Synthesis of State Machines from Multiple Interrelated Scenarios Using Dependency Diagrams

Simona Vasilache
Institute of Information Sciences and Electronics,
University of Tsukuba, Japan

and

Jiro Tanaka
Institute of Information Sciences and Electronics,
University of Tsukuba, Japan

ABSTRACT

Requirements specification is one of the most important phases in developing a software application. In defining the behaviour of a system, requirements specifications make use of a number of scenarios that are interrelated in many ways. Current approaches, even though giving directions on how to translate them into state machines, treat each scenario separately. Because different relationships between scenarios result in different state machines, we believe it is significant to emphasize and represent these relationships. In order to illustrate them we propose a new type of diagrams named dependency diagrams. We offer a set of rules and steps for the synthesis of state machines from multiple inter-related 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.

Keywords: requirements specification, dynamic modelling, scenarios, state machines

1. INTRODUCTION

Requirements analysis represents a crucial phase in the software development process. The main task of the requirements analysis is to generate specifications that describe the behaviour of a system unambiguously, consistently and completely. [1]. Several popular object-oriented methodologies and notations (like OMT [1], UML [2]) make use of scenarios as a means of capturing requirements specifications, as well as a means of communication between users and software developers.

A scenario is a sequence of events that occurs during one particular execution of a system [1], it is one particular “story” of using a system. During the recent years, scenarios have gained considerable popularity. However, we believe that they have not yet received the attention they actually deserve and they have not been used up to their entire potential. Their usefulness lies not only in the ability to capture requirements, but also 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.

State machines (particularly statecharts, originally introduced by D. Harel [4]), represent a compact way of describing the aspects concerning the behaviour of a system. They allow the representation of the dynamic aspects in a compact and elegant manner and 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.

Scenarios are generally not independent of each other; various relationships and dependencies connect them. We make a classification of these relationships and in order to represent them we propose a new type of diagrams. We call these diagrams *dependency diagrams*.

Based on these dependency diagrams and on the initial scenarios, we 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 MACHINE DIAGRAMS

Scenarios as sequence diagrams

In UML, scenarios are represented as sequence diagrams. Sequence diagrams illustrate how objects interact with each other. They focus on showing the sequence of messages sent between objects, that is the interaction between objects from a temporal point of view.

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

State machine diagrams

State machine diagrams represent state machines from the perspective of states and transitions. The representation used in UML's state diagrams 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 state 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. A state transition normally has an event attached to it, but not necessarily. If an event is attached to a state transition, the transition will be performed when the event occurs. If a state transition does not have an event specified, the attached state will change when the internal actions in the source state are executed. Therefore, when all the actions in a state are performed, a transition without an event will automatically be triggered.

State machine diagrams 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. RELATIONSHIPS BETWEEN SCENARIOS; DEPENDENCY DIAGRAMS

Classification of relationships

Since a scenario represents a particular "story" of the execution of a system, in order to describe a system completely we need to know all the possible scenarios. Depending on the application, the number of scenarios varies; however small the number of all possible scenarios is, relationships, dependencies exist between them. Sometimes, one scenario follows other scenario or is conditioned by another one. Many times the order and the timing of their execution are not arbitrary.

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

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

This is why 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 interrelations between them.

When trying to synthesize state machines from scenarios, different relationships between scenarios result in different state machine structures. This fact is of considerable importance and in order to consider all the implications, we make a classification of the relationships and dependencies between scenarios as follows:

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

The execution order of a number of scenarios (defining the time dependencies) falls into one of the following categories: succession (one scenario follows another one), disjunction (at a certain moment in time either one of the scenarios is executed), conjunction (the scenarios are executed simultaneously) and recurrence (a scenario is executed a certain number of times).

Introducing dependency diagrams

In order to be able to represent and make use of these relationships, we introduce dependency diagrams. The notation used in these diagrams is based on the notation used in Message Sequence Charts [5]. One scenario is represented as a rounded rectangle, with connectors for start point and end point (corresponding to entry and exit points). The positioning in space of different scenarios shows the order of execution.

A simple example of a dependency diagram is shown in Fig.1. 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.1 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 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.

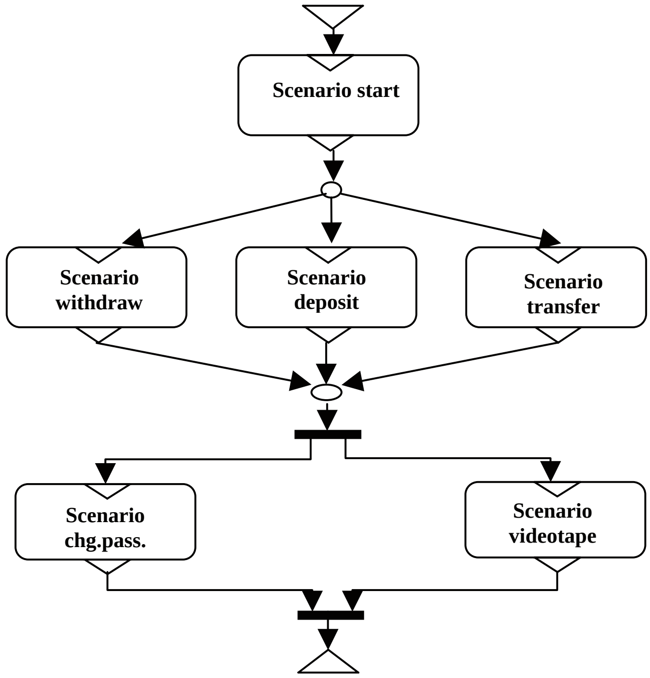

Fig. 1 Dependency diagram for several scenarios of an ATM system

By representing the relationships between various scenarios, we can easily tell what other scenarios would be affected if one scenario were changed. This contributes considerably to the enhancement of traceability.

4. SYNTHESIS OF STATE MACHINES FROM MULTIPLE SCENARIOS

In a scenario, more exactly in its representation as a sequence diagram, there are a number of messages exchanged between objects. Each such message is a tuple: (O_i, O_j, M_{ijk}), where O_i and O_j belong to the set of all objects involved in the system and M_{ijk} depicts the message exchanged between object i and object j. There can be more messages exchanged between the same objects, so k is used to denote these different messages.

Therefore a scenario will be a matrix of tuples including all the messages exchanged between all objects part of that scenario.

For example, if we consider the ATM system, let us assume a scenario (represented in Fig.2) where 4 objects are involved: *user, ATM, consortium and bank.* We will represent these objects as O_1, O_2, O_3 and O_4.

The messages exchanged 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. 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.

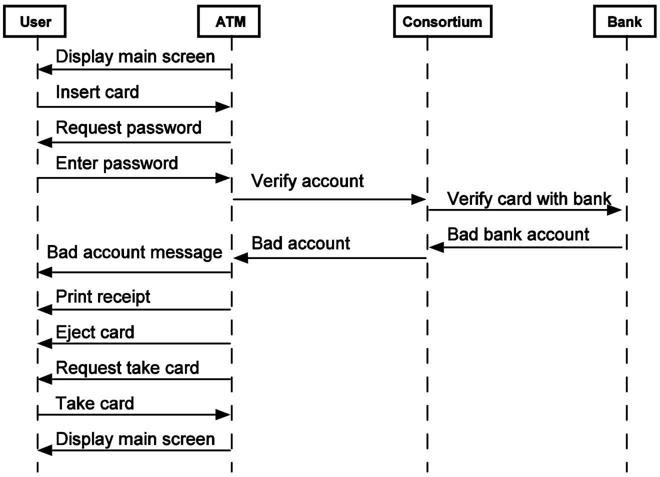

Fig. 2 Sequence diagram (scenario) of an ATM system

Our scenario S_1 will therefore be the following matrix:

$$S_1 = \begin{pmatrix} O_2, O1, M_{211} \\ O_1, O_2, M_{121} \\ O_2, O_1, M_{212} \\ O_1, O_2, M_{122} \\ O_2, O_3, M_{231} \\ O_3, O_4, M_{341} \\ O_4, O_3, M_{431} \\ O_3, O_2, M_{321} \\ O_2, O_1, M_{213} \\ O_2, O_1, M_{214} \\ O_2, O_1, M_{215} \\ O_2, O_1, M_{216} \\ O_1, O_2, M_{123} \\ O_2, O_1, M_{217} \end{pmatrix}$$

There are 4 objects involved in this scenario; from this scenario only, we can synthesize 4 state machine diagrams (one for each object).

In a complete description of this system, there are N scenarios, with a total number of P objects. We will have N matrixes including all the transitions between objects.

The total number of state machine diagrams will be equal to the number of objects in all scenarios. We will therefore have a number of P final state machine diagrams.

Algorithm of synthesis

In order to synthesize the state machine diagrams for all objects, our methodology proposes the following phases:

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

The synthesis of the state machine diagrams involves two steps, *for each object* in the system:

1. creating one initial state machine diagram for each scenario;
2. synthesizing the final state machine diagram from all the state machine diagrams, based on the information in the dependency diagrams.

Creation of initial state machines

The creation of the initial state machines represents the basis for the synthesis of the final state machine. In the following, we are going to succinctly describe how to obtain these state machines.

State machines are the ones that relate events and states. When an event is received, the next state depends on the current state as well as the event. A change of state caused by an event is called a transition. When a transition is triggered, the system leaves its current state, initiates the actions specified for the transition and enters a new state.

A state machine diagram is a graph whose nodes are states and whose directed arcs are transitions (labeled 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 [6]. 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.

Based on [6], we can define the basic rules for generating state machines 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. Before receiving any event, the object is in the default state.

Fig.3 illustrates the state machine diagram corresponding to the ATM object in the scenario given as example in Fig.2. (*Display main screen* is considered to be the default state.)

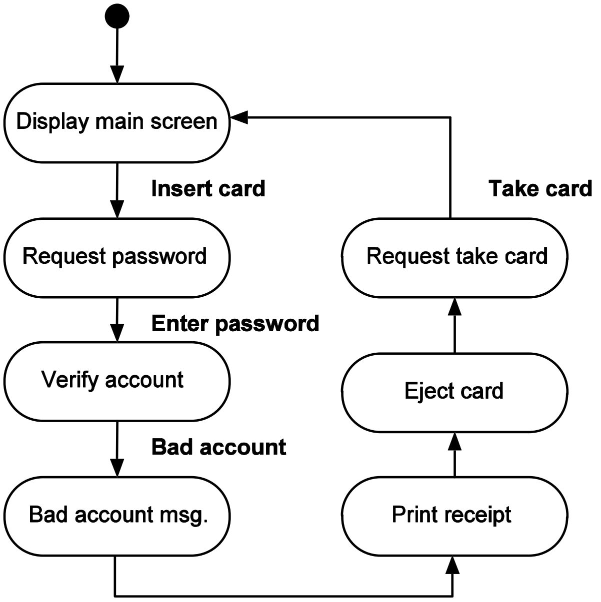

Fig.3. State machine diagram for object ATM

The number of initial state machines for an object O_i will be equal to the number of scenarios in which the object O_i is involved.

Sequentially, the steps of creating initial state machines are the following:

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

Specifically, 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 proceed in the same manner 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,* and *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 at this moment 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, so 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.

Synthesis of final state machines

In order to obtain the final state machines, we need to combine all the initial state machines, making use of the information in the dependency diagrams.

As described before, the dependency diagrams show the possible relationships existing between scenarios. Based on the classification of relationships between scenarios, there are several rules that need to be followed:

- In a succession of two scenarios, the resulting state machine diagram merges the two basic corresponding state machine diagrams.
- If a transition is common to 2 scenarios, it will be taken only once in the final state machine.
- For two scenarios related with a disjunction relationship, their corresponding state machines should be combined with OR.
- If two scenarios are executed at the same time, their corresponding state machines must be combined with AND.
- In the final phase, the state machine diagrams should be refined, with respect to aggregation of states and generalization of states.

Let us consider 3 scenarios of using the ATM: one for withdrawing cash, one for depositing cash and one for transferring money. They are all preceded by a common scenario, that is the scenario where the user inserts the card and password and they are validated by the bank. The dependency diagram given as an example in section 3 shows that the 3 scenarios are related by disjunction. Only one of them can take place at a certain time. (Due to space limitation we will not represent them here.) Based on these scenarios and on the dependency diagram illustrating the relationships between them, we will obtain a state machine diagram as the one in Fig. 4.

The event *OK account* takes the system into one of the states where cash can be withdrawn, cash can be deposited or money can be transferred. The concept of *cluster* is involved [HAR]; we represent a single superstate as a rectangle, and inside it we figure the corresponding states resulting from each scenario. We obtain in this way multilevel state machines. We can cluster, therefore, the 3 states (corresponding to each possible action: withdraw, deposit or transfer) into a new superstate, named "transaction". The semantics of the state "transaction" is the OR (actually, it is exclusive OR) of the 3 states mentioned above.

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

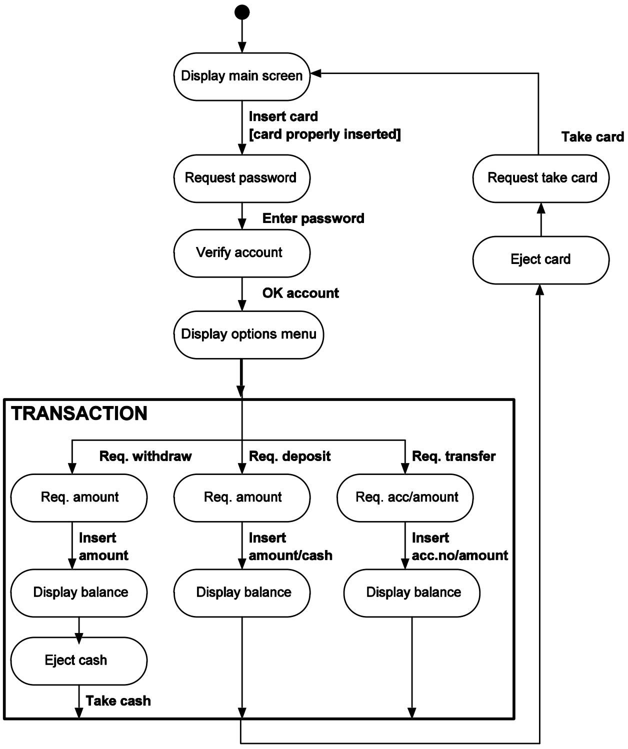

Fig.4. Final state machine diagram for ATM object

Summarizing, if we have a total number N of scenarios, with a total number P of objects, for each of these objects, the number of initial state machines will be less than or equal to N (since each object does not necessarily appear in all scenarios). After the synthesis of the final state machines, one such state machine will exist for each object. Therefore, there will be a total number P of final state machines.

5. DISCUSSION

A complete state machine 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.

Refining the final state machines (consistency between scenarios and state machines)

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

6. RELATED WORK

Several papers deal with the transformation of scenario type models into behaviour models. SCED [8] is a tool for automatic generation of statecharts from single scenarios. In [9], 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 [10] describe an algorithm for model transformation, more precisely an algorithm for transforming collaboration diagrams into state diagrams. Collaboration diagrams describe the interaction among objects, with the focus on space. This means that the links among objects in space are of particular interest and explicitly shown in the diagram. Sequence diagrams (as representation of scenarios) on the other hand, although they also describe how objects interact and communicate with each other, focus on time. Although the two kinds of diagrams are similar (and collectively called 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 [11] 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.

7. CONCLUSIONS

We describe a method of synthesizing state machine diagrams from multiple scenarios, with regard to the relationships between them. We introduce dependency diagrams for showing all the relationships between scenarios. We offer steps and rules for the synthesis of state machines from multiple interrelated scenarios, based on the initial scenarios and on the newly introduced dependency diagrams. Our approach offers complete requirements specifications, an accurate design, as well as an improved traceability.

8. REFERENCES

[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] 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, pp. 231-274.

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

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

[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, pp. 22-34.

[8] 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.

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

[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”, **Proceedings 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] S. Vasilache and J. Tanaka, “Synthesizing statecharts from multiple interrelated scenarios”, **Proceedings of the International Symposium for Future Software Technology ISFST2001**, ZhengZhou, China, 2001, pp. 158-163.

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

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

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

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

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

[19] 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.

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