KATHOLIEKE
UNIVERSITEIT
LEUVEN

DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

RESEARCH REPORT 9947

MODELING THE DIALOGUE ASPECTS OF AN INFORMATION SYSTEM

by
M. SNOECK
G. DEDENE

D/1999/2376/47

Modeling the dialogue aspects of an information system

Monique Snoeck, Guido Dedene

Abstract - **In this paper we investigate techniques offered by current object-oriented development methods for the specification of the user-system dialogue aspect of a software system. Current development methods do not give very extensive guidelines on how to model this aspect and the available techniques need some refinement and elaboration to fit this particular task in the software specification process. The paper first compares a number of approaches. The common elements of these approaches are summarized and further developed into one comprehensive set of techniques that addresses the needs of functional requirements analysis.**

I. INTRODUCTION

Current object-oriented development methods offer a whole set of specification techniques for specifying the various aspects of information systems. More specifically, special attention is given to the static or structural aspects (object-relationship diagram), the behavioral aspects (statecharts) and the interaction aspects (sequence charts). Other dimensions we have to consider are the business rules or domain knowledge that are embedded in the information system and the required user functionality. Current object-oriented development methods spend a lot of attention to the modeling of structural and behavioral aspects of the domain with techniques such as object-relationship diagrams, state charts and interaction diagrams. Much less attention is spent to the modeling of user-system interaction. In addition the techniques current methods offer, are not really sufficient. For example, the specification of a conversation between system and user is difficult to model only with statecharts and object-relationship diagrams. Also sequence charts are not completely adequate because they do not allow to represent branching and iterations. UML proposes several techniques that can be used to model system behavior. However, it is not really clear how these

techniques interact with each other and how they can adequately be combined to model user-system interaction. In addition, the techniques are either very conceptual (and informal) or either very close to software design. In this paper we investigate five current object-oriented development methods and study the techniques they propose for modeling system behavior, and more in particular how these techniques can be used to model the dialogue part of a system (section II). In section III we compare these techniques, identify common elements and eventually, in section IV, propose a theoretical framework of techniques that range from conceptual requirements elicitation to detailed specification for user-system interaction and indicate how each technique can be used as a refinement of the more conceptual technique. Section V gives a final conclusion and outlines further research.

II. Review of current behavior modeling techniques

A. Modeling user functionality in UML

Use cases describe the functional requirements by identifying actors and scenarios of system usage by these actors. As such this technique is a valid candidate to model the interaction between a user and the system. The technique offers some possibilities for modularization by allowing use cases to "include" and "extend" other use cases. The detailed description of a use case mainly focuses on the main flow of events and the possible alternative flows. In all methods [1][6][9], use cases are mainly a support for system design: they are used for finding objects and determining the systems structure. The main problem with this technique is its informal definition: there are no rigorous semantics for the concepts of actor, use case, nor for the relationships "includes" and "extends". Use cases are a good starting point and useful for requirement elicitation, but we need another technique for a more detailed analysis of the dialogue components of a system.

Sequence charts are used to model interaction by means of message passing between objects. Whereas use cases are fairly conceptual, this technique can be characterized as a more detailed design technique. It identifies essential parts of object-oriented *program code*: the objects and the messages they interchange. It is however possible to use sequence charts at a higher conceptual level, for example to model the interaction between a user and the system as a whole. The main shortcoming of this technique is that it does not allow to model alternative scenarios (branching), iterations, ... and so on. To model these, UML [2] proposes activity diagrams. Activity diagrams are akin to Petri nets and focus on the activities that are performed by objects. They allow to model branching, iterations and parallel threads. By organizing the activities into swimlanes, one can partition the responsibilities for activities according to organizational units or objects. Activity diagrams can also be used to model workflows. It is however not clear how to link sequence charts to activity diagrams: for example, do we need to model one sequence chart for each alternative scenario that can be identified in the activity diagram? Such questions remain largely unanswered in UML [2] and in the associated methods [1][6][9].

B. Fusion

In Fusion [3], the system interface is defined by (among others) identifying scenarios of usage. Such scenarios show the flow of communication between (external) agents and the system, and they are documented by means of timeline diagrams. These diagrams are graphically equivalent to sequence charts, but are used at a higher conceptual level: lifelines can be associated to more abstract concepts such as "agent " and "system" and the communication arrows identify high level communication "units". The analysis phase being not concerned with internal messaging between objects, communication is defined in terms of system operations only. System operations are defined as input events and their associated effect. They are invoked by agents only. Responses of the system to the agent are called output messages.

The interface model is formed by developing lifecycle expressions that generalize the scenarios of usage. Lifecycle expressions are defined as regular expressions built using system operations, output messages and sub-lifecycle expressions and the operators sequence, choice, iteration and interleaving. Each system operation is further defined in the data dictionary by means of an operation schema. Fig. 1 is a small example of such a lifecycle expression ([3], p. 53). In this expression, alternatives are separated by a '+' sign, sequence is indicated by a dot '.', interleaving by a double bar '||' and optional elements are between square brackets. Names of lifecycle expressions are written in small capitals, system operations in lower case and output messages names are preceded by a "#".

```
Life cycle ECOSTORAGE = (DELIVERY + COLLECTION)* || STATUS
DELIVERY =load_bay_empty .
          enter_manifest.
          (check_in_drum.#drum_identifier)*
          end_check_in.
             [#discrepancy_in_delivery].
             #delivery_allocation.
             [#drums_to_be_returned]
COLLECTION = ...
STATUS = ...
```

Fig. 1. Lifecycle expression for ECOstorage

C. Syntropy

Syntropy [4] makes a difference between the real-world model (called essential model) and the software model. According to Syntropy, events are fundamental elements of software specification. As such they are one of the building blocks of the essential model. In the software model, interaction between external agents and the software system can be described by means of event scenarios. Such scenarios are documented in a tabular form. In this table, there is one column for each agent. Consecutive rows denote consecutive events. On each row the event

is marked in the column of each involved agent as a stimulus from the user to the system (indicated by a question mark '?') or a response of the system to the agent (indicated by an exclamation mark '!'). Each column of the table is an event scenario, which describes the software system's overall behavior from the perspective of a single agent. Note that in this table the software system is not explicitly modeled. This kind of table is more or less equivalent to a timeline diagram or sequence chart, in that it only can document straight sequences of events: alternatives, repetitions, and other structures cannot be represented. Syntropy does not offer a technique that allows to model the structure of interaction. In fact the complete interaction is partitioned between the system objects that collaborate to realize the scenario. Statecharts of individual objects are used to generalize the scenarios.

D. OO-SSADM

Also OO-SSADM [8] is one of the few methods that make an explicit separation between modeling the domain (called entity-event model) and the required user functionality (called external model). OO-SSADM defines a *User function* as a composite of events and enquiries organized to support the user in carrying out some task or procedure. JSP is proposed as a technique for dialogue design. One JSP-structure is identified for user input and one for system responses. Using the JSP-technique, both structures can be combined to a single structure that identifies the user-system dialogue. OO-SSADM explicitly states that this JSP-structure is not necessarily the best program structure. But it remains of course a valid specification of the dialogue structure. In addition, OO-SSADM also offers a framework for event-driven interface design. This frameworks identifies a number of template user interfaces for different types of events (creation, deletion, modification). These can then be combined into four mini-dialogue patterns. The main characteristic of these dialogue patterns is their linear structure: they have only one scenario and no branching or looping.

E. The Hierarchical Use Case Model

The combination of use cases and sequence charts to model user functionality is far from ideal: there is a significant gap between the conceptual levels of both techniques. It is not really obvious how to refine a use case into one or more sequence charts. Even when the flow of events is described in a structured way, e.g. by means of pseudocode, the transition to a description of user-system interaction is not straightforward. The *Hierarchical Use Case Model* proposed in [7] is one attempt to close this gap. Fig. 2 summarizes this approach.

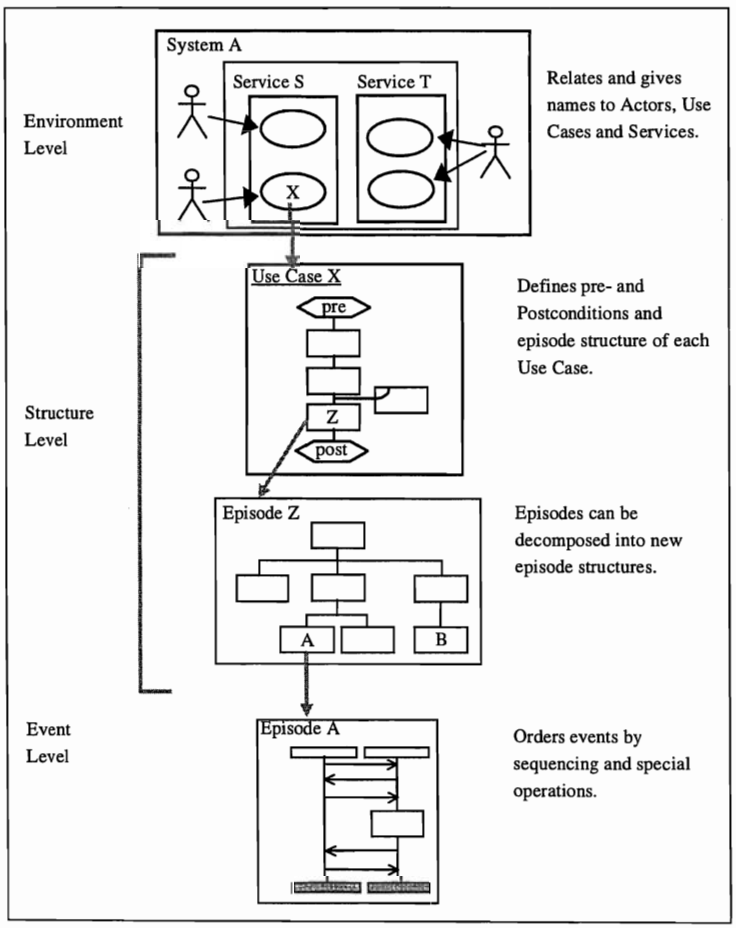

Fig. 1. Overview of the hierarchical use case model

Adding a technique of "episode" modeling between use cases and sequence charts closes the gap. Use cases are refined by specifying the process as an episode structure. Such a structure is a combination of episodes using the operators sequence, choice, iteration, exception and interrupt. Each episode can in its turn be decomposed into a new episode structure. The leaf nodes in this episode decomposition are refined by means of sequence charts. Although it is not explicitly defined in [7], we assume that it is the authors" intent that such leaf-episodes have no branching or repetitions, in other words, that they have a linear sequential structure.

III. COMPARISON

Table 1 summarizes the techniques used in the different methods that were studied. Events are identified as the basic building blocks in four methods: the behavior of a use cases is described as a "flow of events" ([2], p.227-228), system operations in Fusion are defined as input events and their associated effect ([3], p.45), and in Syntropy and OO-SSADM events are already identified during the domain modeling phase and are then used as building blocks during the external or software modeling phase.

It should however be noticed that only Syntropy and OO-SSADM identify real-world events during the domain modeling phase. It are these real-world events that are used to express system behavior from a user's perspective. Also Fusion thinks in this direction in that a system operation is defined as an input event invoked by an agent. In UML and in the HUM however, the events to which use cases and sequence charts refer can both be real- world and information-system events.

All methods also express the need for structuring the behavior according to the basic operations of sequence, choice (branching) and iteration. The need for identifying parallel threads is accounted for in UML by means of Activity diagrams, in Fusion by means of the interleaving operator. In the other methods, parallelism is discussed as a separate topic, but not necessarily related to the

notion of parallel threads in a dialogue. It is however a typical feature of advanced user interfaces that users are allowed to perform several tasks in parallel using a single software system. Current windowing systems offer all necessary technological support for achieving such parallelism. Hence, we believe parallelism to be an essential feature in dialogue modeling.

Table 1. Comparison of behavioral specification techniques.

	Conceptual level	Behavior Structure	Detailed design	Basic building block
UML	Use Cases	Pseudocode, structured English or Natural language	Sequence Chart Activity diagram	Event
Fusion	Usage scenario Time line diagram	Life-cycle expression Operators: sequence, choice, repetition, interleaving	Operation model	System Operation (input event) Output message
Syntropy		Event scenarios	_	Event (stimuli and responses)
OO-SSADM	User function	standard mini-dialogues, JSD-diagrams operations: sequence, choice, repetition	_	Event Enquiry
HUM	Use Case	Episode structure Operators: sequence, choice, repetition, exception, interrupt	Sequence chart	message

IV. A THEORETICAL FRAMEWORK FOR USER-SYSTEM INTERACTION MODELING

During the elicitation phase we need techniques that allow us to identify all required user functions. A possible information source is to look at business processes, refine these to the level of elementary tasks (see Fig. 3). These tasks can then be categorized as either completely manual, fully automated or interactive. Both automated and interactive tasks will require some user function in the software system. Such complex user functions can be specified as use

cases and refined to the suitable granularity using the "includes" and "extends" relationships.

In order to obtain a formal and unambiguous specification of the behavioral aspect of a user function, each use case or complex user function must be defined as a composition of simple user functions. This composition must be defined using sequence, choice, iteration and parallel composition. Complex functions must be decomposed until a granularity of "simple" user functions is obtained. Simple user functions are characterized by the fact that they have a linear dialogue without alternative scenarios, iterations or parallel threads. Typically, such a simple function can be realized by one window in a graphical user interface, or by means of sequence of modal windows. The relationships "includes" and "extends" can be formalized accordingly: the relationship "includes" refers to a substructure, whereas an extension refers to an optional structure, that is the choice between "do nothing" and performing the use case. Finally, each simple function is further documented by means of a finite set of scenarios: one for the basic sequence and one for each possible exception (see further).

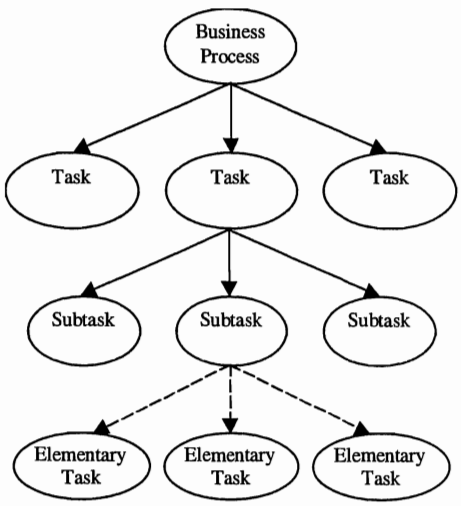

Fig. 3. Business process refined to identify tasks

Both in OO-SSADM and in Syntropy, real-world events are identified as basic components of the domain model or real world model. Events are atomic units of action: they represent things that happen in the world. Without events nothing would happen: they are the way information and objects come into existence (creating events), the way information and objects are modified (modifying events) and disappear from our universe of discourse (ending events). Events are not objects. However, we might choose to record the fact that an event has happened by recording the occurrence of this event as an object. For example in a banking environment, "*withdraw money*" is an event that modifies the state an object "BANK ACCOUNT". We can keep track of all withdrawals by defining "WITHDRAWAL" as an additional object type. An event *withdraw* will from then on have a double effect: it will *modify* the state of an account and *create* a withdrawal. During the analysis stage, it would be irrelevant to determine how both objects will be notified from the occurrence of the withdrawal event. We therefore assume (just as in Syntropy and in OO-SSADM) that events are broadcasted.

The separation of real-world events from information-system events allows a more user-oriented and task-oriented view of information system design. Real-world events are those events that occur in the real-world, even if there is no information system around. Information-system events are directly related to the presence of a computerized system. They are designed to allow the external user to register the occurrence of or invoke a real-world event. For example, the use of an ATM-machine to withdraw money from one's account will invoke the real-world event "*withdraw*" by means of several information-system events such as "*insert-card*", "*enter PIN-code*", "*enter amount*", and so on.

Once events have been identified in the domain model, the whole domain model can be considered as one component, which interface is the set of all events that allow to create, modify and update the information contained in the domain model. User functions are then nothing more than a way to invoke these real-world events. The user function will translate information system events such as

mouse clicks and keystroke actions into the invocation of one or more domain model event.

The meta-model in Fig. 4 represents the components of a specification and identifies the suitable modeling techniques for each specification component. The following simplified banking example shows how the proposed techniques are used to model a complex user function. The (simplified) domain model of the bank contains the object types CUSTOMER and ACCOUNT related by a one-to-many association: a customer holds zero to many accounts and an account is hold by exactly one customer. Business events for CUSTOMER are *create_customer, modify_customer, end_customer*, and for ACCOUNT are *open, deposit, withdraw, close.*

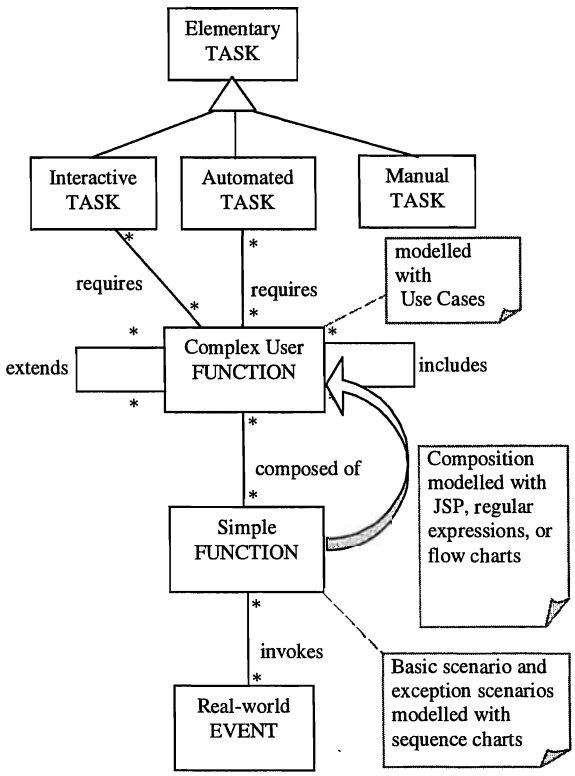

Fig. 4. Meta-model for functionality requirements

The withdrawal of cash by means of an ATM is a complex user function that will eventually invoke a *withdraw* event. Fig. 5 Represents the behavioral structure of this function according to the notations of the HUM. Fig. 6 represents the same structure with a regular expression, such as in Fusion. Notice how this definition uses a recursive definition of the sub-expression SECOND_PASS. The recursive definition can however be avoided by using the empty episode denoted by '1', as shown in Fig. 7.

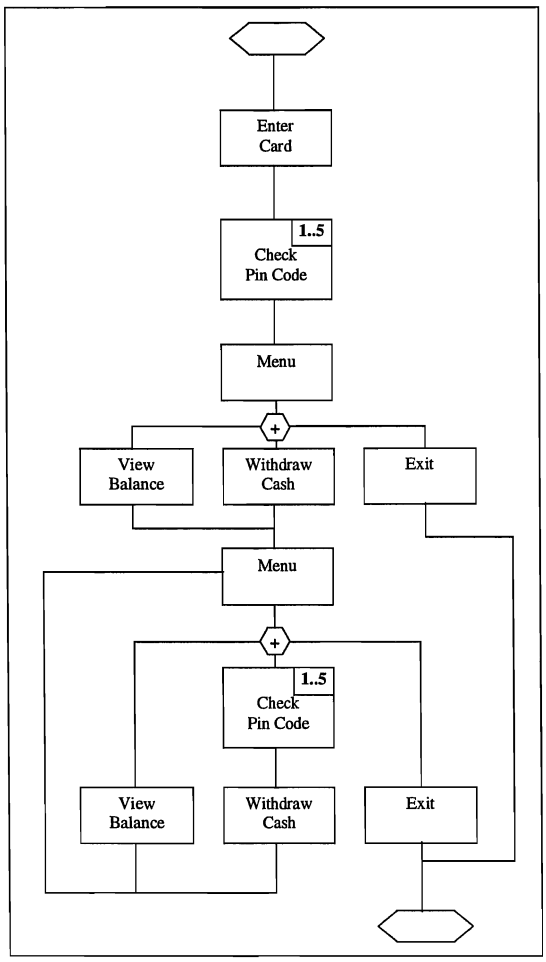

Fig. 5. Episode structure of cash withdrawal with an ATM

```
Use_ATM = Enter_card.(check_pin_code)^1..5.Menu.
(View_balance.SECOND_PASS
+ withdraw_cash.SECOND_PASS
+ exit)

SECOND_PASS = Menu.
(View_balance.SECOND_PASS
+ (check_pin_code)^1..5.withdraw_cash.SECOND_PASS
 + exit)
```

Fig. 6. Function structure of cash withdrawal with an ATM specified with (recursive) regular expressions

```
Use_ATM = Enter_card.(check_pin_code)^1..5.Menu.
(View_balance + withdraw_cash + 1)
.SECOND_PASS*
.exit

SECOND_PASS = Menu.
 (View_balance + (check_pin_code)^1..5.withdraw_cash + 1)
```

Fig. 7. Function structure without recursive definitions

As to be expected, the specification of behavioral structure by means of flowcharts tends to be less structured than the equivalent description by means of regular expressions. Notice also how the formal specification of the dialogue structure allows easily to identify reusable parts of the dialogue.

Refinement of the dialogue aspects of each episode can be done by using a high-level sequence chart. In such a sequence chart, we include all external agents, the function and the domain model. In such a diagram, we can make a clear distinction between "information system events" and "domain events". Events of the first type are those that cross the information system's boundary. The latter are the events generated by the function and broadcasted to the domain objects. Fig. 8 shows a sequence diagram for the simple function Withdraw_Cash. In this example, it is assumed that the withdrawal is successful. A further refinement of simple functions is the specification of all possible exceptions to the

basic linear sequence of events. This means that each exception will identify an additional exit point in case some action fails. In the given example, a possible exception is the refusal of the withdrawal operation by the domain model according to some business rule (e.g. insufficient balance). The exception can be specified as an alternative scenario for the basic scenario. In principle, each simple function will have one basic scenario and any number of "exception scenarios", each documented by means of a sequence chart. The exception scenario for Withdraw_cash is given in Fig. 9.

The specification process given above was presented in a top-down manner. Following the philosophy of the event-driven design method of OO-SSADM, the same result can be achieved by first identifying basic event calling functions and basic enquiries. For the given domain, this means that a simple function is defined for each of the six domain events. View_balance, Search_customer and View-customer can be identified as simple enquiries.

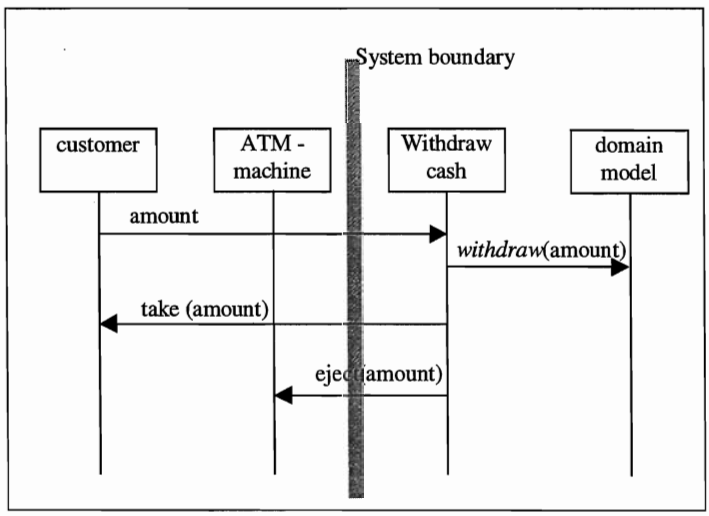

Fig. 8. Sequence chart for Withdraw_Cash

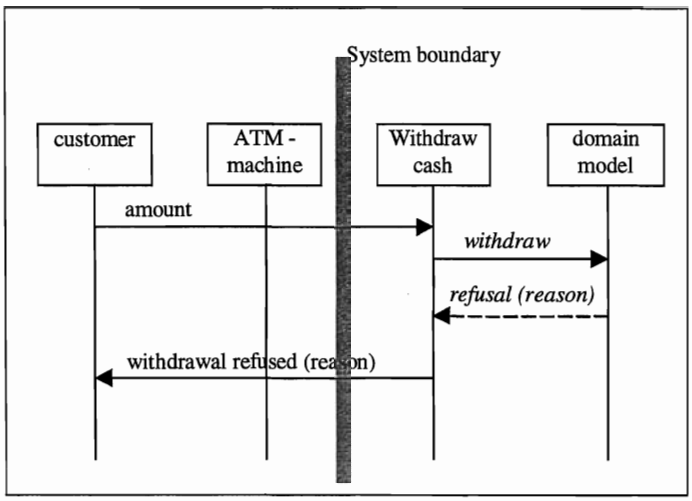

Fig. 9. Exception scenario for Withdraw_Cash

These basic building blocks can then be reused to form more complex dialogues. In the given example, View_ balance and Withdraw_Cash are the simple functions that are reused in the complex function Use_ATM. A service that allows to withdraw money at the counter would reuse the same building blocks and maybe additionally the View_ customer function.

V. Conclusion

User-system dialogue modeling is not very much elaborated in current object-oriented modeling methods. In practice, the user dialogue is probably often directly designed by means of prototyping. We believe however, that a more systematic approach, independent of the eventual implementation technology can improve the design of user-system interaction.

Although the set of methods that was studied is far from exhaustive, all investigated methods show a similar pattern. In addition to the identification of events, some formal specification method is required for establishing the dialogue structure. Dialogue structure has to be decomposed until simple dialogue

components are defined that have a basic linear event sequence. To this basic event sequence one can then add the possible exceptions as alternative paths. As a result of this, each simple function can be described by a finite set of sequence charts: one for the basic scenario and one for each possible exception. The identification of such basic dialogue components is encouraging reuse. If real-world events are identified as basic elements of a domain model, one can build complex functions in a bottom-up way. each domain event will give rise to the definition of one simple function that allows to invoke this event. In addition basic enquiries must be identified, typically at two per domain object: one to view the list of objects in a domain object class and one to view the details of a single object. The simple functions that have been identified in this way can then be composed to form more advanced user dialogues.

Notice that although the proposed techniques were elaborated to model on-line dialogues, they can to some extent be used to model off-line services as well. Further research should assess the usability of the techniques in this area.

Our fist concern will however be the mathematical elaboration of the proposed techniques by means of process algebra. In [10] it has been demonstrated that domain modeling can successfully be formalized by means of a process algebra similar to CSP. It would be interesting to elaborate this process algebra to allow it to formalize functionality modeling techniques as well. In parallel a more advanced practical evaluation of the techniques by means of real-world examples is planned.

VI. REFERENCES

[1] Grady Booch, *Object Oriented Analysis and Design with Applications.* Second Edition, Benjamin/Cummings, Redwood City, CA, 1994.

[2] Grady Booch, James Rumbaugh, Ivar Jacobson, *The unified modeling language user guide*, Addison Wesley, 1999

[3] Derek Coleman et al, *Object-oriented development: The FUSION method*, Prentice Hall, 1994

[4] Steve Cook, John Daniels, *Designing object systems: object-oriented modeling with Syntropy,* Prentice Hall, 1994

[5] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice-Hall International, Series in Computer Science, 1985

[6] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, *Object-Oriented Software Engineering, A use Case Driven Approach*, Addison-Wesley, 1992

[7] Björn Regnell, Michael Andersson, Johan Bergstrand, A hierarchical use case model with graphical representation, *Proceedings of the IEEE international symposium and workshop on engineering of computer-based systems*, March 1996

[8] Keith Robinson, Graham Berrisford, *Object-oriented SSADM*, Prentice Hall, 1994

[9] Rumbaugh, J., Blaha M., Premerlani, W., Eddy, F., Lorensen, W., *Object Oriented Modeling and Design*, Prentice Hall International, 1991

[10] M. Snoeck, G. Dedene, Existence Dependency: The key to semantic integrity between structural and behavioural aspects of object types, *IEEE Transactions on Software Engineering* , Vol. 24, No. 24, April 1998, pp.233-251