Modeling the Dynamic Behavior of Objects On Events, Messages and Methods (Extended Abstract)

Ruth Breu, Radu Grosu

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

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 [5], the Booch method [1] and the new method UML [2] 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.

The focus of our paper is to provide clear concepts and techniques for the dynamic modeling of objects in concurrent environments. The central description technique we rely on is a powerful variant of state transition diagrams [4, 3]. In these diagrams, transitions are associated with triples consisting of a precondition (the guard), a set of input/output events and a postcondition describing the change of state.

The notion of methods we consider is more general than the notion of procedures in a programming language. In our intuition, methods 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 even if 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 pursue the 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 successively. More general, object behavior may comprise internal parallelism and simultaneous computation of methods, i.e., multiple threads. Our object model is characterized by two important assumptions, namely that methods are virtual objects (called clerks) and that messages sent to inexistent objects are returned back as an error. Both assumptions are supported by many standards for open distributed systems and serve as a prerequisite for modeling high-level activities of objects.

We illustrate our approach by specifying the behavior of a simple bank. 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.

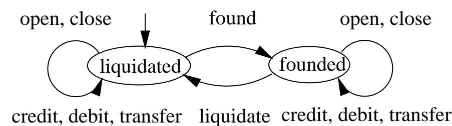
Clients can interact with a bank by opening and closing an account, by crediting and debiting money to an account, resp., and transferring an amount from one account to another account (possibly belonging to a different bank). A bank is a *class manager* for account objects. Banks handle the transactions of the clients. In particular, they manage the access to the accounts.

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
$fA = \{i \mid 1 \leq i \leq 9999\}$	: Set Nat

transitions



The attributes aA and fA allow the bank to keep track of its server objects.

Step 1 – Identify input messages and specify input behavior
The bank can receive the following input messages: $open(o, a)$ – open an account with owner o and amount a, $close(k)$ – close the account k, $credit(k, a)$ and $debit(k, a)$ – credit and debit the amount a to account k, resp., $transfer(f, b, k, a)$ – transfer the amount a from account f to account k at bank b. These messages match exactly the methods which a bank offers.

A (complete) state transition diagram as given above describes the input messages the bank can accept (for brevity, the method arguments are ignored). This specification is often called *life-cycle* specification.

Step 2.1 – Specify each method separately
In order to specify the bank reactions for each method we first have to define the *answer messages*. Moreover, we have to enhance the input messages with *return addresses* 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.

The methods $open, close, credit$ and $debit$ can be specified as simple annota-

tions to the corresponding transition of the life-cycle diagram developed in the previous step. Their specification is given in tabular form below.

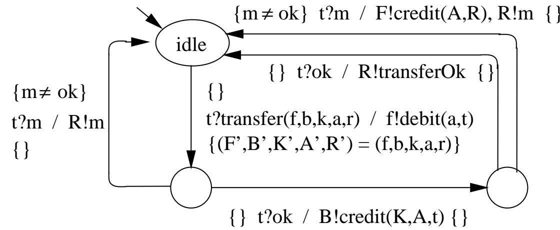
name	in	pre	out	post
open	b_i?open(o, a, r)	$fA = \emptyset$	r!noAcc	
		$fA \neq \emptyset$	$a_{i,k}$!open(o, a, r), $r!k$	$fA' = fA \setminus \{k\}$, $aA' = aA \cup \{k\}$, $k = \text{new}(fA)$
close	b_i?close(k, r)	$k \notin aA$	r!noAcc(b_i, k)	
		$k \in aA$	$a_{i,k}$!close	$aA' = aA \setminus \{k\}$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

The methods specified so far did not comprise interaction 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 such state transition diagram describes a *clerk object* with an own state. This object is identified by an identifier variable which will be bound

in step 2.2 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.

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.

Step 2.2 – Specify the overall object behavior
In the last step we have to integrate the method specifications by defining the overall object behavior. We distinguish two fundamental ways of methods integration: sequential and parallel. Sequential integration corresponds to the usual notion of operation in programming languages and allows one method execution at a time, while parallel integration allows arbitrary many parallel method executions. Both techniques can be combined in a very flexible way. We show below only the parallel integration of the transfer method within the bank object by using the *organizer/clerk* paradigm.

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

The organizing bank delegates the computation of transfers to the clerk objects which act in parallel. This allows many transfers to be executed simultaneously, i.e., banks exhibit both internal concurrency and multiple threads. The organizing bank object keeps track of the active and inactive transfer (clerk) objects by holding the corresponding lists of identifiers aT and fT as attributes.

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.

References

1. G. Booch. *Object Oriented Design.* The Benjamin/Cummings Publishing Company, 1991.
2. G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language for Object-Oriented Development, Version 0.9, 1996.
3. M. Broy, R. Grosu, and C. Klein. Reconciling real-time with asynchronous message passing. Will appear in FME'97 Proceedings, September 1997.
4. R. Grosu, C. Klein, B. Rumpe, and M. Broy. State Transition Diagrams. Technical Report TUM-I9606, Technische Universität München, June 1996.
5. 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