful in various situations. Mainly, as we mentioned above, when common behaviour is detected in two or more scenarios, we can split the scenarios into subscenarios, so that the common part appears as one single subscenario and therefore can be considered a unique entity. Also, when we have a complex process described in one scenario, by splitting it into subscenarios we can create more simple structures, so that the requirements are understood easier. If we want to emphasize a certain aspect inside one scenario, we can separate it clearly from the rest of the scenario, and include it into one subscenario. Summarizing, subscenarios are used in one of the following situations:
- common behaviour is detected in several scenarios;
- a complex course of actions appears in a scenario;
- we need to enhance a situation with a concrete and precise goal inside a scenario [9].
We should pay attention, though, to how many levels we generate, in case we create subscenarios of subscenarios and so on.

4.5 Consistency between scenarios and state machines

The process of synthesis does not end with applying the algorithm and the rules above. Before we can say that we obtained a ”correct” final state machine diagram for each object, we need to address the issue of consistency between the state machines and the scenarios. We have to make sure that the behaviour of the final state machine diagrams reflects the information contained in the scenarios, so that we respect the requirements specifications. There are several issues that we need to consider and we emphasize some of them in the following.

Detecting implied scenarios.

After the synthesis of each state machine diagram for each object, the diagrams may present sets of behaviours that do not appear in the scenarios themselves. This unexpected behaviour and the resulting possible scenarios are called "implied scenarios" [15]. They are called implied because they are not described in the scenario specification, and therefore they allow unexpected behaviour, with respect to the original specifications.

In our approach, the detection of implied scenarios is performed manually, in the final phase, when the user checks if unexpected (and/or undesired) behaviour is reflected in the final state machine diagrams. If such a situation occurs, the state diagrams are corrected manually and the implied scenarios removed.

Messages exchanged between scenarios

There are situations when two scenarios can exchange messages between them. They could be combined into one scenario, but often, for clarifying reasons, as well as for simplicity reasons, this is not done. When illustrating the relationships between them in the dependency diagrams, the messages that are exchanged between the involved scenarios have to be represented carefully. Here it is important to identify correctly the origin and the destination of the message, in order to obtain a correct state machine diagram. We have specified in section 4.1 that the name of the message linking the border of two scenarios must appear in the dependency diagram.

We have to consider the above aspects when creating the state machine diagram corresponding to the objects involved in the exchange of messages. We must identify the origin and the destination of the messages, as well as the objects involved.

An example is given in Fig.5, where two scenarios, *Scenario withdraw* and *Scenario check (bank)* exchange two messages between them, related to the amount of withdrawal and the balance after the operation.

Conflicts between scenarios

During the phase of constructing the dependency diagrams involving so many different scenarios, conflicts between these scenarios are most probably detected. This is the time when they should be solved, and this is where the designer will take the decision related to these conflicts, so that the dependency diagrams will be drawn accordingly. This is not an automatic process, the user has the responsibility to give the appropriate indications as to how to solve the conflicts.

We are currently in the phase of developing a system that semi-automatically synthesizes state machine diagrams for all objects. This is an interactive process, because users must provide the information that cannot be automatically inferred (like solving the conflicts between scenarios or removing unwanted behaviour).

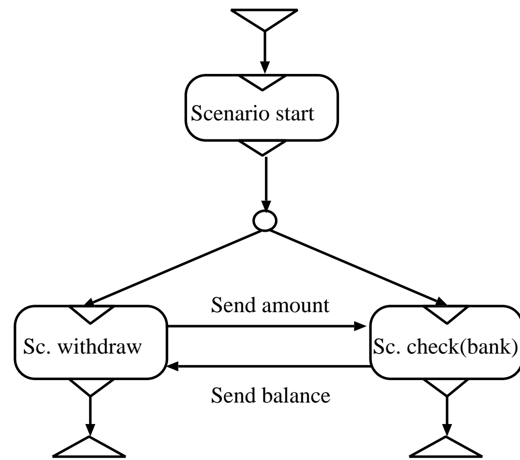

Figure 5. Dependency diagram for ATM with messages exchanged between two scenarios

5 Related work

Several papers deal with the transformation of scenario type models into behaviour models. SCED [16] is a tool for automatic generation of statecharts from single scenarios. In [13], an algorithm for generating UML statecharts from sequence diagrams is given, but the relationships between the sequence diagrams (as representations of scenarios) are limited to the introduction of hierarchy. Schonberger et. al [17] describe an algorithm for model transformation, more precisely an algorithm for transforming collaboration diagrams into state diagrams.

Collaboration diagrams describe the interaction among objects, with the focus on space. This means that the relationships (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. They show how messages are sent and received between a set of objects in order to perform a function. Although the two kinds of diagrams are similar (and called collectively *interaction diagrams*), we believe that sequence diagrams are more suited for use in the analysis phase, as they allow an easier representation of the requirements (when we think of scenarios in the usage of a system, it seems more natural and it requires less effort to focus on the time flow in the development of events).

Ryser and Glinz introduced in [8] a new kind of chart, dependency chart, and a new notation to model the dependencies between scenarios. However, the charts only show the dependencies between various scenarios, without giving directions about the way they could be used for translation into state machine diagrams.

Actually, we can observe that most work in progress related to object-oriented software development produces models that are only loosely coupled. Most methods describe how to specify models, yet do not sufficiently guide the developer in the task of transforming one model type into another.

6 Conclusions and future work

We have described a method of synthesizing state machine diagrams from multiple scenarios, with regard to the relationships among them. We have introduced dependency diagrams as a means of showing all the relationships between scenarios. Our approach offers complete requirements specifications, an accurate design, as well as an improved traceability. We are currently in the process of developing a semi-automatic and interactive system that synthesizes state machine diagrams from multiple interrelated scenarios, with the help of the information contained in the dependency diagrams.

7 Acknowledgements

The authors would like to thank Yann Jacquinot for his contribution to the development of this work.

References

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

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

[3] Rational Software Corporation. *Unified Modeling Language (UML)*, http://www.rational.com.

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

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

[6] J. Ali and J. Tanaka, An object-oriented approach to generate executable code from the OMT-based dynamic model, *Journal of Integrated Design and Process Science* 2(4), 1998, 65-77.

[7] J. Ali and J. Tanaka, Implementing the dynamic behaviour represented as multiple state diagrams and activity diagrams, *Journal of Computer Science and Information Management (JCSIM)* 2 (1), 2001, 22-34.

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

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

[10] S. Vasilache and J. Tanaka, Synthesizing statecharts from multiple interrelated scenarios, *ISFST2001*, ZhengZhou, China, 2001, 158-163.

[11] S. Vasilache and J. Tanaka, Translating OMT state diagrams with concurrency into SDL diagrams, *Proceedings of the International Symposium for Future Software Technology (ISFST2000)*, Guiyang, China, 2000, 21-26.

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

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

[14] Object Management Group, Object Constraint Language Specification, http://www.omg.org.

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

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

[17] 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, 351-383.

[18] J. Whittle and J. Schumann, Generating statechart designs from scenarios. *Scenario-based round-trip engineering, OOPSLA2000 Workshop*, Tampere University of Technology, 2000.

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

[20] R. J. A. Buhr, R. S. Casselman. *Use case maps for object-oriented systems* (Prentice Hall, 1996).

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