TUM

INSTITUT FÜR INFORMATIK

Modeling the Dynamic Behavior of Objects On Events, Messages and Methods

Ruth Breu and Radu Grosu

TUM-I9804
Februar 98

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-02-I9804-300/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Modeling the Dynamic Behavior of Objects On Events, Messages and Methods

Ruth Breu, Radu Grosu

Institut für Informatik, TU München, D-80290 München
email:breur,grosu@informatik.tu-muenchen.de

Abstract. Events, messages and methods are concepts supported by most object-oriented analysis and design techniques. The interrelation between these concepts is however not yet fully understood and guidelines and techniques for method specification remain unprecise and incomplete. In this paper we provide a simple object model which we use both to clarify the above interrelation and to devise a method for specifying the dynamic behavior of objects. In a two layered approach, this method integrates the description of the object's life-cycles with the specification of the object's methods. Our object model is characterized by two important assumptions, namely that methods are virtual objects and that messages sent to inexistent objects are returned back as an error. Both assumptions are supported by practical evidence and allow us to model internal concurrency, multiple threads and attribute sharing in a very simple and elegant way. Our ideas are illustrated on a simple banking example.

Keywords OO analysis, event, message, method, state transition diagram.

1 Introduction

Events, messages and methods are three central concepts for modeling the dynamic behavior of objects. Communication between objects by sending messages, changes of object states caused by incoming events and interface design based on methods are catchwords of object-oriented analysis and design.

However, in most frameworks like OMT [9], the Booch method [2] and the forthcoming method UML [3] messages, events and methods are separate concepts used in different parts and phases of system development. The interrelation between these concepts remains often unclear and is left to the interpretation of the system designer. In particular, the transition from high level event-based descriptions to method specification and method implementation is hardly supported in current object-oriented analysis frameworks.

This lack of preciseness and support can cause severe problems within the design of sequential systems. In environments where objects coexist and act in parallel, guidelines and well-founded techniques for specifying methods are even inevitable and a prerequisite for reliable system design.

The focus of our paper is to provide clear concepts and techniques for the dynamic modeling of objects in such concurrent environments. The central description technique we rely on are state transition diagrams. Most object-oriented analysis techniques offer some kind of state transition diagrams, often based on Harel's state charts [7].

State transition diagrams support an event-based design of objects. They associate each object with a finite set of states and model state changes caused by incoming events. We use a powerful variant of state transition diagrams in which state transitions can be associated with outgoing events and state transitions can be guarded by preconditions and followed by postconditions. The kind of state transition diagrams we propose are well-founded and provided with a formal semantics in a concurrent setting of objects [6, 4].

The notion of methods we conceive is more general than to be an equivalent to procedures in a programming language. In our intuition, methods in the analysis phase model high-level activities of objects. Examples for such methods are the transfer of money in a bank or the reservation of a hotel room. In this view, a general model of concurrently acting objects is inevitable since high-level activities often are conceived to be parallel while their later realization is sequential.

Concerning the design steps for developing a system description based on state transition diagrams, we propose a two-layered technique. In a first stage, a purely event-based description by state transition diagrams is developed. Events are conceived as stimuli at a point in time causing reactions of the stimulated object. The developed state transition diagrams in this stage define for each object allowable sequences of incoming events.

In a second stage of the design, the reactions of an object initiated by events are further specified. Roughly, each such kind of reaction corresponds to a method and the initiating event corresponds to the call of the method. We promote a specification of methods within the framework of state transition diagrams. This has two reasons. First, the use of a uniform framework supports step-by-step design. Relations between different stages can be established and checked. Second and even more important, in a general framework of concurrently acting objects methods generally cannot be modeled in an isolated way but the whole object behavior has to be considered.

Our notion of an object is not limited to the view of a sequential machine reacting to events subsequently. More general, object behavior may comprise internal parallelism and simultaneous computation of methods. This general notion of objects corresponds to modern object-oriented programming languages like Java and is a prerequisite for modeling high-level activities of objects.

The structuring of the following sections is as follows. In section 2, we present some general considerations about the notions of events, messages and methods and sketch our two-layered technique for specifying the dynamic behavior of objects. In the subsequent sections, this technique is applied in detail on a banks and accounts example.

2 Objects, Events, Messages and Methods

Object-oriented techniques offer a variety of notions and concepts that are the basis for comprehending and structuring the dynamic behavior of objects. These notions inherently are based on two different views of objects and their communication with the outside. We call these views the *communication view* and the *interface view*.

The Communication View In the communication view, objects communicate with each other by sending *messages*. Messages are typed data values that are exchanged by objects. The arrival of a message at some object is called an *event*. Events are stimuli at some point of time that cause reactions of the stimulated object.

In the sequel we will use the terms *object communication behavior* or *event-based specification* when referring to specifications of dynamic object behavior based on messages and events.

The best accepted technique for describing object communication behavior are *state transition diagrams*. State transition diagrams are associated with single objects describing state changes initiated by events. The kind of state transition diagrams we use in our approach will be presented in the next sections.

The Interface View In the interface view, the communication between objects is based on the notion of *methods*. A method is a service offered by an object to its clients and calling a method is the only way a client can have access to an object.

In contrast to the notions of message and event which are atomic entities referring to a single point of time, methods can be rather complex processes involving communication with other objects (called servers). In particular, each method needs some period of time to complete. In our general setting of concurrent objects, methods do not have to be processed sequentially within an object, but many methods can be executed in parallel. Moreover, different threads of an object may involve the same method. For these reasons, we will present a technique which conceives methods as private virtual objects which act in parallel with the methods' owner.

Methods can be classified according to their complexity in three categories. In the first category, the execution of a method involves only data local to the object. As a consequence, no additional communication with server objects is necessary. In the second category, the method initially performs some local computation and subsequently delegates the further processing to some server object. Finally, in the third category, the execution of a method involves a complex interaction with server objects.

Current object-oriented analysis frameworks do not integrate the two views of object behavior. As a consequence, it remains unclear how method specifications have to be interpreted within the overall object behavior. The subsequent sections provide guidelines and techniques how an integration of the two views can be achieved.

Interrelations between the two views A first simple observation is that each method call is a message sent from the calling object to the called object (this is, in fact, the Smalltalk terminology). The arrival of such a message at the called object is an event and causes reactions. These reactions can be understood as the execution of the corresponding method.

In a concurrent framework methods are typically accepted subsequently, but handled in an overlapping, concurrent way. The transfer of money from one account to another in our banking example will be of this kind. Since the process of transferring money takes time, the bank should be able to process many transfers concurrently. As a consequence, methods cannot be specified in an independent way but have to be integrated within the overall object communication behavior.

We pursue a two-layered technique for dynamic object specification integrating the event-based and the method-based view. The technique is intended to give a rough guideline and intuition for the design of concurrently acting objects.

Step 1: For each class of objects a set of methods or input messages is identified. Additionally, a state transition diagram is developed describing the input behavior of the objects, i.e. the sequences of input messages the objects can accept.

Step 2: The object reactions are further specified by refining the state transition diagram resulting from the first step. Again, we can distinguish two substeps.

- **Substep 1:** Each method is specified separately. Pre- and postconditions describe the effect the method execution has on the object's attributes, and messages are specified which are sent to server objects. In this state, typically, a set of further messages has to be introduced handling returned values and failure information.
- **Substep 2:** The method specifications are integrated in the state transition diagram of the first stage giving the whole object behavior. It is in this stage that the designer has to decide if methods are handled subsequently or concurrently.

The Object Model A system is represented in our object model as a set of interacting objects which are grouped into classes. Each class consists of a *class manager* and a set of *class instances*.

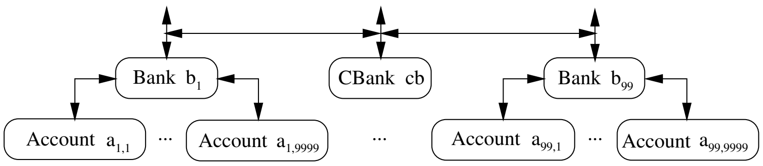
One of our main assumptions is that messages sent to inexistent objects are returned back as an error. This assumption is supported by most standards for open distributed systems and it is assured by the communication medium. In an object-oriented setting this responsibility can be spread among the class managers if all the instance objects in a class are restricted to send their messages through their associated class manager. However, we take a more decentralized approach which simplifies the communication scheme. The class manager does not create or destroy the instance objects and does not control their intercommunication. The class manager merely *activates* or *deactivates* instance objects from a set of objects which already exist and which are parallely composed with the class manager. Inactive objects only return an error message to each method call.

The second assumption is that complex methods which require interaction with other objects are modeled by private *virtual* objects which are also called *clerk* objects. Similarly to real objects, the clerk objects associated to a method already exist and act in parallel with the method's owner which is also called the *organizer* object. The organizer receives stimuli from the environment and delegates their execution to the clerk objects. As a consequence, methods can act in parallel and different threads for the same method are executed by different clerks.

As usual, we classify the communication patterns between a client and its server object in synchronous, asynchronous or delegating. *Synchronous* method call means that the client sends the method call and waits for the answer. *Asynchronous* method call means that the client may accept or send other messages between the method call and the receiving of the answer. *Delegating* method call means that the client sends a method call together with an address to which the answer should be sent and proceeds. The modeling of these communication protocols within the framework of state transition diagrams will be exemplified in the following sections.

Note that our object model is a conceptual model which is very convenient for the analysis or design phase because it is very simple. However, this model does not prescribe the way objects and classes should be implemented. While in the analysis or design phase it is not relevant if all potential objects are already created or if they are created as required, this is an important concern for an efficient implementation. Similarly, while in the analysis phase it is less relevant who returns an error if the addressed object does not exist, in an implementation this is usually the concern of the communication medium. Finally, a particular implementation could restrict the parallelism implied by the organizer/clerk paradigm.

The Banks and Accounts Example We illustrate our approach using a banking system example (see the figure below). This system consists of a set of



concurrently acting banks. Each bank has an owner and a unique bank number ranging between 1 and 99. Banks can be founded and liquidated, i.e. the banking system has a dynamic structure.

The main task of a bank is to organize access to an associated set of accounts. Each account belonging to a bank has an owner, a balance and a unique account number ranging between 1 and 9999.

Clients can interact with a bank by opening and closing an account, by crediting money to an account, debiting money to an account and transferring an amount from one account to another account (possibly belonging to a different bank). All transactions have to refer to existing banks and existing accounts.

The account and the bank objects will be specified in the following sections, illustrating the design steps of our design method.

3 The Specification of Accounts

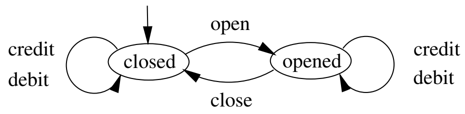
The specification of an account object identified by *ac* is given below. It consists of an attribute declaration part and a state transition diagram. Attributes are defined by their name, their type and (optionally) an initial value. An account has two attributes: the owner name *ow* and the current amount *am* of money.

The state transition diagram has two purposes. First, it defines the interface of the object as a set of input messages. The messages may have parameters which can be missing in a first iteration. Second, the state transition diagram defines the sequences of messages the object can receive. Account objects can be in two states, the *closed*, i.e., the inactive state and the *opened* state, where *closed* is the initial state (indicated by the small arrow). Each transition connecting two states is associated with an input message and, optionally, with a precondition, a postcondition and one or several output messages. In order to reduce the complexity of the diagram, the transitions are given in tabular form. Each line in the table is an annotation of the corresponding transition in the diagram.

attributes

ow = "" : Name
$am = 0$: Int

transitions



name	source	dest	in	pre	out	post
open	closed	opened	$ac?\text{open}(o, a, r)$		$r!\text{ok}$	$ow' = o,\ am' = a$
credit	closed	closed	$ac?\text{credit}(a, r)$		$r!\text{err}$	
	opened	opened			$r!\text{ok}$	$am' = am + r$
debit	closed	closed	$ac?\text{debit}(a, r)$		$r!\text{err}$	
	opened	opened		$am \geq a$	$r!\text{ok}$	$am' = am - r$
				$am < a$	$r!\text{lowBal}$	
close	opened	closed	$ac?\text{close}$		$r!\text{ok}$	

For example, if the account object *ac* is in the state *opened* and it receives a debit message $debit(a, r)$, written as $ac?debit(a, r)$, where a is the amount to be debited and r is the object to which the answer has to be sent, then if the current balance *am* is greater than a an *ok* message is sent to r, written as $r!\text{ok}$, and the current balance is decremented by a. The new balance is written as am' similarly to the Hoare calculus. In the other case, the error message *lowBal* is returned. Strictly speaking, this specification corresponds to two (guarded) transitions in the diagram, one for each precondition.

The reaction to the message *credit* is specified in a similar way. The message *open* activates the account object and sets its initial amount and owner. All methods on account objects are of the most primitive type involving internal data solely.

The precondition (the guard) can be an arbitrary predicate over the attributes and the input messages. The postcondition (the effect) can be an arbitrary predicate over the unprimed attributes, the primed attributes, the input and the output. A transition is activated only if the input matches the input pattern, the output matches the output pattern and both the precondition and the postcondition are true.

Note that the state transition diagram of accounts is complete w.r.t. its methods *credit* and *debit*, i.e., account objects accept these methods in any state and in any order. In a concurrent environment of objects completeness is an important property of state transition diagrams since servers cannot restrict their clients to send specific message sequences. Completeness w.r.t. the class methods *open* and *close*, in contrast, has not to be required since these methods are always sent by the class manager.

4 The Specification of Banks

Banks are complex objects whose methods not only require additional communication with servers but also can act in parallel. A bank is also a *class manager* for account objects. Banks handle the transactions of the clients. In particular, they manage the access to the accounts. Similarly, the *central bank* is the class manager for the banks themselves. Since it does not present new aspects, the specification of the central bank and the activation and deactivation of banks is not presented in this paper. In the specification of the bank we make the bookkeeping activity explicit. However, this behavior could be assumed to be inherited from a predefined general class manager object.

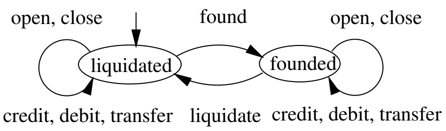
4.1 Define Attributes and Input Messages (Step 1)

A bank has as attributes the bank's owner ow, the account numbers of its active accounts aA and the account numbers of the inactive accounts fA. The bank's identifier is b_i.

attributes

$ow = ""$: Name
$aA = \emptyset$: Set Nat
$fA = \{i \mid 1 \le i \le 9999\}$: Set Nat

transitions



The attributes aA and fA allow the bank to keep track of its server objects. The bank can receive the following input messages: $found(o)$ – activate the bank and set the owner to o, *liquidate* – perform some closing work and deactivate the bank, $open(o, a)$ – open an account with owner o and amount a, $close(k)$ – close the account k, $credit(k, a)$ – credit the amount a to account k, $debit(k, a)$ – debit the amount a to the account k and $transfer(f, b, k, a)$ – transfer the amount a from account f to account k at bank b. These messages match exactly the methods which a bank offers.

Additionally, a (complete) state transition diagram describing the states of the bank and the input messages the bank can accept is given above. Again, for brevity, the method arguments are ignored. The specification obtained in this step is often called *life-cycle* specification.

4.2 Specify each Method Separately (Step 2.1)

In order to specify the bank reactions for each method we have to define first the *answer messages*. Moreover, input messages in most cases have to be enhanced by an answer address indicating the object a possible answer has to be sent to. If the method requires no answer or if the return address can be derived from other information, the return address can be omitted.

For bank objects we introduce the following answer messages: $noAcc$ – a new account cannot be opened, $noAcc(b, k)$ – there is no account number k at bank b, $tansferOK$ – transfer has been successfully completed. In complex diagrams, it is advantageous to keep tables associating the methods with their corresponding input and output messages.

The behavior of the bank object in its inactive state *liquidated* is similar to the behavior of the account object in its inactive state *closed*. We therefore do not further describe it. In the state $founded$ the methods $open, close, credit$ and $debit$ can be specified as simple annotations to the corresponding transition of the life-cycle diagram developed in the previous step. Their specification is given in tabular form below, where the source and the destination state $founded$ is suppressed for brevity. The specification of the methods $found$ and $liquidate$ can be given in an equivalent way to the *open* and *close* methods on accounts.

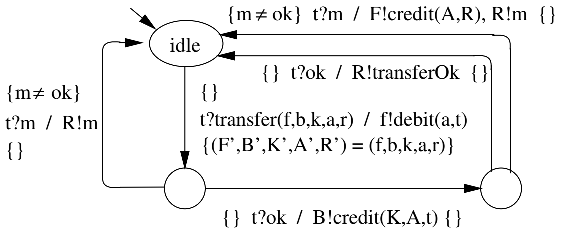
name	in	pre	out	post
open	$b_i?\text{open}(o, a, r)$	$fA = \emptyset$	$r!\text{noAcc}$	
		$fA \neq \emptyset$	$a_{i,k}!\text{open}(o, a, r)$, $r!k$	$fA' = fA \setminus \{k\}$, $aA' = aA \cup \{k\}$, $k = \text{new}(fA)$
close	$b_i?\text{close}(k, r)$	$k \notin aA$	$r!\text{noAcc}(b_i, k)$	
		$k \in aA$	$a_{i,k}!\text{close}$	$aA' = aA \setminus \{k\}$ $fA' = fA \cup \{k\}$
credit	$b_i?\text{credit}(k, a, r)$	$k \notin aA$	$r!\text{noAcc}(b_i, k)$	
		$k \in aA$	$a_{i,k}!\text{credit}(a, r)$	
debit	$b_i?\text{debit}(k, a, r)$	$k \notin aA$	$r!\text{noAcc}(b_i, k)$	
		$k \in aA$	$a_{i,k}!\text{debit}(a, r)$	

The methods *open* account and *close* account are class methods for the account objects. These methods change the active-accounts and the free-accounts bank attributes and activate, respectively deactivate, the corresponding account objects. The *open* method also returns the new account number k; $new(fA)$ is assumed to choose an element out of the set fA.

Note that the new account number k is not the identifier of the corresponding account, because the accounts are private to the bank, i.e., they cannot be addressed directly. We use $a_{i,k}$ to denote the identifier of account number k at bank number i. The mapping a can be imagined as an encryption mechanism which is private to the bank. Since the maximum number of active accounts is limited in the problem statement, a call to open an account may also lead to failure.

The methods *credit* and *debit* have a similar structure. If the given account is not in the set of actual accounts, the message *noAcc* is returned. In the other cases, the method is delegated to the corresponding object. Thus, referring to the communication patterns sketched in section 2, the calls of the methods on account objects in the above table are delegating method calls.

The methods specified so far did not comprise communication with servers and thus could be specified as simple annotations to the life-cycle diagram. The method $transfer(f, b, k, a, r)$, in contrast, requires communication both with the account f from which the amount a of money should be transferred and with the bank b to which the money has to be transferred. The transfer method thus involves a complex process described by a separate state transition diagram given below.



Conceptually, each state transition diagram describes a clerk object with an own state. This object is identified by an identifier variable which will be later bound to the state transition diagram describing the whole object behavior. The state of a clerk object provides the clerk with the data necessary to execute the method.

In our example, the clerk object describing the transfer method is identified by the variable t. It is defined by attributes corresponding to the parameters of the method (denoted in capitals). Informally, for transferring an amount a first a is withdrawn from the account f. If the withdrawal has been successful, a message to the target bank b is sent for crediting a to the account k. If this transaction has been successful, the message $TransferOK$ is sent back to the object identified by the return address r. In the other case, the money is credited again to the account f. Moreover, in all failure cases, corresponding messages are sent back to the return address r.

Note the simple and elegant way we cope with the subtle problem of object deletion. Before the clerk object starts its execution, it is possible that both the source and destination bank as well as the source and the destination accounts have been deleted. In all these cases, the clerk object behaves in the expected, well behaved way.

4.3 Integrate the Methods (Step 2.2)

Up to now the object input behavior and the reactions to input stimuli have been specified in an isolated way. It is in this step that the two specifications are integrated and the overall object behavior is described. We distinguish two fundamental ways how the integration of methods can be handled.

Sequential execution: The object receives the call of the method and performs the object reactions. After the method execution has been completed, the object is able to receive the next input event.

Parallel execution: The object receives the call of the method and performs the object reactions. Concurrently, the object is able to receive the next input event.

Both kinds of method integration will be discussed below and techniques for the integration of the respective state transition diagrams will be presented.

Parallel and sequential handling of methods only give a rough guideline for the integration. Both techniques can be combined in a very flexible way. Since each method is modeled separately, some methods may be executed concurrently, while others may be executed sequentially. Moreover, sequential and parallel execution can be combined within one method in the sense that in parts of the method other input events may be received, whereas other parts of the method are executed exclusively.

Parallel Execution Conceptually, state transition diagrams describe objects as sequential machines. In this framework concurrent behavior is limited to the view of these sequential machines acting in parallel. As we have discussed earlier, conceiving objects as sequential machines is too constraining and in many applications objects incorporate internal parallelism.

The transfer method in our banking system is a typical example for internal parallelism within a single object. Since the communication with the destination bank takes time (up to several days), the transferring bank of course cannot be blocked for other transitions during this period. Thus, a bank receiving the input stimulus to transfer money concurrently executes the transfer and is able to receive new input stimuli. In order to cope with this kind of internal parallelism, we let the bank be the organizer receiving input stimuli by other objects and delegating their execution to transfer clerks. The transfer clerks have been specified in step 2.1. Each transfer clerk t deals with one execution of the transfer method. Thus, in step 2.2, it remains to complete the specification of the organizing bank. This amounts to specify the delegation of the transfer method given as an annotated transition of bank objects below.

name	in	pre	out	post
transfer	$b_i?\mathrm{transfer}(f, b, k, a, r)$		$t!\mathrm{transfer}(a_{i,f}, b, k, a, r)$	$fT' = fT \setminus \{t\}$ $aT' = aT \cup \{t\}$ $t = \mathrm{new}(fT)$

Note that using the organizer/clerk paradigm many transfer transactions can be executed in an overlapping way. In other words we can model both internal concurrency and multiple threads. The bank keeps track of the active and inactive transfer objects by holding the corresponding lists of identifiers aT and fT

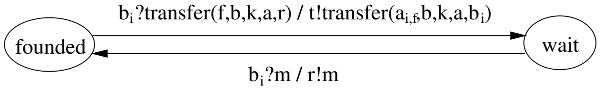
as attributes. As a consequence, we have to add the following attribute declarations.

$$\begin{array}{ll} aT = \emptyset & : \text{Set TransferId} \\ fT = \{t \mid t \in \text{TransferId}\} & : \text{Set TransferId} \end{array}$$

Concurrent behavior within a single object may lead to concurrent access to the object's data, i.e. its attributes. In our conviction the early phases of design should be concerned with behavioral aspects of objects only and questions of data access should be deferred to later stages of the design. In our approach concurrent behavior within a single object is modeled by the introduction of virtual clerk objects having an own state space. That way, attribute sharing is replaced by message passing between organizers and clerks.

Sequential Execution Sequential execution of some method means that the object receives the input message and subsequently executes the corresponding method. After the execution has been finished, a new input message can be received.

In the organizer and clerk objects paradigm, this amounts to a synchronous communication between the organizer and the clerk object. The organizer object delegates the execution of the method to the clerk object, but now waits for the answer. By this kind of modeling, the modular specification of methods is maintained.



The bank object sends a message to the transfer object giving its own identifier for the return address. The answer is then forwarded to the object r. Note that in this case, only one transfer object t has to be composed in parallel with the bank object. Moreover, the identifier t of this object is a constant attribute of the bank.

Banks as specified above receive the message to transfer money and handle the whole transfer transaction before a new message can be received. During the transfer transaction, the bank is blocked for other transactions. This kind of modeling could be adequate if the banks are related by a fast electronic connection.

Sequential and concurrent modeling of the transfer method could be combined in such way that the organizing bank could perform the withdrawal from the source account before it delegates the transaction to the clerk object. In this sense, the two techniques integrating method specifications and object input behavior presented in this section just give a rough guideline how methods can be combined and the overall object behavior is defined.

5 Conclusion

In the preceding sections we have presented a design method for modeling the dynamic behavior of objects in the framework of state transition diagrams. The presented concepts can be applied in various object oriented analysis frameworks such as OMT or UML. A major focus has been the integration of method specifications and object life-cycle specifications.

Based on a very simple and intuitive object model, object behavior may include internal concurrency and multiple threads. Moreover, a technique for modeling object creation in state transition diagrams has been presented. This technique is both formally founded and easy to handle. We considered a general notion of methods covering high-level activities of objects. In our approach these methods are modeled by own objects.

Similar ideas have been discussed in the context of object-oriented business process modeling. In [1] and [8], for instance, the modeling of business processes either by methods or by own objects is compared. Our approach goes further in the respect that the specification of high-level processes is integrated in the overall object behavior and the distinction between modeling these processes by methods or by own objects becomes superfluous.

Concerning the specification of methods, most analysis techniques suggest the use of pre- and postconditions, e.g. [5] and [9]. Since the specification of methods by pre- and postconditions requires methods to be atomic units, these approaches do not consider inter-object communication within methods and thus are limited to the design of sequential systems. In our approach, the use of pre- and postconditions for modeling concurrent systems is enabled by associating them with atomic events instead of complex methods.

Other analysis frameworks suggest the use of techniques describing inter-object communication for specifying methods. Examples are the use of variants of message sequence charts, e.g. in [9] or interaction diagrams [2]. These techniques are aimed to specify exemplary behavior of methods and thus are not powerful enough to describe the overall object behavior.

Nevertheless, diagrams describing inter-object communication are a valuable technique since they support the description of chains of message calls. These chains of message calls are distributed along the state transition diagrams of the addressed objects and thus are modeled only implicitly in our approach. In future work, we will therefore combine the two kind of techniques in order to give the designer more freedom to model explicitly various aspects of object behavior. Due to their expressive power and their semantic background, our framework of state transition diagrams can serve as a basis for such an integration.

References

1. M. Bauer, C. Kohl, H.C. Mayr, and J. Wassermann. Enterprise modeling using OOA techniques. In G. Chroust et al., editor, *Proc. CON 94, Workflow Management*. Oldenbourg, 1994.
2. G. Booch. *Object Oriented Design*. The Benjamin/Cummings Publishing Company, 1991.
3. G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language for Object-Oriented Development, Version 0.9, 1996.

4. M. Broy, R. Grosu, and C. Klein. Reconciling real-time with asynchronous message passing. Will appear in FME'97 Proceedings, September 1997.
5. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object-Oriented Development: The Fusion Method.* Prentice-Hall International, Inc., 1994.
6. R. Grosu, C. Klein, B. Rumpe, and M. Broy. State Transition Diagrams. Technical Report TUM-I9606, Technische Universität München, June 1996.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
8. G. Müller-Luschnat, W. Hesse, and N. Heydenreich. Objektorientierte Analyse und Geschäftsvorfallsmodellierung. In H.C. Mayr and R. Wagner, editors, *Objektorientierte Methoden für Informationssysteme.* Springer, 1993.
9. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design.* Prentice Hall, 1991.

This article was processed using the LaTeX macro package with LLNCS style