Published in:
Proceedings of Second IEEE International Symposium on Requirements Engineering (RE'95), York, UK, March 1995.

Improving the Use Case Driven Approach to Requirements Engineering [*]

Björn Regnell, Kristofer Kimbler, Anders Wesslén

Dept. of Communication Systems, Lund University, Sweden
(bjornr, chris, wesslen)@tts.lth.se

Abstract

This paper presents the idea of Usage Oriented Requirements Engineering, an extension of Use Case Driven Analysis. The main objective is to achieve a requirements engineering process resulting in a model which captures both functional requirements and system usage aspects in a comprehensive manner. The paper presents the basic concepts and the process of Usage Oriented Requirements Engineering, and the Synthesized Usage Model resulting from this process. The role of this model in system development, and its potential applications are also discussed.

1. Introduction

When dealing with complex systems, it does not seem feasible to go directly, in one step, from an informal requirements description provided by the customer to a formal requirements specification. Too rapid formalization of requirements may have several negative consequences, such as a substantial semantic gap between the requirements description and the requirements specification or incompleteness of the latter. It is also very difficult to produce a formal specification without a deep understanding of what the customer and the end users expect from the system, and how they intend to employ it in practice. This kind of information is rarely provided at the outset of system development.

This paper presents Usage Oriented Requirements Engineering (UORE) which tries to address the above issues in a structured and systematic way. The concept of UORE originates from Use Case Driven Analysis (UCDA), a key contribution of the Objectory method [1].

Our objective is to improve the original UCDA by extending it with a *synthesis phase* where separate use cases are integrated into a Synthesized Usage Model (SUM). This model captures both functional requirements and system usage aspects. To facilitate this integration, UORE introduces a formal, graphical representation of use cases and abstraction mechanisms for representing user and system actions.

2. Use Case Driven Analysis

This section presents and discusses UCDA as defined in the Objectory method [1]. The basic concepts of UCDA are actors and use cases. An *actor* is a specific role played by a system user, and represents a category of users that demonstrate similar behaviour when using the system. By users we mean both human beings, and other external systems or devices communicating with the system. An actor is regarded as a class, and users as instances of this class. One user may appear as several instances of different actors depending on the context.

A *use case* is a system usage scenario characteristic of a specific actor. During the analysis we try to identify and describe a number of typical use cases for every actor. Use cases are expressed in natural language with terms from the problem domain. The descriptions of actors and use cases form the Use Case Model (UCM).

Advantages. UCDA helps to cope with the complexity of the requirements analysis process. By identifying and then independently analysing different use cases we may focus on one, narrow aspect of the system usage at a time.

Since the idea of UCDA is simple, and the use case descriptions are based on natural concepts that can be found in the problem domain, the customers and the end users can actively participate in requirements analysis. In consequence, the developers can learn about the potential users, their actual needs, and their typical behaviour.

Disadvantages. The lack of synthesis is probably the main drawback of UCDA. The Use Case Model that we get from UCDA is just a loose collection of use cases. In the subsequent phases of Objectory, these use cases are directly used to create the so-called Analysis Model. This model describes the structure of the system and is a step towards design. What we really would like to get from requirements analysis, is a model which captures the functional requirements and system usage, without any design aspects.

[*] This work is supported by National Board for Industrial and Technical Development, (NUTEK), Sweden, Reference Dnr: 93-2850.

Abbreviations

AIO	Abstract Interface Object
AUS	Abstract Usage Scenario
SUM	Synthesized Usage Model
UCDA	Use Case Driven Analysis
UCM	Use Case Model
UCS	Use Case Specification
UORE	Usage Oriented Requirements Engineering

Although use cases are perfect material for creating test cases, the UCM resulting from UCDA cannot be used for automatic generation of test cases. This limits its applicability as a reference model for validation and verification.

There are also several problems with the interpretation of the actor and use case concepts, as defined in Objectory. No clear definition of the semantics of use cases, and no consistent guidelines on how the use cases should be described are provided. It is not clear what kind of events we should concentrate on while describing use cases; external stimuli-responses only, or internal system activities as well. Objectory treats use cases as classes with inheritance-like relations, but, at the same time, they are seen as sequences of events. Object-orientation purists tend to treat everything as objects, but here we find the class interpretation rather artificial and confusing.

Objectory associates every use case with a specific actor, but, at the same time, allows use case descriptions in which several actors are involved. Moreover, an actor is defined as a specific role played by a user. This means that, in extreme, one physical user can appear as different actors in a single use case. These uncertainties leave too much room for free interpretation of the actor and use case concepts, and may cause a lot of confusion.

The number of use cases may be very large in cases of complex systems. Since produced independently, there might be inconsistencies between use case descriptions. Moreover, use cases might be contradictory, as they express goals of different actors. Objectory does not offer support for resolving such problems.

A specific use case can not occur in every situation. What we need for each use case is a specification of the context in which it can be triggered and successfully accomplished. This issue is not addressed by UCDA.

In general, UCDA, as defined in Objectory, does not fully address the following issues:

- Use cases are not independent. They may overlap, occur simultaneously, or influence each other.
- Use cases occur under specific conditions. They have invocation and termination contexts.
- The level of abstraction of use cases and their length are matters of arbitrary choice.
- The use cases can, in practice, guarantee only partial coverage of all possible system usage scenarios.

3. Usage Oriented Requirements Engineering

The proposed UORE process aims at removing some of the weaknesses of UCDA stated in the previous section. UCDA is extended with a synthesis phase, where use cases are formalized and integrated into a Synthesized Usage Model. The SUM captures functional requirements and system usage in a more formal way than the UCM.

The SUM is intended to be a part of requirements specification, and a reference model for validation and verification. The SUM captures the following related aspects:

- Categories of system users and their objectives
- Domain objects, their attributes, and operations
- Stimuli and responses of user-system communication
- User and system actions, their possible combinations and usage contexts
- Scenarios of system usage, their flows of events, and trigger conditions.

The process of UORE consists of two phases, *analysis* and *synthesis*, as shown in fig. 1. The analysis phase has an informal requirements description as input, and produces the use case model containing descriptions of actors and use cases. This model, in turn, is used as input to the synthesis phase which formalizes the use cases, integrates them, and creates the synthesized usage model.

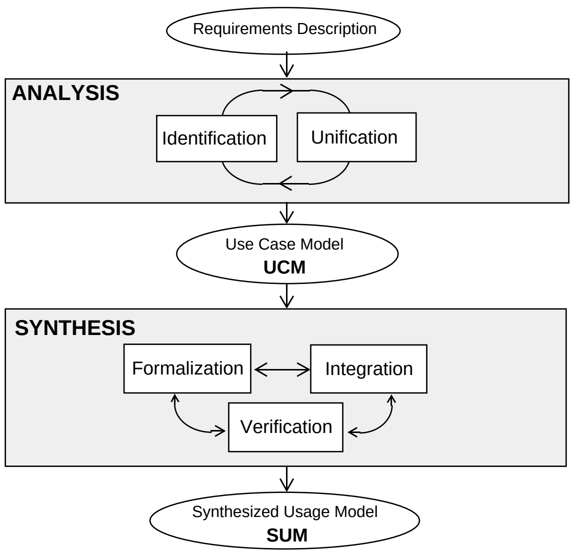

FIGURE 1. The process of UORE

3.1 Analysis phase

The analysis phase of UORE resembles the original UCDA, and consists of two interrelated activities:

1. Identification of use cases and actors
2. Unification of terminology.

The first activity aims at finding and describing actors and use cases. The second activity unifies the terminology of these descriptions. For this purpose the problem domain objects and their attributes are identified and described in a *data dictionary.* The focus is on entities manipulated by the actors, externally observable system operations, and elements of the user interface.

The unification of terminology is important, especially as different use cases may be described by separate persons or groups. The terminology is gradually extended and revised as more and more use cases are identified. The unified terminology is enforced by inspections. The two activities of the analysis phase are performed iteratively.

To illustrate UORE we will use a well-known example of an Automated Teller Machine (ATM) [2]. ATM offers basically two services: *cash withdrawal* and *account control.* In fig. 2 we show examples of actors and use cases.

Actors:
ATM customer – uses the ATM to withdraw cash or control the account balance.
ATM supervisor – supervises and maintains the operation of the ATM.
ATM database – the external system maintaining account information.

1. *Withdraw Cash, normal case*
Actor: *"ATM customer"*

1.IC	**Invocation Conditions:**
1.IC.1	The system is ready for *transactions*.
1.FC	**Flow Conditions:**
1.FC.1	The user's **card** *is valid*.
1.FC.2	The user enters a *valid* **code**.
1.FC.3	The user enters a *valid* **amount**.
1.FC.4	The machine has the *required amount of* **cash**.
1.FE	**Flow of Events:**
1.FE.1	The user inserts the **card**.
1.FE.2	The system *checks if the* **card** *is valid*.
1.FE.3	A *prompt for the* **code** is given.
1.FE.4	The user enters the **code**.
1.FE.5	The system *checks if the* **code** *is valid*.
1.FE.6	A *prompt "enter amount or select balance"* is given.
1.FE.7	The user enters the *amount*.
1.FE.8	The system *checks if the amount is valid*.
1.FE.9	The system *collects the* **cash**.
1.FE.10	The **cash** *is ejected*.
1.FE.11	A *prompt "take cash"* is given.
1.FE.12	The user takes the **cash**.
1.FE.13	The **card** *is ejected*.
1.FE.14	A *prompt "take card"* is given.
1.FE.15	The user takes the **card**.
1.FE.16	The system *collects* **receipt** *information*.
1.FE.17	The **receipt** *is printed*.
1.FE.18	A *prompt "take receipt"* is given.
1.FE.19	The user takes the **receipt**.
1.TC	**Termination condition:**
1.TC.1	The system is ready for *transactions*.

2. Withdraw Cash, amount invalid
Actor: *"ATM customer"*

2.IC	**Invocation Conditions:**
2.IC.1	Same as 1.IC.1.
2.FC	**Flow Conditions:**
2.FC.1	Same as 1.FC.1 - 1.FC.2.
2.FC.2	The user enters an *invalid* **amount**.
2.FE	**Flow of Events:**
2.FE.1	Same as 1.FE.1 - 1.FE.8
2.FE.2	The *"invalid amount" message* is given.
2.FE.3	A *prompt for "retry"* is given.
2.FE.4	The user *aborts the transaction.*
2.FE.5	Same as 1.FE.13 - 1.FE.15
2.TC	**Termination condition:**
2.TC.1	Same as 1.TC.1.

3. Account Control, normal case
Actor: *"ATM customer"*

3.IC	**Invocation Conditions:**
3.IC.1	Same as 1.IC.1.
3.FC	**Flow Conditions:**
3.FC.1	Same as 1.FC.1 - 1.FC.2.
3.FE	**Flow of Events:**
3.FE.1	Same as 1.FE.1 - 1.FE.6.
3.FE.2	The user *selects "balance"*.
3.FE.3	The system *collects balance information*.
3.FE.4	The *balance is displayed*.
3.FE.5	Same as 1.FE.13 - 1.FE.19
3.TC	**Termination condition:**
3.TC.1	Same as 1.TC.1.

Problem domain objects in **bold** face.
Defined and unified terminology in *italics*.

FIGURE 2. Use case description examples

Differences. As mentioned above, the analysis phase of UORE resembles the Objectory version of UCDA. There are, however, a number of issues that make our approach different:

- Changed semantics of actors and use cases
- Identification of use case contexts
- Strict application of the single-actor view
- Explicit unification of terminology
- Structured description of use cases.

In UORE, an actor represents a user (a person or an external system) that belongs to a set of users with common behaviour and goals. An UORE actor does not necessarily model a *single* role played by a user, as in Objectory. In our opinion, the single-role semantics of actors may lead to use cases which address too narrow aspects of system usage. This, in turn, disables analysis of how different system operations interact. (Some systems may allow a user to play multiple roles at the same time.)

Unlike Objectory, which treats the use cases as classes, we regard them just as examples of system usage. We consider use cases as "experimental material" which will be further investigated in the synthesis phase.

In UORE, each use case describes the system behaviour, as seen by one actor only. This *single-actor-view* approach makes the use case concept simpler. We assume that the actor involved in a use case communicates with other actors through the system. No situations with direct actor-to-actor communication are modelled. In other words, the narration of the use cases distinguishes only between the actor and the "rest". If a system usage scenario involves several actors, this scenario should be modelled by several use cases, one for each involved actor. This provides a clear criterion for constructing use case descriptions and reduces their complexity. To conclude, we can say that the UORE principle for use case definition is: "*multiple roles, yes; multiple actors, no*".

In UORE, the description of each use case contains a list of conditions defining a context in which the specific flow of events of the use case can occur. The *invocation conditions* and *termination conditions* define the system state before and after the use case, while the *flow conditions* state the assumptions about the user and system behaviour during the use case. A flow condition is not necessarily true at the invocation of the use case, but it becomes true at some point in the use case. A flow condition is thus a temporal assertion that implicitly refers to a "future" point in the flow of events of the use case. These different conditions are an important aid in the synthesis phase for finding relations between use cases.

In order to avoid some typical problems with natural language descriptions, all the use cases should use the same terminology and format. The terminology of these descriptions is unified across different use cases, as discussed above. The examples in fig. 2 show a possible structure of use case descriptions. A systematic numbering of events supports traceability within and between the models of the analysis and synthesis phases. Furthermore, when describing a use case, we can use such numbers to refer to identical conditions and sequences of events in other use cases, in order to make the description shorter.

3.2 Synthesis phase

The synthesis phase formalizes the use cases, integrates them, and creates the Synthesized Usage Model. The synthesis phase consists of three activities:

1. Formalization of use cases
2. Integration of use cases
3. Verification.

These three activities are carried out in an iterative manner, until an agreement upon the correctness and completeness of the SUM is reached. In the following sections we will describe each activity and the concepts they use.

3.3 Formalization activity

The formalization activity aims at producing a formal Use Case Specification (UCS) for each use case identified in the analysis phase. The product of this activity is a collection of UCS's, represented in the formal, graphic language of message sequence charts (an extension of [3]). Each UCS expresses the temporal ordering of user *stimuli*, system *responses*, and *atomic operations.*

The formalization activity has the following steps:

1. Identification of abstract interface objects
2. Identification of atomic operations
3. Creation of one UCS for every use case.

The concepts used in the formal representation of user-system communication, and the steps necessary in the creation of a UCS are explained below.

Abstract interface objects. The user never communicates directly with a software system. Some sort of interface is always involved in this communication. The interface transforms the user's *stimuli* into *messages* (software events) and, messages from the system into *responses* comprehensible to the user. This transformation is not necessarily straightforward. The three basic elements of user-system communication; the *user*, the *interface* and the *system*, are inherently parts of system usage, consequently they can be found in use case descriptions.

The entities that form the nature of user-system communication will be called Abstract Interface Objects (AIO). They are abstract in the sense that they do not necessarily represent concrete interface objects. Instead, they model responsibilities (see [4]) that can be mapped to one or more real interface objects. The intention is to avoid any design decisions at this stage.

Identification of abstract interface objects is achieved by examining all the use cases and the problem domain terminology, and searching for entities that take part in the actor-system communication. An AIO is characterized by its sets of stimuli, responses, messages, and states.

Atomic operations. On a conceptual level we can describe the elements of the system's capabilities by atomic operations. They are operations performed by the system, and have effect on the users. A system operation is atomic from an actor's point of view, if it does not require any communication with this actor during its execution. However, other actors may see the same operation as a combination of other atomic operations and communication protocols. For example, the operation *card validation* is atomic from the *ATM Customer* actor's point of view, although from the *ATM Database* actor's point of view it is a sequence of operations and communications.

The atomic operations are identified from the use cases by focusing on system operations that do not require interaction with the actor involved in the use case. Every system action is described and given a unique name to be used uniformly in all use case specifications. We will not elaborate here on the specification of atomic operations.

Formal use case specification. The formalization activity produces formal use case specifications. After identifying all abstract interface objects and atomic operations, we transform the flow of events of every use case into a UCS that models the temporal relations between AIO stimuli/responses/states and atomic operations.

We illustrate the notation of UCS by our ATM example. The UCS corresponding to the use case "withdraw cash, normal case" is shown in fig. 3. The left-most time axis of fig. 3 represents the specified actor. The right-most time axis represents the system. Between the actor and the system we have the different AIO's involved in this use case. The AIO states are drawn as diamonds on the AIO time axis, and the atomic operations are drawn as boxes on the system's time axis.

3.4 Integration activity

The integration activity aims at merging different use case specifications and producing a Synthesized Usage Model. The SUM consists of a collection of *usage views*, one for each actor. The integration activity consists of the following three steps:

1. Identification of user and system actions
2. Creation of abstract usage scenarios
3. Integration of abstract usage scenarios.

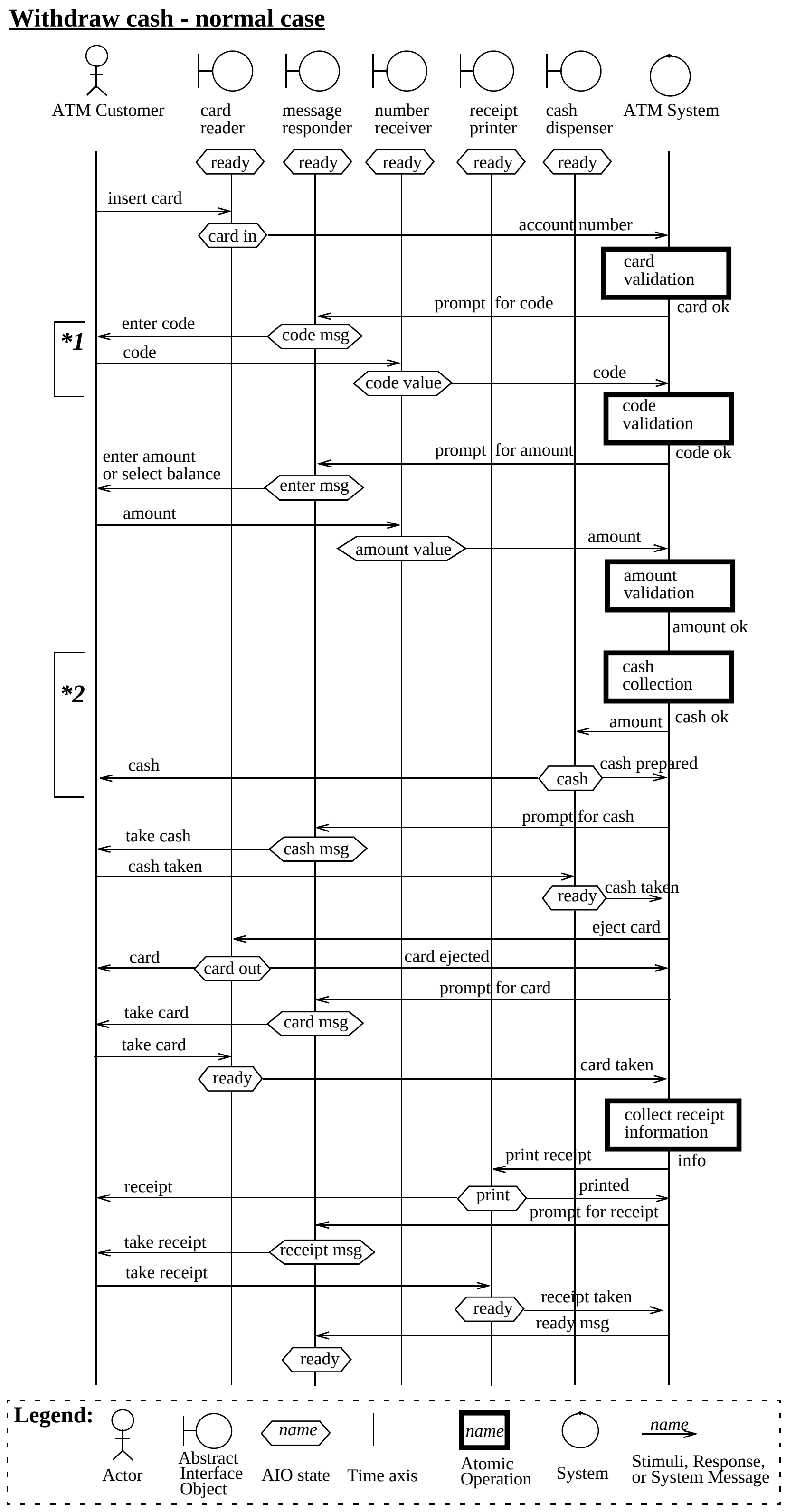

FIGURE 3. Use Case Specification

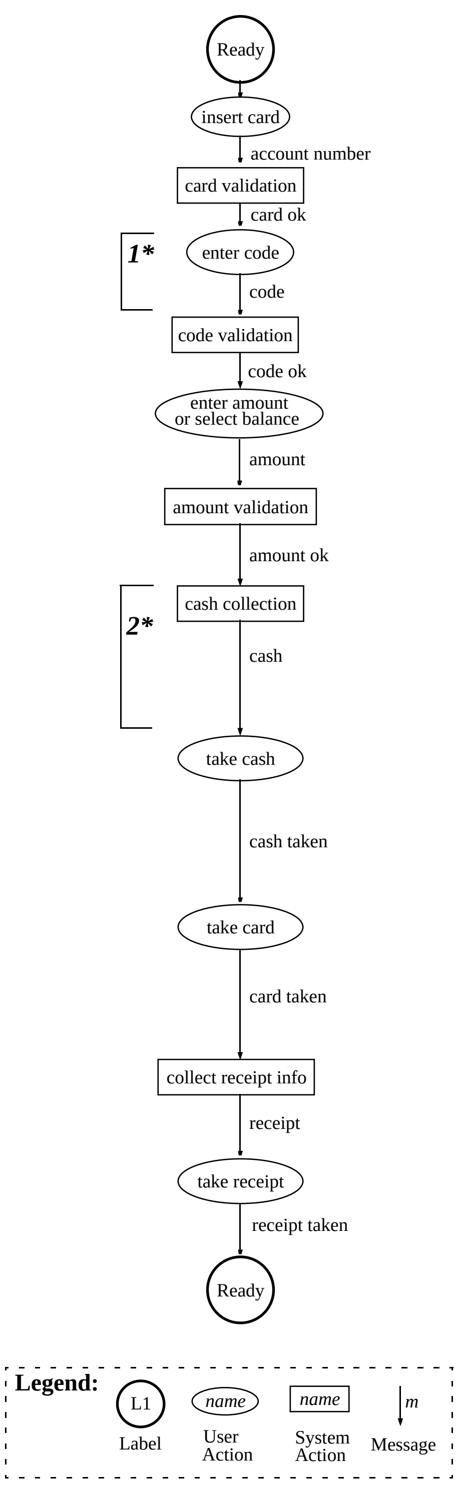

FIGURE 4. Abstract Usage Scenario

Note: For the example in 3.4, two UCS parts are marked **1* and **2*, and correspond to the actions *1** and *2**.

User and system actions. In a use case, the control shifts between the user and the system. When we formally represent system usage we would like to have an abstraction mechanism that conceals the detailed protocol of the interaction during the user-controlled parts and the system-controlled parts of a use case. We use the terms *user actions* for protocols where the user is in control, and *system actions* for protocols where the system is in control.

The first step of the integration activity aims at extracting such abstract protocols. Hence, in several UCS's we can identify actions such as "enter code" and "cash collection", which form a demarcated protocol with a sequence of related events, resulting in a single message. All such UCS parts are uniformly defined with a name and description.

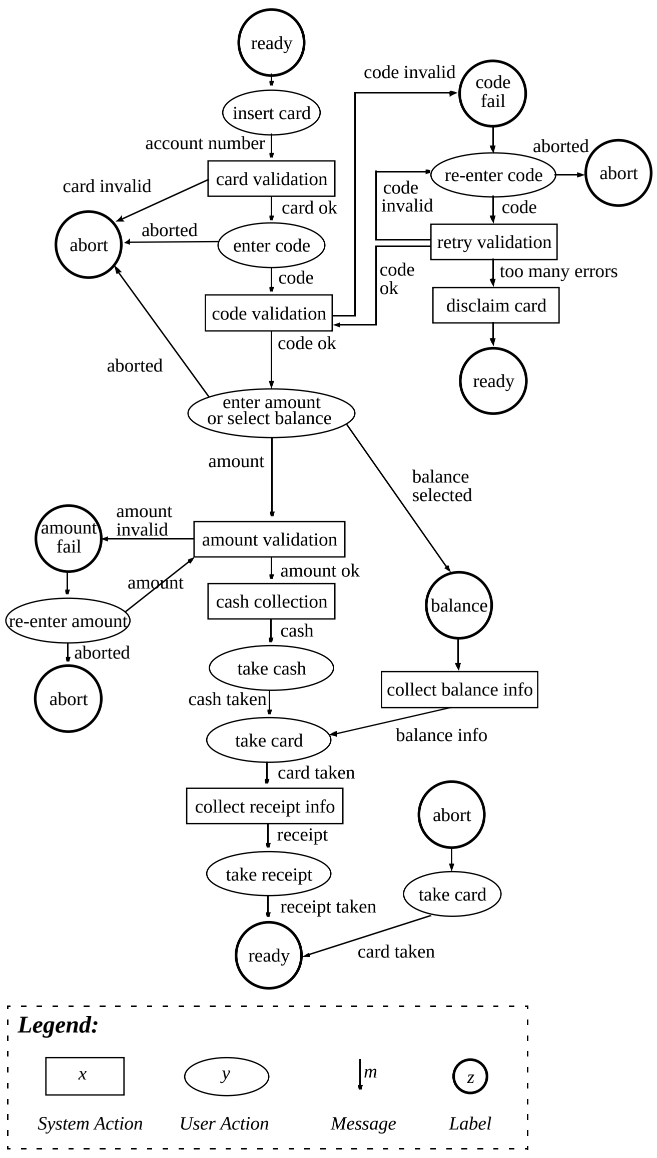

FIGURE 5. The usage view for "ATM Customer"

To illustrate this, in fig. 3 a UCS part is marked with ***1**, which corresponds in fig. 4 to the user action ***1*** "enter code" and the resulting message "code". Similarly, the UCS part denoted ***2** corresponds to the system action "cash collection" and message "cash", marked with ***2***.

An action can have different outcomes. For example, the system action "code validation" may result in the events "code OK" or "code invalid". An action can thus represent a collection of similar protocols, with different outcomes. An action can also be seen as a state where the user or the system tries to accomplish some specific task. The user and system actions could be described internally by finite state machines, as proposed in [5]. This possibility is, however, not yet incorporated into UORE.

Abstract usage scenarios. Using the abstraction mechanisms of user and system actions, use case specifications can be expressed in a more condensed way. Every UCS is transformed into an Abstract Usage Scenario (AUS), drawn as a sequence of user actions (bubbles) and system actions (boxes) interconnected with transitions (arrows) that represent the resulting messages of each action. The invocation and termination context of an AUS is indicated by labels (circles). A label denotes an external system state, i.e. a subset of the carthesian product of all AIO states. In fig. 4 a sample AUS is shown.

The main purpose of creating AUS's, is to make the synthesis feasible even if we have a very large number of use cases. By raising the abstraction level we hide information, to make "clean threads" and then "weave them together" in the last step of integration.

Synthesized Usage Model. The SUM consists of one usage view per actor. A usage view is synthesized from all Abstract Usage Scenarios produced for one specific actor. A usage view is created by finding similar parts of Abstract Usage Scenarios and merging them. The result is a directed graphs with three types of nodes: *user actions*, *system actions*, and *labels*. These nodes have the same meaning as in Abstract Usage Scenarios. Labels are used to maintain traceability between usage views and AUS's. Additional labels can be introduced to divide large graphs into separate sub-diagrams, thus promoting scalability. An example of a usage view is given in fig. 3. (This usage view is an integration of more use cases than shown in the examples in fig. 2.)

To summarize, the SUM contains descriptions of the following elements:

1. Actors
2. Usage views
3. Use case specifications
4. Abstract interface objects
5. User actions and system actions
6. Data dictionary with problem domain objects.

3.5 Verification activity

The purpose of the verification activity is to obtain a consistent and complete SUM. There are two verification steps related to the formalization activity and integration activity respectively:

1. Verification of UCS
2. Verification of SUM.

The verification of a UCS is performed as a rigorous inspection where the UCS is compared with the corresponding use case in the UCM. The reviewers check that the UCS is a correct transformation of the informal use case description, meaning that everything in the use case is contained in the UCS and that the objects in all UCS's are consistently defined.

The second verification step aims at ensuring that the SUM completely covers every UCS. Here is a great potential for automatic verification, where a tool could check that every AUS is a possible path in the corresponding usage view. It is possible that, during the synthesis phase, new user and system actions are discovered and incorporated in a usage view, thus making more usage scenarios valid in addition to the defined AUS's. In the verification of the SUM, such additional usage scenarios can be created by traversing the graphs of the usage views. In this way, the "experimental material" of use cases is used to build a model that enables the discovery of yet unidentified scenarios, and thus making the SUM a model that covers *more* than the initial experiments.

4. Applications of SUM

The Synthesized Usage Model is designed to be used as a *reference model* for the remaining phases of system development. The SUM captures not only functional requirements, but also system usage. The SUM is a source of information about *what* the system is supposed to do, and *how* it should behave from the user's point of view in different usage contexts. Therefore, the SUM can form a backbone for the whole development process including system design, verification, and validation. This role of a formal usage model in system development is discussed in [6]. In the following sections we discuss the potential benefits of SUM in system design as well as in verification and validation. A report on practical experiences in the field of telecommunication is also given.

4.1 System design

The SUM captures functional and behavioural aspects of the system that are important for system design. The user and system actions are abstractions of user-system communication protocols that produce system stimuli and responses.

The semantics of an action is defined by the different contexts in which it can occur, and the set of abstract interface objects it encapsulates. This information can be directly applied in the *external design*, where the mapping of AIO's to actual interface objects, and the concrete shape of the user interface is determined. The SUM can also be used for creating a prototype of the user interface.

The set of atomic system operations and their usage contexts is a valuable information for *internal design.* Here we have to consider the fact that some system operations can be atomic for one actor, but not for another. This information can be useful for finding a robust object structure of the system, and for allocating functionality to objects, as suggested in [1].

4.2 System verification and validation

In order to ensure the correctness of the system implementation and requirements traceability, the system can be verified against the SUM by means of testing. The possibility of *automatic generation of test cases* is one of the most important properties of the SUM. Each usage view of the SUM can be used to generate test cases in the form of "re-created" usage scenarios. These scenarios contain both stimuli to the system and the expected system responses, thus enabling automatic verification of the test results. Though a strategy of test case selection is beyond the scope of this paper, in the next paragraph we briefly discuss the possibility of using the SUM for statistical usage testing.

Statistical usage testing. In statistical usage testing [7], test cases are derived from a usage model. This model describes both functional and statistical properties of system usage. Experiences with the so-called *state hierarchy model* [8, 9], shows that it is feasible to generate test cases automatically from a model of system usage. These test cases are samples of the expected system usage and have the necessary statistical properties that enable certification of the system's reliability.

Statistical usage testing is a black-box testing technique, as it derives the usage model from the requirements specification. Unfortunately, there is a substantial gap between the usage model required for statistical usage testing and the traditional requirements specification. What we need from requirements analysis is an explicit description of system usage. By using the SUM as an element of requirements specification, we can possibly bridge this gap and make test preparation easier. By this approach, test preparation can concentrate on modelling statistical properties by adding probabilities to the SUM.

4.3 Practical experiences

Though UORE, as described in this paper, has been used only in minor case studies, the key elements of the method (synthesis of use cases, single-actor views, and SUM) have already been applied in practice. In the analysis of interactions between pan-European telecom services, these elements of UORE have yielded positive results [10, 11].

The problem of undesired feature interactions is a major threat to the rapid deployment of new telecom services. An interaction occurs when one service feature changes or disables the operation of another feature. One of the approaches to this problem is to detect and resolve interactions during requirements analysis.

By applying the use case driven approach to the analysis of the pan-European candidate services, and synthesizing the use cases, a behavioural model of these services and their features was obtained. This model corresponds to one usage view of the SUM, i.e. it represents the behavioural aspects of the services as seen by one actor, *service user*. This model was analysed by a custom-designed tool that automatically detected a large number of potential feature interactions. The tool used the possibility of re-creating service usage scenarios (in this case different scenarios of telephone calls) from the model, as described in [12] and [10].

5. Conclusions

The ideas introduced in this paper clarify and formalize several important aspects of UCDA. It is our belief that UORE is a significant improvement of UCDA, by its criteria for finding, describing, formalizing, and synthesizing use cases. However, it is still untried on the large scale, and it remains to be proven that UORE is easier to use and gives a better support in requirements engineering than the original UCDA. By empirical studies we hope to prove the benefits of the SUM as a system reference model.

In summary, the main contributions are:

- The improvement of the actor and use case concepts
- The formalization of use case descriptions
- The idea of use case synthesis
- The Synthesized Usage Model
- The process of Usage Oriented Requirements Engineering.

There are still a number of issues to be addressed in future research, for example:

- Formal description of the invocation, termination, and flow conditions of use cases
- Formal description of procedural and non-procedural properties of user and system actions
- Further refinement of use case synthesis – integration of different usage views in the SUM
- Formal definition of the syntax and semantics of SUM
- Transformation of the SUM into a test model suitable for statistical usage testing
- Automation of verification and validation of the test results by using SUM as a reference model of system behaviour.

Acknowledgements

We would like to thank the following persons who helped in reviewing this paper: Claes Wohlin, Daniel Söbirk, Henrik Cosmo, Helen Sheppard, and Anna-Sofia Maurin.

References

[1] Jacobson, I., et al. *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley, 1992.

[2] Sommerville I., *Software Engineering*, fourth edition, Addison-Wesley, 1992.

[3] ITU-T Recommendation Z.120. *Message Sequence Chart (MSC)*, Telecommunication Standardization Sector of International Telecommunication Union, 1993.

[4] Wirfs-Brock, R., et al. *Designing Object-Oriented Software*, Prentice Hall, 1990.

[5] Zave P. “Feature Interaction and Formal Specification in Telecommunications”, *IEEE Computer*, August 1993.

[6] Wohlin C., Regnell B., Wesslén A. and Cosmo H. “User-Centred Software Engineering - A Comprehensive View of Software Development”, *Proceedings of Nordic Seminar on Dependable Computing Systems*, Denmark, August 1994.

[7] Mills, H. D., Dyer, M. and Linger, R. C., “Cleanroom Software Engineering”, *IEEE Software*, pp. 19-24, September 1987.

[8] Runeson, P. and Wohlin, C., “Usage Modelling: The Basis for Statistical Quality Control”, *Proceedings of 10th Annual Software Reliability Symposium*, pp. 77-84, Denver, Colorado, USA, 1992.

[9] Wohlin, C. and Runeson, P. “Certification of Software Components”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp 494-499, 1994.

[10] EURESCOM Project EU-P230, *Enabling pan-European services by cooperation of PNO’s IN platforms*, Deliverable 4, EURESCOM, Heidelberg, December 1994.

[11] Kimbler K., Kuisch E. and Muller J., “Feature Interactions among Pan-European Services”, *Feature Interactions in Telecommunications Systems*, IOS Press, Netherlands, 1994.

[12] Kimbler K. and Söbirk D., “Use Case Driven Analysis of Feature Interactions”, *Feature Interactions in Telecommunications Systems*, IOS Press, Netherlands, 1994.