Formal modelling of use cases with X-machines

Dimitris Dranidis, Kalliopi Tigka, Petros Kefalas

Computer Science Department
CITY LIBERAL STUDIES
Affiliated Institution of the University of Sheffield
Tsimiski 13, 54624 Thessaloniki, Greece
{dranidis,kefalas,tigka}@city.academic.gr

Abstract. Use cases are a popular method for capturing the behavioral requirements of software systems. They are usually written in informal text form describing the interactions between users and the system. Use cases are the central driving artifacts in Unified Process (UP), an agile software methodology. Testing plays a very important role in UP and other agile methodologies, such as Extreme Programming. X-machines is a formal method for the specification of systems. Furthermore, a method for testing systems specified by X-machines exists that generates a complete test case set. This paper proposes the integration of X-machines in the UML use case model, in order to facilitate the generation of a complete test case set for system testing. We present a transformation that semi-automatically transforms use case text into its corresponding X-machine model and we demonstrate the transformation by using the example of an ATM. We also suggest some improvements in the design of X-machine models, such as the use of compound inputs (consisting of interaction functions and data) and a structured representation of the memory, giving an object-oriented flavor, and we discuss the benefits of these improvements.

1 Introduction

Use cases are a method for capturing and documenting the functional requirements of a system. They were introduced by Jacobson [12, 14] and are currently part of UML [21]. However, UML does not support the specification of use cases at the level of scenario descriptions. UML defines only the concept of use case diagrams which collectively illustrate the names of use cases, their users (actors), and their relationships. Several methodologies [14, 13, 20], among them also agile methodologies, such as the Unified Process (UP) [17], suggest that the software development process should be use case driven. In such a process, use cases are not only used for documenting and analyzing the requirements, but they also drive other project activities such as planning, design, and testing.

Testing plays a very important role in all agile methodologies, including UP and Extreme Programming (XP) [2], since it provides a safety net for incremental development and adaptation to change (change of design or change of customer requirements). System testing is based on executing all paths (scenarios) of the use cases. To facilitate and support testing, the use case model needs to be enhanced with a method to describe use case scenarios and their individual steps (i.e. the user interactions and the expected system responses). Ideally, this method should also generate a complete set of test cases for system testing.

Use cases are usually written in plain text. Many authors support that this is usually the best choice, keeping in mind that one of the main purposes of use cases is the communication between customer and developer [14, 5]. However, text is ambiguous, thus leading to different interpretations. Moreover, textual descriptions are prone to mistakes and incompleteness. On the other hand, formal specification methods have exact semantics, thus providing a unique meaning to the description; formal specifications can also be checked for consistency and completeness.

There have been several attempts to formally specify use cases. Back et al. [1] propose the enhancement of use case diagrams with formal documents (contracts) using the refinement calculus. Overgaard and Palmkvist [19] provide an operational semantics for use cases using an object-oriented specification language. In [8], use cases are specified in Abstract State Machine Language, and test cases are generated; this work extends [7] in which use cases are formalized using Z. In [3] the focus is on analyzing use case diagrams, use case texts, and sequence diagrams, to derive functional system test requirements. None of these formalizations deal with the problem of complete test generation.

In this paper, we propose the specification of use cases with X-machines [6, 11], a method with which we can derive a complete set of test cases for system testing. We present a method for transforming use case text into its corresponding X-machine model and we demonstrate the transformation by using the example of an ATM. We also suggest some improvements in the design of X-machine models, such as the use of compound inputs (consisting of interaction functions and data) and a structured representation of the memory, giving an object-oriented flavor, and we discuss the benefits of these improvements.

2 Use Cases

In a use case driven development, the functional specification consists of a set of use cases, describing the behavior of the system from the perspective of its users. Each use case satisfies a user goal. It is initiated by an actor and terminated when the goal is satisfied or failed. Each use case consists of scenarios, representing alternative usages of the system, depending on user choices and system state. A scenario is a sequence of action steps, which are either interactions of the actors with the system, or system responses.

Use cases are written using some textual form. Although there is no standard for writing use cases, there are several guidelines (in form of templates, or writing rules) for writing use cases. The style that we use in this paper follows the guidelines suggested by Cockburn [5].

Each use case consists of a main success scenario and many extensions. The main scenario illustrates the sequence of events when everything goes smoothly. Alternative scenarios (branches) are separately presented in the extensions. Each extension is identified by the step it is extending and a unique letter. An extension consists of the condition that triggers the extension scenario and the steps which are executed in this case. Unless otherwise stated, the flow continues with the repetition of the step that failed and caused the execution of the extension.

The style of writing action steps is also very simple. Actor interactions (or system responses) are expressed using the form: actor/system verb object, e.g. "Customer enters password" (or "System validates password").

In the following example we present a use case from an ATM. The customer wishes to withdraw money by using his card.

Use Case: Withdraw
Actors: Customer

Main success scenario
1. Customer inserts card
2. System validates card
3. Customer enters PIN
4. System validates PIN
5. Customer enters amount
6. System validates amount can be withdrawn
7. System ejects amount
8. System ejects card
9. Customer takes amount and card

Extensions
2a. Invalid card:
  2a1. System notifies user, ejects card, end of use case
4a. Invalid PIN:
  4a1. System notifies user, requests PIN again
  4a2. Customer reenters PIN
4b. Invalid PIN entered 3 times:
  4b1. System retains card, end of use case
6a. Amount exceeds balance:
  6a1. System notifies user, requests amount again
  6a2. Customer reenters amount

2.1 Use case model

Use case texts are not the only documents in the functional specification of a system. They are part of the use case model which consists of three kinds of artifacts:

– A use case diagram which illustrates all the significant use cases and their associations with the actors. This is like a context diagram which shows an overall picture of the usage of the system.
– A textual description for each use case.
– A system sequence diagram for each use case scenario.

System sequence diagrams [18] are special instances of UML sequence diagrams which illustrate the sequence of interactions of the actors with the system. The system is treated as a black box. They add more detail to use case scenarios by imposing the need of definition of system operations and the necessary information passed to the system in terms of parameters (Figure 1).

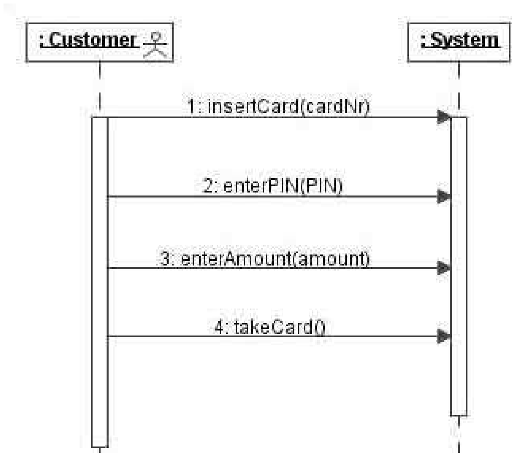

Fig. 1. System sequence diagram of main success scenario of use case Withdraw.

A different system sequence diagram is needed for each different scenario of a use case. The scenarios are not related to each other diagrammatically. To overcome these problems, we propose the addition of X-machines in the use case model. Using X-machines we can: (a) present the sequence of interactions of all scenarios of a use case in the same diagram; (b) present the sequence of interactions of many use cases in the same diagram; and (c) combine the information of UML system sequence diagrams and state diagrams.

3 X-Machines

X-machine is a formal method [6], which is capable of modelling both the data and the control of a system. X-machines employ a diagrammatic approach of modelling the control by extending the expressive power of finite state machines. Transitions between states are no longer performed through simple input symbols but through the application of functions. In contrast to finite state machines, X-machines are capable of modelling non-trivial data structures by employing a memory, which is attached to the X-machine. Functions receive input symbols and memory values, and produce output while modifying the memory values.

3.1 Definition of X-machine

A particular class of X-machines is *stream X-machines* which is defined as a construct as follows [11]: $SXM = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0, T)$ where:

- Σ and Γ is the input and output finite alphabet respectively;
- Q is the finite set of states;
- M is the (possibly) infinite set called memory;
- Φ is the type of the machine SXM, a finite set of partial functions ϕ that map an input and a memory state to an output and a new memory state, $\phi : \Sigma \times M \rightarrow \Gamma \times M$;
- F is the next state partial function that given a state and a function from the type Φ, provides the next state, $F : Q \times \Phi \rightarrow Q$ (F is often described as a transition state diagram);
- q_0 and m_0 are the initial state and memory respectively; and
- T is the set of terminal states.

The sequence of transitions (path) caused by the stream of input symbols is called a computation. The computation halts when all input symbols are consumed. The result of a computation is the sequence of outputs produced by this path.

Stream X-machines can be thought to apply in similar cases where Statecharts [9] and other similar notations do. However, apart from being formal as well as proved to possess the computational power of Turing machines [11], X-machines have other significant advantages since they offer a strategy to test the implementation against the model and a strategy to verify the model against user requirements [16]. In principle, X-machines are considered a generalization of models written in similar formalisms. Concepts devised and findings proven for X-machines form a solid theoretical framework, which can be adapted to other, more tool-oriented methods, such as Statecharts.

3.2 Notation for memory and inputs

In this paper we introduce a slightly different notation from the one used in the X-machine literature [15]. Firstly, we use compound inputs instead of simple inputs. Each compound input is a pair of an interaction function and an input value (e.g. `enterAmount.x`). This notation simplifies the definition of input sets. Secondly, memory is structured as sets of classes, consisting of sets of objects. This object-flavored representation of memory simplifies the access to the complex structure of the memory, by allowing the use of the dot notation for accessing objects and attribute values (e.g. `atm.currentcard.id`). It has to be emphasized that this different notation does not retract anything from the formal notation of X-machines. Both enhancements are purely syntactical and can be transformed to the usual formal notation.

Modelling the memory The memory is organized as sets of objects (classes) with their associated attributes. This modelling allows the transformation of a class diagram into its memory representation.

A memory instance is described as a set of classes. Each class is a pair: the set of objects belonging to this class and a set of attribute mappings. Each attribute mapping maps to an object the value of its corresponding attribute. Attribute values are objects themselves. Object are identified by their names.

The type of the memory is the set of all memory instances preserving the structural properties of objects. If OBJECT is the set of all objects in the universe then the memory type is a set of pairs: disjoint powersets of OBJECT and mappings for the attributes.

Modelling inputs as pairs of interaction functions and data The inputs trigger the execution of processing functions and cause state transitions. We define inputs as pairs of interaction functions and their arguments. This choice seems more natural when a software system is implemented in which each input is handled by a function. Assume that we have the case of a system in which the user has to enter a number to be used as the amount in a transaction. Probably numbers are entered in other occasions in the same program for different purposes. To distinguish this specific numerical input, we write it as: `enterAmount.x` where `x` is the number entered and `enterAmount` the interaction function called. The dot notation can also be met in other formalisms such as CSP [10] in which it is used for compound objects: channels and data.

This is simply a design mechanism which makes the design of systems simpler. It does not impose any changes to the theoretical model of X-machines. The set of function-parameter inputs can be flattened to a set of simple inputs by prefixing its input with the function symbol, as one would do in the first case we described above.

4 From Use Cases to X-Machines

To transform a use case to the corresponding X-machine, we first determine the states and the transitions, then the memory structure, the input and output sets, and finally we define the transition functions.

4.1 States and Transitions

States and transitions are derived by examining the steps of both the main success scenario and the extensions:

– There is an initial state corresponding to that state in which the actor triggers the use case. Although, this state usually coincides with the final state of the state diagram, we choose to notate the final state as a different node in the state machine, to emphasize the termination of the use case.
– Each user interaction introduces a new transition leading to a new state in the X-machine state diagram. For instance, the user interaction `insertCard(cardId)` introduces a new transition labeled with the processing function: `enterValidCard` and a new state: `waiting PIN`.
– Each extension introduces a new transition from the same starting state of the previous user interaction. The transition leads to a new state, if an interaction follows in the extension steps. If there is no user interaction in the extensions, unless otherwise stated, the transition loops back to the same state. For instance, the extension "2a. Invalid card" introduces a new transition labeled with the processing function: `enterInvalidCard` to the state: `card ejected`.

Each user interaction and all its subsequent system functions until the next user interaction are modelled by a single processing function. So a processing function does not only model the user interaction but also all the processing that guarantees that all subsequent system functions are correctly executed. The processing function is not always triggered by the interaction (which provides the necessary input) but it contains, as guard expressions, all the conditions that have to be satisfied for the transition to occur.

The state transition diagram is depicted in Figure 2.

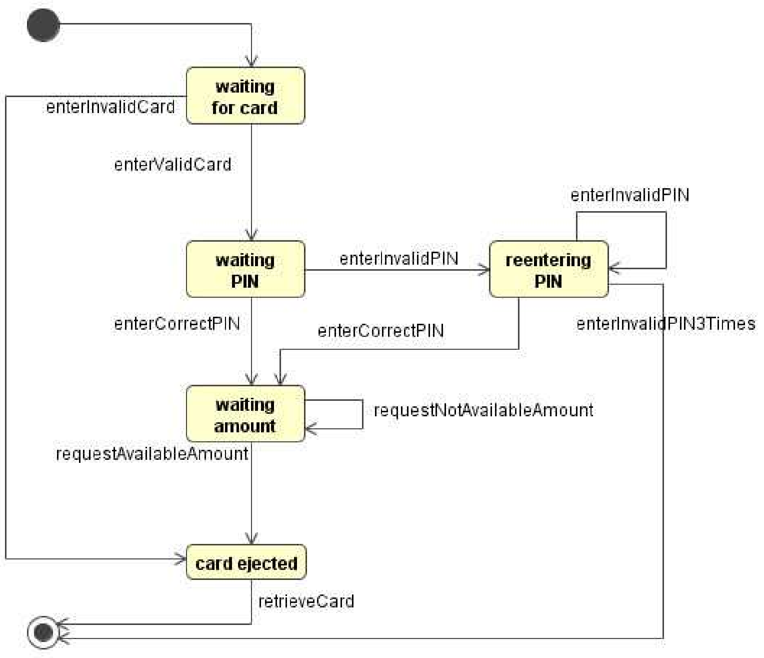

Fig. 2. X-machine state transition diagram.

4.2 Memory

The memory structure and contents cannot be directly derived from the use case text. The object-flavored memory structure that we propose is easily derived from the domain model which is usually represented as a static class diagram. A domain model is constructed either after or before writing of use case texts. It presents the concepts of the problem domain and their relationships. Concepts are represented as classes with attributes and relationships as associations between classes. A partial class diagram for the ATM is shown in Figure 3.

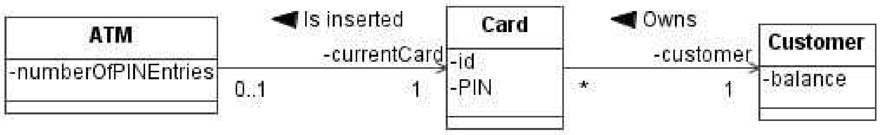

Fig. 3. Class diagram (Domain model) of the ATM.

Since memory has to be filled with specific values, we define sample objects for operating the machine. A small number of objects is used in order to exercise different scenarios of the use case. The memory instance below could be the initial memory state of our X-machine.

```
M = { Customer, Card, ATM }

ATM = ( {atm}, {currentcard, numberOfPINentries} )
Customer = ( { customer1, customer2 }, {balance} )
Card = ( { card1, card2, card3 }, {id, customer, PIN} )

where
    atm.currentCard = null, atm.numberOfPINentries = 0,
    customer1.balance = 100, customer2.balance = 100,
    card1.id = 97, card1.customer = customer1, card1.PIN = 3234,
    card2.id = 98, card2.customer = customer1, card2.PIN = 3278,
    card3.id = 99, card3.customer = customer2, card3.PIN = 3137.
```

In the above example the memory consists of three classes of objects: `ATM`, `Customer`, and `Card`. There are two objects of class `Customer`: `customer1` and `customer2`. The object `customer1` has the value `100` for its attribute `balance`. The object `card1` of class `Card` holds the values: `97`, `customer1`, and `3234`, where each one corresponds to the value of the attributes: `id`, `customer`, and `PIN`. The `atm` object of class `ATM` stores the current card inserted in the system in its attribute: `currentCard`. The dot notation `customer1.balance` is interpreted as `balance(customer1)`. The dot is simply a convenience for using the mapping $\texttt{balance} = \{(\texttt{customer1}, \texttt{100}), (\texttt{customer2}, \texttt{100})\}$. To refer to the balance of the second customer in our specification we use the syntax: `Customer.customer1.balance`. To refer to the balance of customer owning the card inserted in the atm we write: `ATM.atm.currentCard.customer.balance`. The dot binds left to right.

Notice the use of `null` as a void object. The value of attribute `currentCard` of the object `atm` is `null`, since no card is initially inserted in the system. We assume that the memory contains a class $\texttt{NULL} = (\{\texttt{null}\}, \{\})$, which can be used for any type, and other primitive types, such as `Integer`.

4.3 Input and Output Sets

Inputs are directly derived from the system sequence diagram. For the ATM the input set is:

$$\texttt{insertCard.Integer} \cup \texttt{enterPIN.Integer} \cup \texttt{enterAmount.Integer} \cup \texttt{takeCard.NULL}$$

where, for instance $\texttt{insertCard.Integer} = \{\texttt{insertCard.x} \mid \texttt{x} \in \texttt{Integer}\}$.

As output set we define the set of messages that the ATM would display after each transition: “Enter PIN”, “Enter amount”, “Take card and amount”, “Invalid card, take card”, “Wrong PIN”, “Card retained”, “Amount not available”

4.4 Processing Functions

For each processing function we have to define the input and the memory state that trigger the function, the output that the function produces and the memory update. Below we provide definitions for two processing functions of the ATM example. The rest of the functions have similar definitions and are omitted due to space limitation.

```
enterCorrectPIN ( enterPIN.x , mem ) = ( "Enter amount", mem)
    if x == mem.ATM.atm.currentCard.PIN

enterInvalidPIN ( enterPIN.x , mem ) = ( "Wrong PIN", mem’)
    if x !=  mem.ATM.atm.currentCard.PIN and
       mem.ATM.atm.numberOfPINentries < 2
    where mem’ = update mem with (
       ATM.atm.numberOfPINentries = ATM.atm.numberOfPINentries + 1
    )
```

Notice that the guard conditions of the two processing functions are disjoint. This is a design constraint for avoiding non-determinism in the final description. A specific input may trigger only one transition at each state.

5 Generation of Test Cases

There exists a testing strategy based on X-machines [11], which is a generalization of W-method [4] that, under certain assumptions [11], it is proved to find all faults in the implementation. In addition, the method requires that the X-machine models satisfy the design for test conditions, i.e. they are complete with respect to memory (any basic function will be able to process all memory values) and output distinguishable (any two different processing functions will produce different outputs on each memory/input pair). In case that the X-machine is not complete, as in the ATM example presented in the previous section, then it is straightforward to introduce additional input symbols such as to make processing functions complete.

When the above requirements are met, the W-method may be employed to produce the k-test set X, where k is the difference of the number of states of the X-machine model and the implementation. The test set X consists of processing functions for the associated automaton, and it is given by the formula $X = S(\Phi^{k+1} \cup \Phi^k \cup \ldots \cup \Phi \cup \{\epsilon\})W$, where W is a characterization set and S a state cover. Informally, a characterization set is a set of processing functions for which any two distinct states of the machine are distinguishable and a state cover is a set of processing functions such that all states are reachable from the initial state. The W and S sets in the ATM X-machine are:

$$W = \{\texttt{enterValidCard}, \texttt{enterCorrectPin}, \texttt{retrieveCard}, \texttt{enterInvalidPin3Times}\}$$

$$\begin{aligned} S = \{ & \langle\epsilon\rangle, \langle\texttt{enterValidCard}\rangle, \langle\texttt{enterValidCard}, \texttt{enterInvalidPin}\rangle, \\ & \langle\texttt{enterValidCard}, \texttt{enterCorrectPin}\rangle, \langle\texttt{enterInvalidCard}\rangle, \\ & \langle\texttt{enterValidCard}, \texttt{enterCorrectPin}, \texttt{requestAvailableAmount}, \texttt{retrieveCard}\rangle\} \end{aligned}$$

The derived test set X, e.g. for $k = 0$, is the following:

$$\begin{aligned} X = \{ & \langle enterInvalidCard, retrieveCard\rangle, \\ & \langle enterValidCard, enterInvalidPin, enterCorrectPin\rangle, \\ & \langle enterValidCard, enterCorrectPin, requestAvailableAmount, retrieveCard\rangle, \\ & \langle enterInvalidCard, enterValidCard, enterCorrectPin\rangle, \ldots\} \end{aligned}$$

The fundamental test function is defined recursively, and converts these sequences into sequences of inputs of the X-machine as described in [11]. The corresponding test-set containing sequences of inputs for the ATM is the following:

$$\begin{aligned} TS = \{ & \langle invalidCardId, \texttt{takeCard}\rangle, \\ & \langle validCardId, invalidPIN, validPIN\rangle, \\ & \langle validCardId, validPIN, amount, \texttt{takeCard}\rangle, \\ & \langle invalidCardId, validCardId, validPIN\rangle, \ldots\} \end{aligned}$$

where *invalidCardId*, *validCardId*, *invalidPIN*, *validPIN*, and *amount* are appropriate input values, as for instance `insertCard.50`, `insertCard.99`, `enterPIN.1000`, `enterPIN.3137`, and `enterAmount.10` respectively. The test-set so produced is proved to find all faults in the ATM implementation. The testing process can therefore be performed automatically by checking whether the output sequences produced by the ATM implementation are identical with the ones expected from the ATM model.

6 Discussion

The transformation described in this paper is performed manually. We believe that the transformation can be partially automated. Full automatization seems rather difficult due to several issues discussed below.

The state transition diagram and the outputs are easily determined by analyzing the use case. No design decisions are incorporated in this diagram. The use case (and the corresponding modelled system) is described as a labelled transition system. This part of the process can be fully automated.

The inputs cannot be automatically extracted from the use case description. For the definition of the inputs, a name for the interaction function and the parameters must be determined. A system sequence diagram is usually constructed to provide this information.

The use case description cannot be used for automatically deriving the memory. The memory is constructed with the aid of the domain model. The process of transforming the domain model to an X-machine memory presentation can be automated. This is possible due to the object-flavored memory organization that we have introduced.

Finally, the transition functions have to be specified. The following things have to be considered: (a) the inputs to which the function respond, (b) the guard condition, (c) the function's output, and (d) the memory update. (a) and (c) are already determined in previous steps of the process. It remains to provide specifications for (b) and (c).This is the part of the process, which cannot be automated at all.

The fact that the process cannot be fully automated is an indication that the X-machine specification is more expressive than the artifacts of the use case model. The predicates of the guard expressions formally specify necessary preconditions of the processing functions. Furthermore, the update of the memory formally specifies the postconditions of each processing function. These specifications result to a more rigorous model, which allows the verification of the implementation.

7 Conclusion

In this paper we have presented an enhancement of the use case model with a formal method. X-machines are used for formalizing all scenarios of use cases. Apart from the increased expressiveness of the produced model, the most important benefit of the formalization is the ability of producing a complete test set for system testing. This last benefit is very important in agile methodologies such as the UP or XP. Use cases are functional in their nature. To formalize them using X-machines, we derived the structure of the memory from the domain model. By doing so, we also validated the domain model, since an incomplete domain model would not allow the execution of X-machines. Thus our formalization also bridges the functional world of use cases with the object-oriented world of class diagrams used in the domain model.

Further research will focus on the automatization of the transformation process. This would allow fast adaptation of the formal models to changing requirements, another important aspect of agile methodologies.

References

1. R. Back, L. Petre, and I. Paltor. Formalising UML use cases in the refinement calculus. Technical Report TUCS-TR-279, Turku Centre for Computer Science, Turku, Finland, 1999.
2. K. Beck. *Extreme programming explained: embrace change.* Addison-Wesley, 2000.

3. L. Briand and Y. Labiche. A UML-based approach to system testing. In *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference*, 2001.
4. T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.
5. A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2000.
6. S. Eilenberg. Automata, languages and machines. *Academic Press, New York*, A, 1994.
7. W. Grieskamp and M. Lepper. Using use cases in executable Z. In *ICFEM 2000 - IEEE Conference on Formal Engineering Methods*, pages 111–120, 2000.
8. W. Grieskamp, M. Lepper, and W. Schulte. Testable use cases in the abstract state machine language. In *APAQS'01 - Second Asia-Pacific Conference on Quality Software*, 2001.
9. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
10. C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
11. M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution.* Springer Verlag, Berlin, 1998.
12. I. Jacobson. Object oriented development in an industrial environment. In *Proc. of the OOPSLA-87: Conference on Object-Oriented Programming Systems*, pages 183–191, Languages and Applications, Orlando, FL, 1987.
13. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.
14. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Wokingham, UK, 1992.
15. E. Kapeti and P. Kefalas. A design language and tool for x-machine specification. In *Proceedings of the 7th Panhellenic Conference on Information Techology, Greek Computer Society, Ioannina*, 1999.
16. P. Kefalas, M. Holcombe, G. Eleftherakis, and M. Gheorghe. A formal method for the development of agent-based systems. In V. Plekhanova, editor, *Intelligent Agent Software Engineering*, pages 68–98. Idea Group Publishing Hershey, 2003.
17. P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, 2nd edition, 2000.
18. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall, 2nd edition, 2001.
19. G. Övergaard and K. Palmkvist. A formal approach to use cases and their relationships. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 309–317, 1998.
20. D. Rosenberg and K. Scott. *Use case driven object modeling with UML: a practical approach.* Addison-Wesley, 1999.
21. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.