TUM

INSTITUT FÜR INFORMATIK

Using the SysLab Method - A Case Study

Alexander Vilbig, Bernd Deifel, Sascha Molterer,
Andreas Rausch, Marc Sihling

TUM-I9751
Dezember 1997

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-12-1997-I9751-100/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Using the SysLab Method - A Case Study

A. Vilbig, B. Deifel, S. Molterer, A. Rausch and M. Sihling
Institut für Informatik
Technische Universität München
D-80290 München

15th December 1997

Abstract

In this paper we present a first case study which follows the SysLab method through the analysis phase of the software-engineering process. We specify the account management system of a bank using different description techniques defined by the SysLab method. This example illustrates a possible development process and allows an evaluation of the current status of the SysLab project.

Contents

1 Introduction

Within the SysLab project a set of description techniques (both graphical and textual) has been developed to represent different views of a software system [Thu97, Pae97, Het96, GKRB96]. Until now, the main effort has been to define a formal semantics for these descriptions and to integrate them into a mathematical system model [KRB96]. In this paper we present an example of a software development in its early phases using some of the description techniques available[1]. We also illustrate the relationships between them and describe a possible methodical development process.

This paper is structured as follows: In Section 2 we introduce the example of the bank account system, on which our case study is based. Section 3 describes the analysis phase of the software development process using the different description techniques of the SysLab method. Section 4 summarizes our development process and Section 5 concludes with an evaluation and ideas for further improvement of the SysLab method.

2 Informal Description of an Account Management System

The following text represents the informal description of the account management system:

Accounts are owned by customers. Customers can open and delete accounts, deposit and withdraw money and initiate a money transfer to other accounts (possibly at other banks). As a result of a transfer, money is withdrawn at most one day after the transfer form has been handed in by the customer. Eventually the money must be deposited to the other account and an acknowledgement must be sent to the bank initiating the transfer. It is not possible to decrease the account (due to a withdrawal initiated by a transfer or directly by the account owner), if there are earlier decrease requests (initiated by transfers or directly by the account owner) which have not been fulfilled.

Each bank manages accounts with numbers from 10000 to 99999. Numbers of deleted accounts can be reused. Withdraw and deposit is only allowed for existing accounts. Accounts may not be overdrawn. Banks identify each other also with numbers (100-999). The set of active banks might be changing. Transfer is only possible between active banks.

[1] We try to utilize graphical notations as far as possible, because in our opinion they are more intuitive than textual representations. However, some concepts have to be described with a textual syntax.

3 Analysis

3.1 Overview

During the analysis phase both developer and customer work together to specify the business requirements of the sytem. Therefore the documents of the analysis phase form the basis for communication between them. However, they also serve as the specification for the design phase of the development process. In order to satisfy both demands several different description techniques are used, depending on the method and the business requirements in question. The SysLab method offers three different system descriptions: a *Business Process Model (BPM)* [Thu97], an *Extended Entity Relationship Diagram (EERD)* [Het96] and a *System Interface Model (SIM)* [Pae97].

During the early analysis phase the customer identifies the relevant business processes, which are then modeled within different BPMs by the developer. These documents should be the subject of system validation by the customer.

Parts of the EERDs describing the static data model can be developed independently from business process modelling, although cross-connections to BPMs should be considered. A BPM contains data flow whose structure should be specified within an EERD. Figure 1 illustrates the dependencies between the different system descriptions.

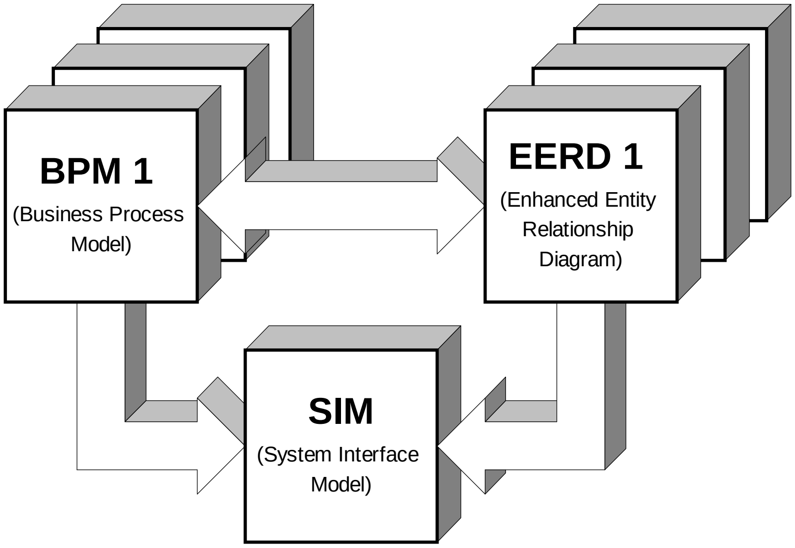

Figure 1: Schematic development process

The development of EERDs and BPMs is an iterative process. After starting with a top-down approach there is usually an alternation of bottom-up and top-down analysis until the model is sufficiently detailed.

At the end of the early analysis phase a given BPM illustrates the system dynamics from a single business-oriented point of view, whereas the EERDs present a static data-oriented view of the system. Now, during the late analysis phase, the SIM allows the developer to describe the whole system within a more formal, dynamic view based on its behaviour. It uses *State Transition Diagrams (STD)* [GKRB96] for both modeling dynamic system behaviour and formal specification of dynamic data constraints.

The SIM specifies the system interface through externally triggered services, which represent a group of activities within a role of a BPM [Pae97]. Therefore the concept of a role forms the connection between BPMs and the SIM.

The aforementioned general principles will be illustrated in the following sections which cover the development of the different system descriptions for the bank account system.

3.2 Busines Process Model

The first step in business process modeling is to identify processes which communicate with the environment of the system. We thereby identify the following processes:

1. A customer opens a new account,
2. a customer deletes an old account,
3. a customer deposits money to a given account,
4. a customer withdraws money from a given account,
5. a customer initiates the transfer of money to another account,
6. a foreign bank wants to transfer money to a local account.

As an example of developing a BPM, we concentrate on the transfer of money to another account. This is certainly the most complex and interesting business process, because it also includes communication with another bank.

We begin at a very abstract level by identifying the basic sequence of activities: First someone has to ask for a transfer. This event triggers the execution of the transfer and after its completion someone receives the result of the operation. Figure 2 depicts this sequence of activities in a graphical notation. The boxes represent the identified activities *issue transfer order*, *execute transfer* and *receive response*. The arrows between the activities represent their sequence in time as well as the flow of data between them.

The next development step comprises the refinement of the first abstraction level. Important activities (with respect to the business process) which are to be specified in more detail get subdivided into several sub-activities. In our example this procedure applies to the activities *execute transfer* and *receive response*.

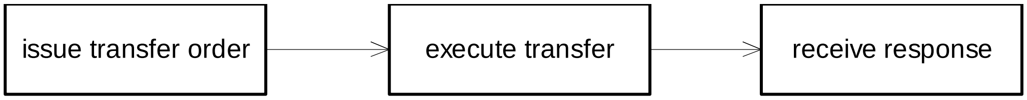

Figure 2: BPM "Transfer Money" - 1st Abstraction Level

Figure 3 shows the result of this refinement: The activity *execute transfer* is replaced by five sub-activities (*receive & check transfer order*, *enter transfer order*, *check order*, *do transfer* and *log transfer*), whereas *receive response* gets divided into four sub-activities (*receive confirmation*, *receive rejection*, *receive result* and *receive failure report*).

Due to a better traceability beetween the different abstraction levels we introduce a so-called *abstraction frame*. The concept of abstraction frames is a generic addition to the graphical representation of BPMs in [Thu97]. We propose to denote them with a broken line surrounding the involved activities. It groups several activities, which together form a refinement of an activity at a higher level of abstraction. In Figure 3 the abstraction frames indicate the refinement of *execute transfer* and *receive response* and are labelled as such.

The refinement of *receive response* becomes necessary when considering timing and possible actors who participate in this activity: *check order* delivers an immediate result about the validity of a given transfer order and leads to the execution of either *receive confirmation* or *receive rejection*. The activity *do transfer*, on the other hand, might need up to one day to determine the success of the transfer itself, as specified in the informal description of the bank account system (see Section 2). Consequently, a special activity *receive failure report* reacts to a possible failure within *do transfer*. The introduction of the activity *receive result* which follows both *receive confirmation* and *receive rejection* is motivated by the real-world business process: A customer initiates the money transfer (*issue transfer order*) and expects an immediate response from the bank (*receive result*) by interacting with a bank clerk who operates the bank account system (*receive & check transfer order*, *enter transfer order*, *receive confimation* and *receive rejection*).

Please note that although the SysLab method allows to explicitly associate activities with roles like customer or clerk at any time during business process modeling, we are only implicitly considering roles at this point. The mapping between activities and roles will be discussed in detail later in this section to clarify the methodical development process.

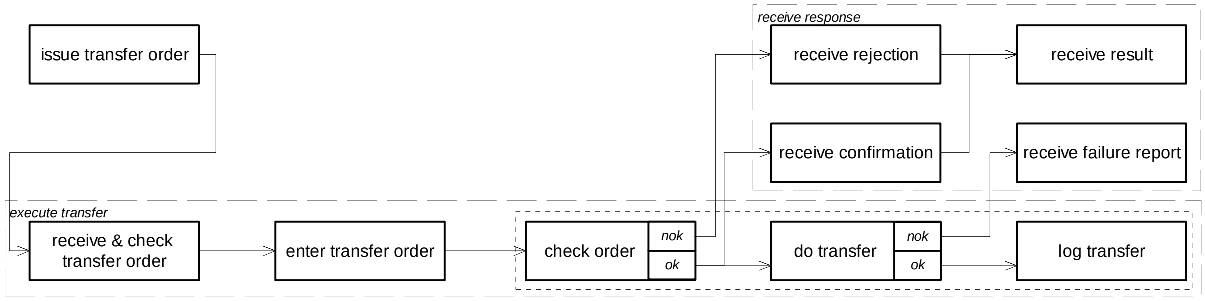

Figure 3: BPM "Transfer Money" - 2nd Abstraction Level

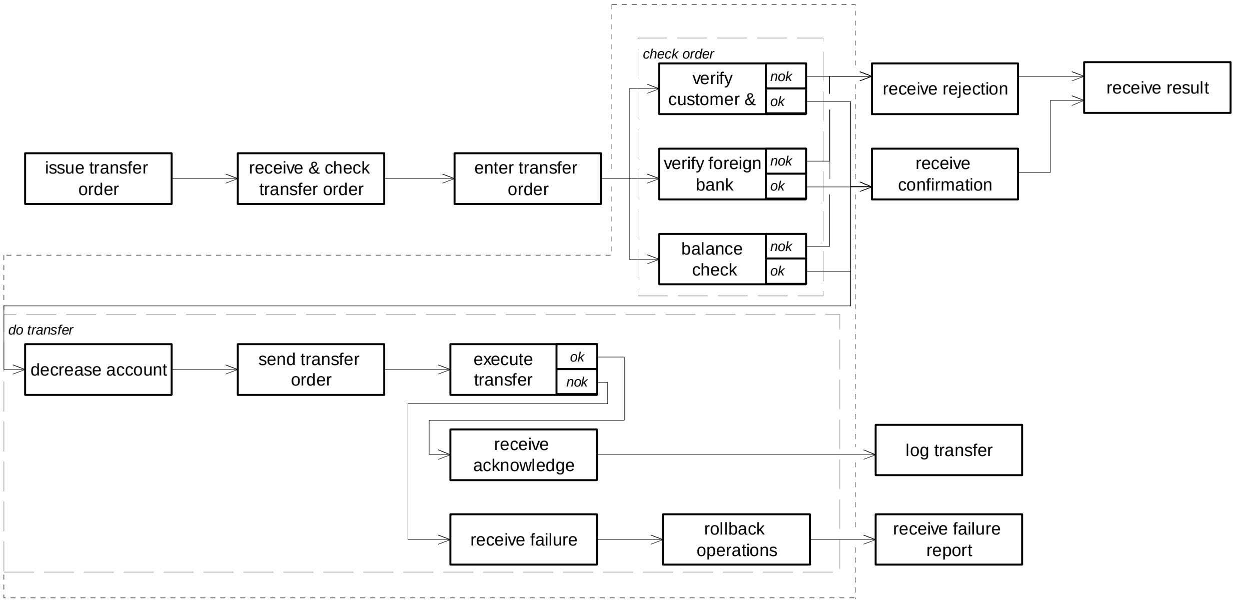

Figure 4: BPM "Transfer Money" - 3rd Abstraction Level

The bank account system itself is concerned with the three activities *check order*, *do transfer* and *log transfer*. They belong to a single transaction because the balance of the two participating accounts has to remain in a consistent state. The term *transaction* is used in this contex to describe the concept of atomic execution which leaves the system in a consistent state at all times, even if an activity is aborted due to internal errors. Elementary activities possess this property by definition, whereas several activities have to be explicitly grouped within a transaction frame. In the graphical representation we indicate this property by surrounding the activities with a dotted box.

Within the transaction we first check whether the transfer order is allowed to happen (*check order*). Certain integrity constraints on the data of the order have to be fulfilled and the order form itself must contain correct information. The exact nature of these constraints will be considered in a further refinement step. If the checks are passed, the actual transfer gets done (*do transfer*) and a log of the whole process is saved for later use (*log transfer*).

We proceed with a second refinement step, as the activities *check order* and *do transfer* need to be specified in more detail. The resulting third abstraction level of the BPM is shown in Figure 4.

The activity *check order* is subdivided into three logically independent activities *verify customer & account info*, *verify foreign bank* and *balance check*. The activity *verify customer & account info* has to verify the static information about the customer and the source account, e.g. whether the account exists and if the customer is the actual owner of the account he wants to access. *Verify foreign bank* checks the available data about the participating foreign bank (its existance and state of activity, for example) and eventually *balance check* ensures that the source account does not get overdrawn by the transfer [2].

The activity *do transfer* is refined into six sub-activities: *Decrease account* decreases the source account by the amount of the transfer order, *send transfer order* sends the transfer order to the foreign bank and *execute transfer order* increases the target account. Depending on the success of the transfer process at the foreign bank either *receive acknowledge* or *receive failure* are triggered. If the whole transfer has been successful, the transaction is finished. Otherwise the activity *rollback operations* assures that the system remains in a consistent state.

The next basic development step comprises the mapping of activities to roles, i.e. grouping the various activities from a logical point of view. This process of defining the roles is not restricted to a single BPM as the same roles may occur in different business processes. We do not further refine this BPM because our model also serves as a means of communication between the customer and the developer of the system. So we suggest to use only a small number of abstraction levels (up to a maximum of about four levels) in order to avoid the danger of going too deep into the details and thereby losing the understanding with the customer.

Although the specification of the activities within the BPMs is generally independent from the roles they are associated with during this development step, it may sometimes be useful to take roles into account if they are already known. An example of this approach is the refinement of the activity *receive response* into *receive confirmation*, *receive rejection* and *receive result* (see Figure 3): The former two activities are related to a bank clerk who operates the bank account system whereas the latter belongs to the customer - two different roles which are obviously part of the business process in question.

In our example we identify the four roles *Customer*, *Bank Clerk*, *Bank Account System* and *Foreign Bank*: The *Customer* provides the orders, the *Bank Clerk* represents the interface between the *Customer* and the *Bank Account System*, whereas the main functionality of the system is contained within the roles *Bank Account System* and *Foreign Bank*.

The mapping of activities to these roles is illustrated by the exemplary BPM "Transfer Money" as seen in Figure 5. To associate roles with activities, we extend the graphical representation of the corresponding BPM by adding a role identifier beneath every activity. The activities *issue transfer order* and *receive result* belong to the role *Customer* as the initiator of this business process. The activities *receive & check transfer order*, *enter transfer order*, *receive rejection*, *receive confirmation* and *receive failure report* have been mapped to the role *Bank Clerk*. The role *Foreign Bank* contains only the activity *execute transfer order*, whereas all remaining activities belong to the role *Bank Account System*.

We introduce an alternative graphical representation of the role-activity mapping in Figure 6. The roles are ordered by rows which are separated by horizontal dotted lines. The activities are ordered from left to right according to their sequence in time and placed into

[2] Note that the confirmation of a transfer order requires **all** checks to be passed whereas a **single** failed check results in a rejection. Currently the SysLab method provides no means to integrate these semantics into a BPM.

the row of the role they belong to. This layout clarifies the logical sequence of the business process and emphasizes the interaction between the participating roles.

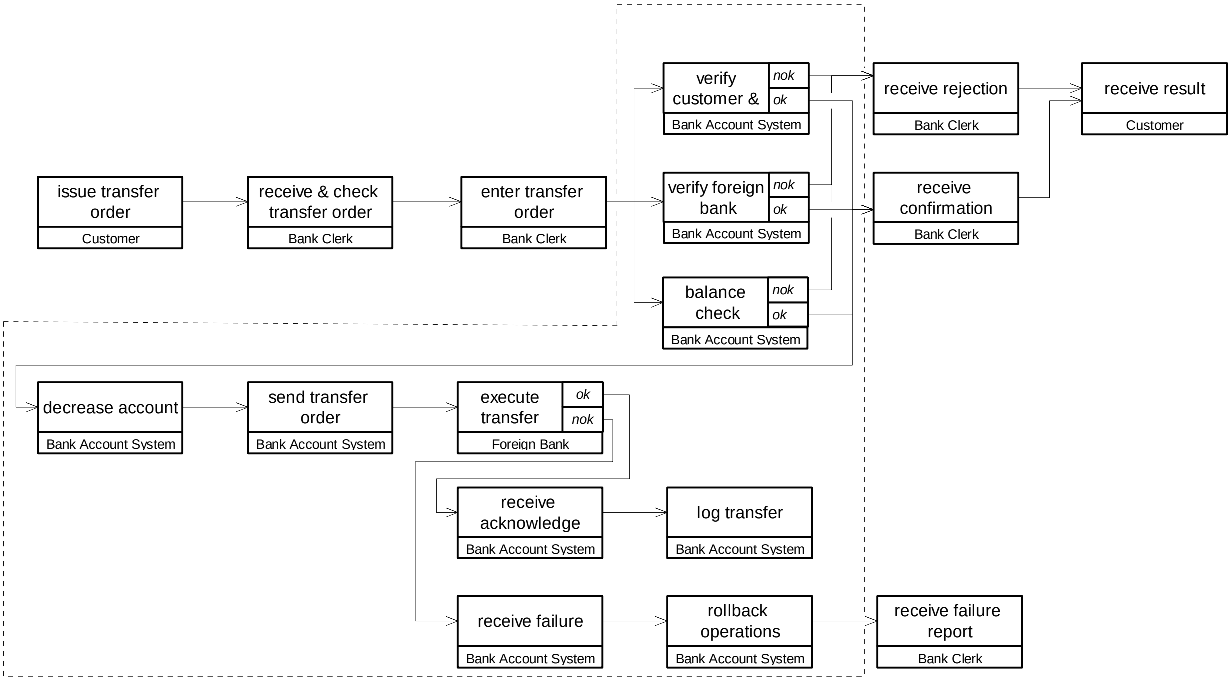

Figure 5: BPM “Transfer Money” - 3rd Abstraction Level with Roles

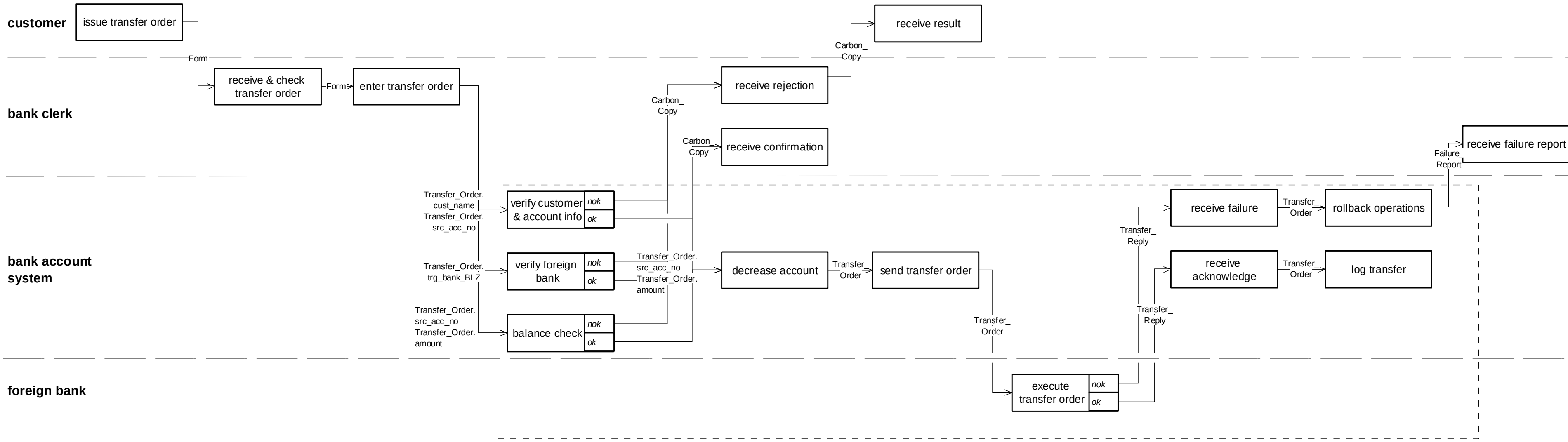

Figure 6: BPM "Transfer Money" - Role-Activity Mapping

During the development of the different BPMs, we are also able to identify the data flow between certain activities. In figure 6 this information is captured by labelling the arrows with an identifier for the data which flows from one activity to the next. The activity *enter transfer order*, for example, provides the activity *verify customer* & *account info* with the name of the customer and the number of the source account (*Transfer_Order.cust_name* and *Transfer_Order.src_acc_no*). Knowledge gained by identifying the data flow during business process modelling should be used when specifying the data model. This principle will be illustrated in the following section.

3.3 Data Model

This section illustrates the use of EERDs, a description technique provided by the SysLab method to model the static data of a system [Het96].

The first development step comprises the definition of data entities (see Figure 7). As already mentioned in Section 3.2, we identify some of them by looking at the data flow within the different BPMs. The BPM "Transfer Money", for example, motivates the introduction of a data entity *Transfer_Order* because of the data flow between the activity *enter transfer order* and the activities *verify customer* & *account info*, *verify foreign bank* and *balance check* (see Figure 6). In a very similar way, the data entities *Withdraw_Order* and *Deposit_Order* can be obtained from the data flow within the BPM of the processes "Deposit Money" and "Withdraw Money". Independent from business process modeling, we also specify data entities which correspond to relevant real-world entities like *Customer*, *Account* and *Bank*.

After the definition of data entities we derive the relationships between them. We suggest to establish these relationships between data entities before specifying any of their attributes. This prevents the erroneous inclusion of other entities as "foreign-key attributes" within the entity they are related to. Again, this can be done according to real-world relationships. As an example consider the relationship *owns* between the entities *Customer* and *Account* in Figure 7: A customer may own zero or more accounts at one bank. We indicate this cardinality by adding the label (0,*) at the customer's side of the relationship. A given account, on the other hand, is owned by exactly one customer. Therefore we attach the label (1,1) to the account's side of the relationship.

During the next development step of the data model we define the specific attributes of each data entity (see Figure 8). They are listed in a separate table with their name, type and an indication whether they are optional or part of the key. This notation closely resembles traditional ER diagrams. Some basic attributes, like account number for example, can be derived from the corresponding real-world entities. Additional attributes may be introduced later when specifying the different services during the development of the system interface model (see Section 3.4). An example of this approach is the attribute *reply* of the entity *Transfer_Order*, which is utilized by the service *transfer_money_external* (see

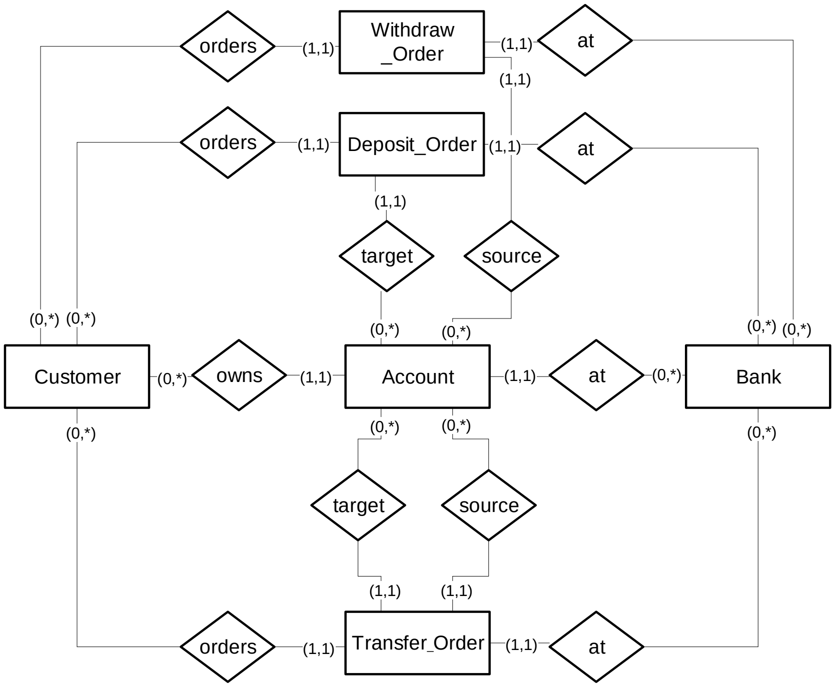

Figure 7: EER-Model of the Bank Account Management System (a)

Figure 9) for communication with a foreign bank[3].

Apart from defining data entities and attributes, we also have to specify certain data types on which the attributes are based. We use the syntax of MINI-Spectrum [Het96] for an algebraic specification of these data types. This allows for an abstract expression of possible constraints on the data, a principle which is illustrated by the definition of different kinds of numbers in Figure 8: First we define a basic data type *Nb* which is derived from natural numbers *Nat* but only supports the relational operators $\leq$ and $<$. We thereby preclude the (possibly dangerous) use of arithmetic operations in the context of account and bank numbers. From *Nb* we then derive the data types *AccountNb* and *BankNb* by adding additional axioms which determine their allowed interval (as required for account numbers and bank numbers by the informal system description in Section 2).

[3] Note that we omit the specification of the entities *Withdraw_Order* and *Deposit_Order* for the sake of clarity.

```
Nb = [
data Nb = Nat

≤.,.<.: Nb x Nb → Bool;
]

Money = [
data Money = Nat;
]

State_ Reply = [
data State_ Reply = Success| Failure;
]

Transfer_Reply = [
datastatus = State_Reply,
time _ assurance = Time;
]
```

```
AccountNb = [
data AccountNb = Nb;

axioms ∀x: Nb in
∀x:10000 ≤ x and x ≤ 99999
endaxioms
]

BankNb = [
data BankNb = Nb;

axioms ∀x: Nb in
∀x:100 ≤ x and x ≤ 999
endaxioms
]
```

Entity Type	Customer		
Attribute	**Type**	**Opt**	**Part of Key**
name	String		x
address	Adr		

Entity Type	Account		
Attribute	**Type**	**Opt**	**Part of Key**
number	AccountNb		x
money	Money		

Entity Type	Bank		
Attribute	**Type**	**Opt**	**Part of Key**
BLZ	BankNb		x
name	String		
address	Adr		

Entity Type	Transfer_Order		
Attribute	**Type**	**Opt**	**Part of Key**
number	Nb		x
cust_name	String		
src_acc_no	AccountNb		
src_bank_BLZ	BnkNb		
trg_acc_no	AccountNb		
trg_bank_BLZ	BankNb		
order_time	Time		
reply	Transfer_Reply		
amount	Money		

Figure 8: EER-Model of the Bank Account Management System (b)

3.4 System Interface Model

After identification of the participating roles during business process modelling (see Section 3.2), we would like to describe the way they behave and interact with each other. This is done by specifying the *services* a given role offers to other roles or to the environment, thereby defining its *interface* [Pae97].

The transition from a set of BPMs to a specification of the system interface is very much an intuitive process. In a first approach, every business process modeled by a BPM may be represented within a single service (like "Open Account", for example). However, the execution of certain services requires communication with other roles (e.g. during the transfer of money a foreign bank has to be contacted). This communication has to be done through calls to services of partner roles, thereby causing the need to introduce further services. On the other hand, one should try to specify a lean system interface, covering only the basic business functionality of each role.

In our example, we are mainly interested in the interface of the role *Bank Account System*, which offers numerous services regarding the management of bank accounts. In a first development step we derive the services of its interface from the different requests one can pose to the bank account system (like “Open Account”, “Close Account” etc.). To keep this example concise, however, we will concentrate in this section on the service “Transfer Money”, which also includes internal communication with other roles (see Figure 6).

Please note that the SysLab method distinguishes between *interaction services* and *transaction services* [Pae97]: An interaction service is not allowed to change the data state of its role and is therefore limited to communication actions only. A *transaction service*, however, may process the role’s data but requires exclusive access to this data during its execution. The most important conclusion is that a transaction service cannot call other transaction services within the same role. As a consequence we have to specify three different services:

- *transfer_money_external*, which is basically the transfer of money as specified during business process modelling,
- *receive_money_transfer*, which takes care of a money transfer initiated by a foreign bank and
- *transfer_money_internal*, which handles a money transfer within the same bank.

The specification of the interface of a given role begins with a header which describes its attributes and its communication partners. We use a textual notation as proposed in [Pae97] to describe the attributes and communication partners of a role (or service), whereas state transition diagrams define the behaviour of a service concerning the role’s data state. STDs allow us to show the way a service acts on the states by introducing pre- and postconditions. If the systems resides in a state A, a transition from A to B exists and the according precondition holds, then the systems changes to state B and the postcondition holds. Note that several transitions with a fulfilled precondition may exist. In this case one of them is chosen in a nondeterministic way.

The interface of the role *Bank Account System* is specified as:

```
role bank_account_system = {

   attributes              customers : Set Customer,
                           accounts  : Set Account,
                           banks     : Set Bank
   partner                 clerk     : Bank_Clerk
}
```

Transfer of money to a different bank

The transaction service *transfer_money_external* is considered with a bank-to-bank transfer (i.e. the target account is located at a different bank as the account the money is withdrawn from) and has to communicate with a foreign bank. It either accepts an acknowledgement from the foreign bank (including an assurance about the time needed to finish its part of the transfer) or reports a failure to the role *Bank Clerk*. In order to access relevant service data several times during its execution, *transfer_money_external* uses the local attributes a, b and c.

```
   transaction service transfer_money_external = {

      partner         log       : Process_Logger,
                      fb        : Foreign_Bank

      attributes      a         : Account,
                      c         : Customer,
                      b         : Bank

      trigger input   order     : Transfer_Order from clerk on input_order
              input   reply     : Transfer_Reply from fb on input_reply

              output  o1        : Transfer_Order to fb on output_trans_order,
              output  o2        : Transfer_Order to log on output_log,
      final   output  o3        : Failure_Report to clerk on output_report
}
```

Figure 9 shows the corresponding state transition diagram which describes the behaviour of this service[4].

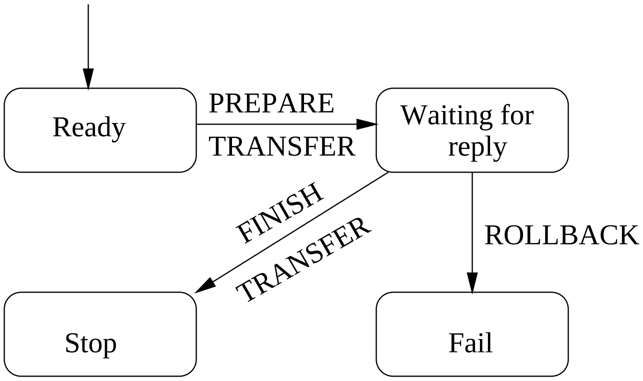

Figure 9: STD “Transfer Money External”

The necessary checks at the local bank are expressed in the precondition of the transition PREPARE_TRANSFER (i.e. account exists, customer exists, customer is owner of this account, etc.). The transfer order is received on the input channel and an according request to the foreign bank is sent on the output channel.

[4] The arrow connecting two states is labelled with the name of a transition. The transition itself is shown in a box below to keep the diagram simple.

PREPARE TRANSFER		
pre	:	$\exists$ lc $\in$ customers, $\exists$ la $\in$ accounts, $\exists$ lb $\in$ banks . order.src_acc_no = la.number $\wedge$ order.cust_name = lc.name $\wedge$ order.trg_bank_BLZ = lb.BLZ $\wedge$ owns(lc , la) $\wedge$ la.money $\geq$ order.amount $\wedge$ a = la $\wedge$ b = lb $\wedge$ c = lc
in	:	input_order?[order]
out	:	output_trans_order![order]
post	:	

Upon receiving a positive reply (including an assurance about the time needed to process the transfer) from the foreign bank within the transition FINISH_TRANSFER, the local bank logs this process for later use. The decrease of the account at the local bank is expressed within the postcondition.

FINISH TRANSFER		
pre	:	reply.status = Success $\wedge$ reply.time_assurance $\leq$ 1 day
in	:	input_reply?[reply]
out	:	output_log![order]
post	:	a'.money = a.money - order.amount

If the foreign bank cannot process the transfer successfully, the transition ROLLBACK expects a corresponding reply and reports a failure[5]. The local account is not decreased, because the postcondition of this transition is empty.

ROLLBACK		
pre	:	reply.status = Failure
in	:	input_reply?[reply]
out	:	output_report![Failure]
post	:	

Transfer of money within the same bank

The money transfer within the same bank requires a special service *transfer_money_internal*, because we cannot call the service *receive_money_transfer* within the same role. The behaviour of *transfer_money_internal* with respect to the checks necessary and the effect on the participating accounts is nevertheless very similar to *transfer_money_external*.

```
transaction service transfer_money_internal = {

   partner         log       : Process_Logger

   attributes      a1, a2    : Account,
                   c         : Customer
```

[5] To keep this example simple we assume `Failure Report` to be of the same type as `Reply.status`.

```
      trigger input   order      : Transfer_Order from clerk on input_order

              output  o1         : Transfer_Order to log on output_log
      final   output  o2         : Failure_Report to clerk on output_report
}
```

Figure 10 shows the corresponding STD.

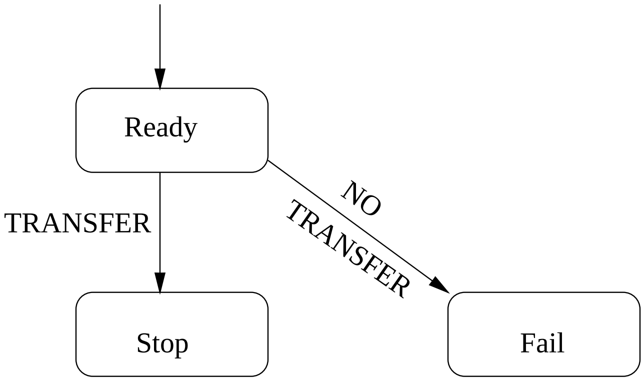

Figure 10: STD "Transfer Money Internal"

As the transfer is done internally, the transition TRANSFER shows its result very clearly in the postcondition (i.e. both accounts are changed by the same amount).

<table>
<tr><th colspan="3">TRANSFER</th></tr>
<tr><td>pre</td><td>:</td><td>$\exists$ lc $\in$ customers, $\exists$ la1,la2 $\in$ accounts . (order.src_acc_no = la1.number) $\wedge$ (order.trg_acc_no = la2.number) $\wedge$ (la1 $\neq$ la2) $\wedge$ (order.cust_name = lc.name) $\wedge$ (owns(lc,la1)) $\wedge$ (la1.money $\leq$ order.amount) $\wedge$ c = lc $\wedge$ a1 = la1 $\wedge$ a2 = la2</td></tr>
<tr><td>in</td><td>:</td><td>input_order?[order]</td></tr>
<tr><td>out</td><td>:</td><td>output_log![order]</td></tr>
<tr><td>post</td><td>:</td><td>a1'.money = a1.money - order.amount $\wedge$ a2'.money = a2.money + order.amount</td></tr>
</table>

In case one of necessary checks fails (indicated by "else" in the precondition), the transition NO_TRANSFER reports a negative result.

<table>
<tr><th colspan="3">NO TRANSFER</th></tr>
<tr><td>pre</td><td>:</td><td>else</td></tr>
<tr><td>in</td><td>:</td><td>input_order?[order]</td></tr>
<tr><td>out</td><td>:</td><td>output_report![Failure]</td></tr>
<tr><td>post</td><td>:</td><td></td></tr>
</table>

Processing of an incoming request to transfer money

The transaction service *receive_money_transfer* describes the interface to other banks, which like to transfer money to a local account (in a way it also represents the expected behaviour of the role *Foreign Bank* as seen in Figure 6).

```
   transaction service receive_money_transfer = {

      partner         log       : Process_Logger,
                      fb        : Foreign_Bank

      attributes      a         : Account

      trigger input   request   : Transfer_Order from fb on input_request

              output  o1        : Transfer_Order to log on output_log
      final   output  o2        : Transfer_Reply to fb on output_reply
}
```

Figure 11 shows the corresponding STD.

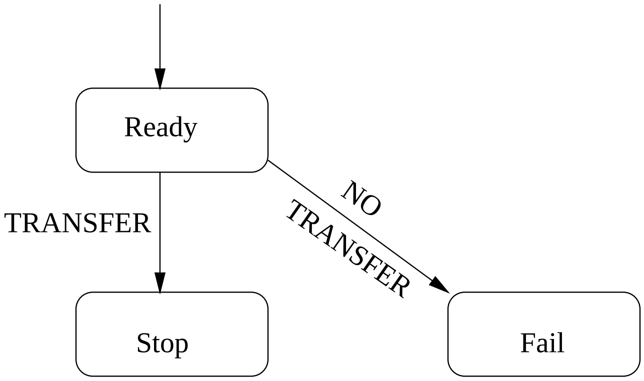

Figure 11: STD "Receive Money Transfer"

Upon reception of a transfer order, the target account at the local bank has to be verified. Furthermore the bank account system has to assure that the transfer request will be processed within the given time[6]. These constraints are expressed within the precondition of the transition TRANSFER. When both conditions are met, the target account is increased by the given amount, an acknowledgement is sent to the calling bank and the process itself is logged.

[6] We assume the existance of an auxiliary function *current_time()* which returns the current system time.

TRANSFER		
pre	:	$\exists$ la $\in$ accounts . (request.trg_acc_no = la.number) $\wedge$ (current_time() - request.order_time $\leq$ 1 day) $\wedge$ a = la
in	:	input_request?[request]
out	:	output_reply![Transfer_Reply(Success, current_time())], output_log![request]
post	:	a'.money = a.money + request.amount

In case one of the conditions doesn't hold, the transition NO_TRANSFER sends a negative reply to the calling bank.

NO TRANSFER		
pre	:	else
in	:	input_request?[request]
out	:	output_reply![Transfer_Reply(Failure, current_time())]
post	:	

The specification of the three services *transfer_money_external*, *transfer_money_internal* and *receive_money_transfer* constitutes the system interface of the bank account system concerned with transfering money from one account to another. Using this rather high level of abstraction allows us to provide a concise specification of the dynamic data constraints within the pre- and postcondition of the STD transitions. We do not specify *how* these constraints are to be fulfilled - clearly a task belonging to the design phase.

4 Summary

In this section we want to summarize our proposed development process while using the SysLab method during the analysis phase of the software engineering process. As stated previously, the relevant business processes and occurring data entities can generally be modeled in parallel, although cross-connections should be considered where necessary (i.e. modeling of data entities which are attached to data flows in a BPM). Refinement of the different BPMs allows for an iterative development process as more and more in-depth knowledge about the business processes gets incorporated into the model. After reaching a sufficiently high level of detail, the various activities of a given business process are logically assigned to different roles within the system. These roles encapsulate the basic business functionality and define the scope of the system in question.

The next development step defines the system interface model by specifying the available services within each role identified during business process modeling. In our example almost every business process corresponds to a single service within the role *Bank Account System*, but this is obviously not a general principle. Deriving a system interface model from a set of business process models is not straightforward and depends on the skill and experience of the developer.

Once the services have been identified, state transition diagrams are used to describe their behaviour and to express the existing dynamic data constraints. Further refinement of the system interface model to separate independent data into different roles and services acting upon the data leads to the design phase. Although the description techniques of the SysLab method cover this transition, the design phase itself is beyond the scope of this case study.

5 Conclusion

In this section we want to discuss problematic areas of the development process and to present ideas for further improvements. During the course of our case study we noticed the following problems and deficiencies within the SysLab method:

- The business process model does not define a precise semantics for transaction error handling, i.e. a given BPM does not specify how the system reacts to a failure of a certain transaction. We feel that technical failures, like a broken communication channel, may be ignored at this stage of the development process, whereas business related failures, like a negative reply from the foreign bank, should be modeled explicitly (e.g. by introducing a special rollback activity as an extension to the existing notation).
- Although it is possible to specify a distributed transaction which extends across several roles (e.g. the transfer of money between two banks) during business process modeling, the system interface model only allows for transaction services within a single role. These orthogonal semantics may cause severe problems when deriving a system interface model from the relevant BPMs.
- We feel that the concept of mutual exclusive transaction services which lock all of the role data during their execution is too strict, because it hinders the development of a lean system interface (cf. the split of the service *transfer_money* into *transfer_money_internal* and *transfer_money_external*) and the reusability of internal services. We propose to allow for a locking of role data on a smaller scale and the possibility to specify synchronisation partners (e.g. by further classifying a service through language constructs like "operates on" and "synchronizes with").
- The process of further refining the original system interface model is currently not clearly defined within the SysLab method. It is not obvious how consistency between different levels of abstraction can be assured (e.g. when splitting a transaction service of a role into several services of different roles). The suggested data-driven refinement of roles leads directly into the design phase, because explicit communication structures concerning data exchange between the roles have to be modelled. We think the developer should have more methodical assistance by the SysLab method during this transition from the analysis to the design phase.
- The iterative development of the various BPMs requires many changes to the graphical representation and generates a large amount of specification documents for a

non-trivial system. We therefore recognize the need for a tool which provides support in creating, editing and maintaining these documents.

Although further work concerning tool support, the transaction concept and the means to refine a system interface model seems necessary, we demonstrate in this paper the feasibility to specify a real-world system using the SysLab method. Of course the development of a complete and comprehensive methodology is still very much work in progress. We hope to have contributed to this process with our case study.

Acknowledgements

We thank Herbert Ehler, Barbara Paech, Bernhard Rumpe, and Veronika Thurner for discussions and for reading draft versions of this paper.

References

[GKRB96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State transition diagrams. TUM-I 9630, Technische Universität München, 1996.

[Het96] Rudi Hettler. Description techniques for data in the syslab method. TUM-I 9632, Technische Universität München, 1996.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical system model for distributed information processing systems - syslab system model. In E. Naijm and J. Stefan, editors, *FMOODS'96, Formal Methods for Open Object-based Distributed Systems*, pages 323–338, 1996.

[Pae97] Barbara Paech. A framework for interaction description with roles. Technical Report TUM-I 9731, Technische Universität München, Institut für Informatik, June 1997.

[Thu97] Veronika Thurner. A formally founded description technique for business processes. Technical Report TUM-I 9753, Technische Universität München, Institut für Informatik, December 1997.