A Formally Founded Description Technique for Business Processes

Veronika Thurner
Department of Computer Science
Technical University of Munich
80290 Munich, Germany
thurner@informatik.tu-muenchen.de

Abstract

As a means of modeling typical system behavior, we present a description technique for business processes derived from data flow nets and provide it with a formal semantics based on functions and their composition. The formalism features black box and glass box view and a concept of refinement. As it is intuitively understandable and formally well founded, the notation is equally adequate for the needs of application domain experts and system engineers in requirements engineering[1].

1. Introduction and motivation

Many approaches to requirements engineering involve a detailed modeling of key aspects such as system structure, data or behavior. These models are the basis for communication between expert users and system analysts, and the foundation for system design and implementation later on in the system development process. Thus, they are a decisive factor for software quality and correction costs [7].

A basic idea of system modeling is the reduction of complexity by focussing on a single system view and only a small set of system aspects at a time. In behavior modeling, a first step consists of the analysis and documentation of typical system behavior in an exemplaric way. Thus, single system runs or scenarios are examined.

In many approaches to behavior modeling that deal with exemplaric system behavior, scenarios are employed for documenting the interaction of objects, system components or organizational units (see, for example, message sequence charts [13], interaction diagrams of Booch [1], sequence diagrams of UML [2], or process object schemes [9]). Thus, scenarios are often arranged according to structural system aspects. The resulting behavior model is intermingled with, and dominated by, the system architecture. Consequently, constraints that are not due to any causal dependencies originating from the behavioral model itself are added which restrict the order of process execution and thus the possible amount of parallelism. Other modeling techniques, such as activity diagrams in [2] or the process notion of [14], already include aspects of system state. However, although this integrated modeling of several system aspects at a time might still work with small examples, it quickly turns to be difficult and hard to handle as complexity increases.

In contrast to this, we suggest a task oriented approach to behavior modeling. Focussing on the system's major tasks, we develop a business process model that is cross functional to the underlying structural organization and which includes the relevant behavioral context of the system's environment. Methodically, we start by modeling single exemplaric system runs. As application domain experts find it comparatively easy to relate their share of activities in system behavior when following a specific example process, this approach is extremely helpful for capturing and discussing the users' view on system behavior and requirements.

We introduce a description technique that supports behavior modeling independantly from organizational or geographical boundaries. It documents causal dependencies between processes and their execution that are due to the exchange of messages and events between processes. However, no artificial sequentialization or other constraints on the order of process execution are introduced, thus allowing for a maximum of possible parallelism in process execution.

Our modeling technique includes black box and glass box view as well as a refinement mechanism which supports behavior modeling across different levels of abstraction. Furthermore, we reduce redundancy in our process model by introducing process types.

To support unambiguity, we introduce a formal semantics based on functions and their composition, which provides a flexible modeling and abstraction mechanism focusing on data dependencies rather than on partially ordered sequences of event exchanges between objects. Thus it supports our modeling intentions stated above.

[1]This work was supported by the Bayerische Forschungsstiftung.

2. Concrete syntax

We model business processes as a key aspect of system behavior. By business process, we denote a set of tasks that consume and produce service results such as data or material [15]. The exchange of these service results defines causal relationships between business processes and tasks.

We focus on exemplaric system behavior, modeling the execution of process instances. As multiple instances of a single process may occur within the model of a system, we introduce process types for reducing redundancy. Based on the set of defined process types, instances of these types can be composed into process networks which desribe exemplaric sequences of system behavior. A process type defines the interface, internal behavior and refinement structure, which are common to all of its instances.

Each of these aspects corresponds to a view on a process type. The black box view describes a process type's interface or functionality. The manipulation of data during the execution of a process is dealt with in the glass box view. Finally, the refinement view defines the decomposition of a single process type into a network of process types of finer granularity.

We provide a notation consisting of graphical and textual elements. The basic graphical aspects were derived from data flow nets [8]. Moreover, we incorporate and enhance some notation ideas taken from the modeling language GRAPES V3 [17]. Textual aspects of our notation are provided in extended Backus-Naur form as introduced in [6]. The non-terminals ⟨process-type⟩, ⟨text⟩, ⟨function⟩ and ⟨predicate-expression⟩ are not specified any further within this work.

2.1. Black Box View

Next to the process type's name, the black box view specifies the process type's signature in terms of bundles of typed input and output ports, as evident and relevant on the current level of granularity. Both process type and port names must be unique throughout a model. For reasons of readability, in the examples throughout this paper we name a process type's input and output ports by in and out, respectively, followed by a raised suffix denoting an abbreviation of the associated process type's name. Different input or output ports of a single process type are distinguished by a numbering index. Figure 1 shows an example of the graphical representation of a process type.

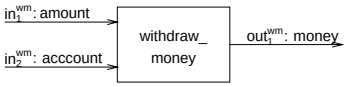

Figure 1. Process type with ports.

When a process type is refined into a process network in a subsequent step of development, the refined definition may be supplemented by additional input and output ports which will not be added to the model's more abstract levels.

With regard to the distribution of processes to execution components later on in the development process, roles may be associated optionally with process types. The role identifier is designated at the lower border of the process type symbol, as shown in Figure 2.

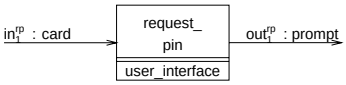

Figure 2. Process type and role.

External processes are executed outside of the system under consideration. As illustrated in Figure 3, we denote external processes by a dashed process type symbol. Often, a process types binding is implicitly determined by the associated role. However, for methodical reasons, it is helpful to allow an explicit declaration of the process binding. By default, process types are assumed to be internal.

Figure 3. Process type with external binding.

For the black box view of a process instance, the name of the process type is preceded by the instance identifier in a separate section of the process symbol (confer Figure 4). The identifiers of process instances are unique throughout the whole model of the system.

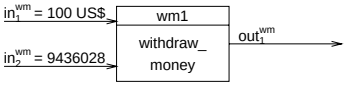

Figure 4. Process instance and input values.

2.2. Glass Box View

Whereas the black box view defines a process type's input/output behavior, the glass box view describes the internal manipulation of data during the execution of a process instance as well as pre- and postconditions. The modeling of nondeterminism is supported.

The glass box view documents any information on the computation scheme that derives output data from input data, which is known at the current stage of the modeling process. In the computation scheme, input and output data are parameterized by the corresponding port names.

When executing an instance of a process type, specific values are assigned to its input ports, respecting the port types which are defined in the corresponding black box view. Output values are determined by executing the computation scheme specified in the glass box view on the specific values assigned to the input ports.

In our notation, we do not introduce any graphical symbols for specifying the glass box view, as we do not expect an adequate gain in readability here. Rather, we use a textual notation. Depending on the degree of knowledge that is available on the computation scheme, it is described either mathematically by specifying a function, or as text enhanced by some mathematical elements. Pre- and post-conditions are specified as predicate expressions.

glass box process type ⟨process-type⟩ = {
  computes ⟨text⟩ | ⟨function⟩
  pre ⟨predicate-expression⟩
  post ⟨predicate-expression⟩
}

For the glass box view of our example process type $withdraw\_money$, we employ a textual representation with some mathematical elements.

glass box process type $withdraw\_money$ = {
  computes $out_1^{wm} = f_1^{wm}(in_1^{wm}, in_2^{wm})$, with
  $f_1^{wm}(in_1^{wm}, in_2^{wm}) =$
  $$= \begin{cases} \text{requested amount of money} \\ \quad \text{if requested amount} < 400 \\ \text{requested amount of money} \\ \quad \text{if requested amount} > 400 \text{ and} \\ \quad \text{account deposit} \geq \text{requested amount} \\ \text{no money} \\ \quad \text{if requested amount} > 400 \text{ and} \\ \quad \text{account deposit} < \text{requested amount} \end{cases}$$
  pre $true$
  post $true$
}

2.3. Refinement View

The refinement view describes how a process type of coarse granularity is refined by a process network [5], constructed from process types of finer granularity. By linking the output port of one process to the input port of another process, we connect processes via channels denoted by the pair of ports $(outport, inport)$. We restrict our model to acyclic structures.

Furthermore, the refinement view specifies how input and output ports of the process type on the coarser level of granularity are mapped on the input and output ports of the refining process network. In a correct refinement, all the ports on the coarser level of granularity are redirected to corresponding ports on the refining level. Consequently, the refining process network contains at least the equivalents to the ports of the coarse grain process type.

Figure 5 illustrates the refinement of process type $withdraw\_money$ from our example in Figure 1. Operator $*$ symbolizes the duplication of the message assigned to a port and the redirection of the copies.

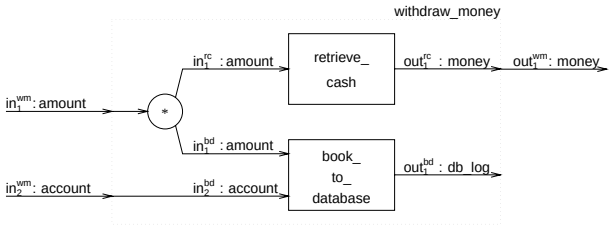

Figure 5. Process type refinement.

Within a refining process network, a single process type may occur multiply. However, in our graphical representation these different occurances are distinguished by their geometrical position within the diagram. Thus, the structure of connecting channels is unambiguous as well.

When a new instance of a process type is created, it is assigned an identifier which is not yet assigned to any other process instance within the model. Furthermore, if a refining process network is defined for this process type, a corresponding refining network of process instances is created.

3. Semantics

The semantics of our description technique for business processes is based on functions and their composition. Compositionality is necessary for formalizing refinement. We assign a function with adequate input/output signature to each process type, which formalizes the process type's computation scheme. This usage of function composition is related to computation forms, discussed e.g. in [4].

Some existing approaches to process modeling define a semantics based on event traces (for example [12]). The technique of event traces may be applied efficiently for modeling process networks where the execution of processes is partially ordered.

In our notion of processes, however, we also allow modeling on a more abstract level which is especially helpful at the beginning of the modeling process, when the modelers' understanding of business processes is still rather vague. We achieve this by focussing on process causality due to data dependencies. A data flow from a process A to its successor process B indicates that at some time during its processing, process B receives input from process A. However, we do not restrict process execution by specifying any relationship between the end of the execution of process A

and the beginning of process execution of B, thus allowing flexible refinement possibilities of A and B as well as their interaction at later stages in the modeling process.

This concept of loose dependencies is not supported by other popular techniques for behavior modeling, such as petri-nets [16] or statecharts [11], which imply an ordering of process execution rather than of message exchange. The concept of interacting processes in [15] introduces another approach to modeling process interaction without ordering process execution, but does not provide a formal semantics.

In the following, let PT denote a set of identifiers of process types, PI a set of identifiers of process instances, P a set of identifiers of ports, F a set of function symbols, and S denote a set of data sorts.

3.1. Semantics of an isolated process type

The black box definition of a process type specifies its typed input/output functionality, which on the level of semantics corresponds to the signature of the function that is associated with a process type. Thus, with a process type $p \in PT$ we associate a function $f^p \in F$ with functionality

$$\textbf{fct}\, f^p : s_{in_1^p} \times \ldots \times s_{in_{i_p}^p} \rightarrow (s_{out_1^p} \times \ldots \times s_{out_{o_p}^p}),$$

where $s_{in_1^p}, \ldots, s_{in_{i_p}^p} \in S$ and $s_{out_1^p}, \ldots, s_{out_{o_p}^p} \in S$ denote the sorts associated with input ports $in_1^p, \ldots, in_{i_p}^p \in P$ and output ports $out_1^p, \ldots, out_{o_p}^p \in P$ of process type p.

The body of function f^p corresponds to the computation method that is given by field **computes** in the glass box definition of a process type. Precondition **pre** of the process type is incorporated in the function body as well. On the level of semantics, process execution is equivalent to the evaluation of the associated function on specific input values.

With our example process type $withdraw\_money$ from Figure 1, we associate a function

$$\textbf{fct}\, f^{withdraw\_money} : amount \times account \rightarrow (money).$$

So far, we assumed our processes to be deterministic. However, the semantics can easily be generalized to cover nondeterministic processes as well, by associating with a process type not a single function, but a set of functions. For every single execution of an instance of this process type, we nondeterministically choose one function of the associated set, which is then executed to compute the result in a deterministic fashion.

3.2. Semantics of a process network

Via the concept of refinement, a process type is represented in more detail by a network of process types of finer granularity that are linked via some of their input and output ports. On the level of semantics, process type refinement corresponds to expressing a function by the composition of other functions. When the refinement level contains supplementary input and output ports that were not present on the coarser modeling level, a restriction of the input/output functionality of the composition of refining functions is necessary as well.

In Figure 5, our example process $withdraw\_money$ from Figure 1 is refined into a process network which is constructed from the process types $retrieve\_cash$ and $book\_to\_database$. With the refining process network, we associate function $f^{ref(withdraw\_money)}$ with signature

$$\textbf{fct}\, f^{ref(withdraw\_money)} : amount \times account \rightarrow (money \times db\_log).$$

This signature of the refining function may be restricted to the signature of the original function $f^{withdraw\_money}$ as follows.

$$f^{withdraw\_money} = f^{ref(withdraw\_money)}|_{1,2 \rightarrow 1}$$

Here, indices at the left of restriction operator $|_{.\rightarrow.}$ symbolize input restriction, whereas indices at the right denote a restriction of output. Thus, restriction $1, 2 \rightarrow 1$ in the above example indicates that the coarse level function $withdraw\_money$ uses inputs 1 and 2 and component 1 of the output tuple of the refining process network $ref(withdraw\_money)$.

In the refining process network, process types $retrieve\_cash$ and $book\_to\_database$ occur. With these, functions $f^{retrieve\_cash}$ and $f^{book\_to\_database}$ are associated, with the following signatures.

$$\textbf{fct}\, f^{retrieve\_cash} : amount \rightarrow (money)$$
$$\textbf{fct}\, f^{book\_to\_database} : amount \times account \rightarrow (db\_log)$$

Function $f^{ref(withdraw\_money)}$ may be expressed by composing its refining functions. The first component of the result tuple of $f^{ref(withdraw\_money)}$ is determined by function $f^{retrieve\_cash}$, the second component by function $f^{book\_to\_database}$ according to

$$f^{ref(withdraw\_money)}(in_1^{wm}, in_2^{wm}) = (f_1^{retrieve\_cash}(in_1^{wm}), f_1^{book\_to\_database}(in_1^{wm}, in_2^{wm}))$$

for input parameters of sort $amount$ assigned to port in_1^{wm} and of sort $account$ assigned to port in_2^{wm}. Here, $f_o^p(in_1, \ldots, in_{i_p})$ denotes the oth component of the resulting output tuple $(o_1, \ldots, o_{o_p})$ of $f^p(in_1, \ldots, in_{i_p})$, where $1 \leq o \leq o_p$ holds.

Analogously to multiple refinement of process types, the composition of functions across different levels of hierarchy may be executed several times.

4 Syntactic enhancements: switches

For modeling purely exemplaric system behavior, decision statements with different possible outcomes within a process network are not necessary, since we model merely that system behavior that was actually executed in a specific exemplaric system run. Possible alternatives of the specific system run which were not actually executed are not modeled. Rather, the different observed system runs are modeled as a set of exemplaric behavior.

Process networks that differ only within a few sections, but which otherwise coincide with respect to structure and content, we refer to as variants. For reducing redundancy within the model of process networks obtained from exemplaric system runs, we carry out some abstraction and comprise the set of variants within a single process network. Depending on the degree of similarity, alternative process networks may either be united to their superset, or combined by introducing decision processes, which we call switches.

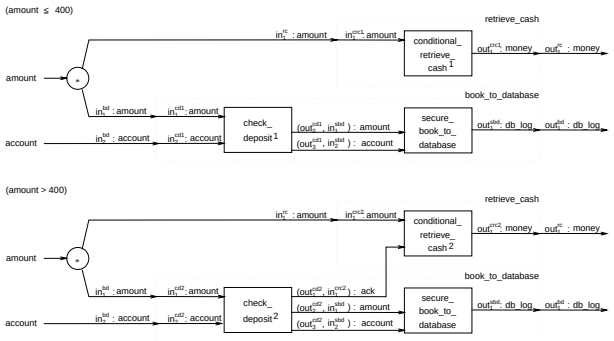

Figure 6. Alternative process networks.

Figure 6 illustrates process networks on the second refinement level of our example process $withdraw\_money$. Depending on the values of the input parameter of sort $amount$, different variants of process type $check\_deposit$ and $conditional\_retrieve\_cash$ are executed, which produce different results or consume different input.

Each variant is a process type. We symbolize the similarity of alternative process types by type names that differ merely in a raised index. The variants of process types $check\_deposit$ and $conditional\_retrieve\_cash$ in Figure 6 correspond to the following signatures.

$$\begin{aligned}
&\textbf{fct } f^{check_deposit^1} : \\
&amount \times account \rightarrow (amount \times account) \\
&\textbf{fct } f^{check_deposit^2} : \\
&amount \times account \rightarrow (ack \times amount \times account)
\end{aligned} \tag{1}$$

$$\begin{aligned}
&\textbf{fct } f^{conditional_retrieve_cash^1} : amount \rightarrow (money) \\
&\textbf{fct } f^{conditional_retrieve_cash^2} : amount \times ack \rightarrow (money)
\end{aligned} \tag{2}$$

4.1. Superset of alternative processes

The alternative process networks of our example differ merely in omitting a single data flow. Otherwise, they are of identical structure and meaning. Alternative process networks which are similar in this sense may be united to a single process network, as illustrated in Figure 7. We achieve this by combining alternative process types to a single new process type which unites the previous alternatives. Using these uniting process types, the uniting process network may be defined.

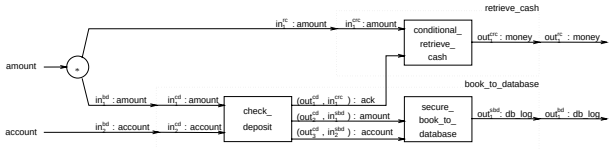

Figure 7. Superset of alternative processes.

Note that uniting process variants into their superset does not add any new syntactic concepts. Thus, we can model this kind of process union without adding additional aspects to our description technique introduced in section 2.

Here, alternative process types are combined to form a single process type, whose input and output is made up of the union of all inputs and outputs of the different alternatives. In this union, those ports of different process types which correspond in their meaning are identified and united to a single port in the new process. Thus, the activity of uniting ports is not carried out merely on the syntactical level. Rather, it requires a systematic analysis of the meaning and usage of the separate ports.

The different alternatives of process execution do not show in the graphical representation of the uniting process in Figure 7. However, in the computation scheme of the glass box view as well as in the associated functions on the level of semantics, these variants are reflected as different cases in decision statements.

In the uniting process network, the different alternatives are encapsulated within the process types $conditional\_retrieve\_cash$ and $check\_deposit$. The functions corresponding to these process types are of the following signatures.

$$\begin{aligned}
&\textbf{fct } f^{check_deposit} : \\
&amount \times account \rightarrow (ack \times amount \times account) \\
&\textbf{fct } f^{conditional_retrieve_cash} : \\
&amount \times ack \rightarrow (money)
\end{aligned}$$

In these functions, the different alternatives are incorporated as decisions. For the uniting process types, the associated function may be expressed with respect to the functions

of finer granularity.

$$f^{conditional\_retrieve\_cash}(in_1^{crc}, in_2^{crc}) =$$

$$= \begin{cases} f^{conditional\_retrieve\_cash^1}(in_1^{crc}) & \text{iff } in_1^{crc} \leq 400 \\ f^{conditional\_retrieve\_cash^2}(in_1^{crc}, in_2^{crc}) & \text{iff } in_1^{crc} > 400 \end{cases}$$

$$f^{check\_deposit}(in_1^{cd}, in_2^{cd}) =$$

$$= \begin{cases} out_2^{cd} = f_1^{check\_deposit^1}(in_1^{cd}, in_2^{cd}) \wedge \\ out_3^{cd} = f_2^{check\_deposit^1}(in_1^{cd}, in_2^{cd}) & \text{iff } in_1^{cd} \leq 400 \\ f^{check\_deposit^2}(in_1^{cd}, in_2^{cd}) & \text{iff } in_1^{cd} > 400 \end{cases}$$

For input parameters of sort $amount$ assigned to port in_1^{wm} and of sort $account$ assigned to port in_2^{wm}, the functions that are associated with the processes in our example are defined as follows.

$$f^{ref(ref(withdraw\_money))^{1,1}}(in_1^{wm}, in_2^{wm}) =$$
$$(f_1^{conditional\_retrieve\_cash^1}(in_1^{wm}),$$
$$f_1^{secure\_book\_to\_database}(f_1^{check\_deposit^1}(in_1^{wm}, in_2^{wm}),$$
$$f_2^{check\_deposit^1}(in_1^{wm}, in_2^{wm})))$$
$$f^{ref(ref(withdraw\_money))^{2,2}}(in_1^{wm}, in_2^{wm}) =$$
$$(f_1^{conditional\_retrieve\_cash^2}(in_1^{wm},$$
$$f_1^{check\_deposit^2}(in_1^{wm}, in_2^{wm})),$$
$$f_1^{secure\_book\_to\_database}(f_2^{check\_deposit^2}(in_1^{wm}, in_2^{wm}),$$
$$f_3^{check\_deposit^2}(in_1^{wm}, in_2^{wm})))$$

For the uniting superset (confer to Figure 7) of the similar process networks, we get the following function.

$$f^{ref(ref(withdraw\_money))}(in_1^{wm}, in_2^{wm}) =$$
$$(f_1^{conditional\_retrieve\_cash}(in_1^{wm},$$
$$f_1^{check\_deposit}(in_1^{wm}, in_2^{wm})),$$
$$f_1^{secure\_book\_to\_database}(f_2^{check\_deposit}(in_1^{wm}, in_2^{wm}),$$
$$f_3^{check\_deposit}(in_1^{wm}, in_2^{wm})))$$

When the decision statements by which the alternative functions are united do not partition the possible combinations of parameter values into disjunct sets, the uniting process type turns to be nondeterministic. In this case, as previously pointed out in section 3.1, we associate a set of functions with the uniting process type. Each of these functions covers all possible combinations of parameter values, where in those cases of more than one possible behavior, each function restricts itself to a single behavior possibility. On the other hand, each of the behavioral possibilities must be covered by at least one of the functions. For each instance of an execution of a nondeterministic process instance, one function of the corresponding set of functions is selected in a nondeterministic way, and then evaluated. Altogether, the set of associated functions models exactly the behavior of the nondeterminstic process type.

4.2. Switches

Different process networks may be congruent in certain subparts, but may differ to a higher extent in other areas. For example, process networks which start identically may continue differently regarding structure and content, in the case that depending on the evaluation of parameter values at a certain point, different possible subsequent process subnetworks may be pursued. In our example in Figure 6, different variants of $check\_deposit$ may be executed, each of which is succeeded by a different process network.

When alternative process networks differ greatly in their input/output functionality in some areas, it is suitable to keep them as process variants rather than uniting them to their superset. These process variants may be encapsulated by input and/or output switches.

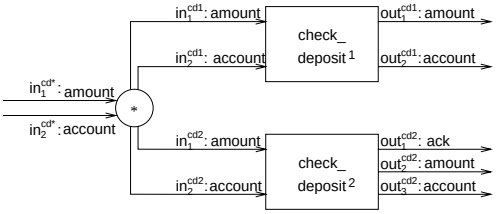

Figure 8. Differing output functionality.

As an example, Figure 8 illustrates process types which coincide in their meaning and their input functionality, but differ in output functionality, as described in equation 1. We unite these alternative process types into an output switch whose black box view is shown in Figure 9. Note that the syntax of the glass box description of switch process types is identical to that of regular process types.

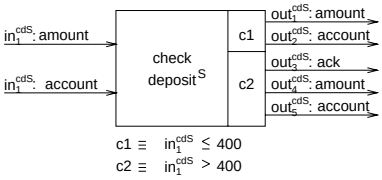

Figure 9. Output switch

When the output switch is integrated within a process network, the process network splits into different process networks succeeding the output switch.

The function associated with an output switch is of the same input functionality as each of the functions of the original alternative process types. However, its output functionality consists of the cartesian product of output functionalities of the orignial functions, which yields for our example

$$\textbf{fct}\ f^{check\_deposit^S} : amount \times account \rightarrow$$
$$\rightarrow (amount \times account \times ack \times amount \times account).$$

Then, function $f^{check\_deposit^S}$ may be expressed using the original alternative functions as follows.

$$f^{check\_deposit^S}(in_1^{cdS}, in_2^{cdS}) =$$
$$= \begin{cases} out_1 = f_1^{check\_deposit^1}(in_1^{cdS}, in_2^{cdS}) \wedge \\ out_2 = f_2^{check\_deposit^1}(in_1^{cdS}, in_2^{cdS}) \text{ iff } in_1^{cdS} \leq 400 \\ out_3 = f_1^{check\_deposit^2}(in_1^{cdS}, in_2^{cdS}) \wedge \\ out_4 = f_2^{check\_deposit^2}(in_1^{cdS}, in_2^{cdS}) \wedge \\ out_5 = f_3^{check\_deposit^2}(in_1^{cdS}, in_2^{cdS}) \text{ iff } in_1^{cdS} > 400 \end{cases}$$

According to this definition, we assign the results of the corresponding subfunction to those output ports that correspond to the fulfilled decision case. Output ports of decision cases that do not evaluate to $true$ have empty output as value, so that subsequent functions will not be triggered for execution. Thus, when processes and functions are linked to form a network, only those branches of the process network are executed which correspond to decision cases that evaluate to $true$.

In our example, the decision statement provides for disjunct cases in evaluation of variable assignments. However, if cases should overlap, the resulting nondeterministic behavior is resolved by splitting it into an equivalent set of functions, as described in section 4.1.

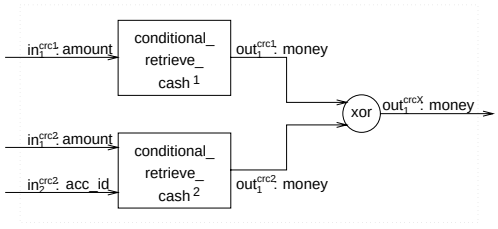

Figure 10. Differing input functionality.

Figure 10 illustrates an example of similar process types which coincide in their output functionality but differ in their input functionality, as described in equation 2. We unite them into an input switch.

We introduce the supplementary function $xor(.,\ldots,.)$ for uniting equally typed channels. If only one of the input channels of xor holds a defined value, this value is output on the outgoing channel. Whenever more than one input channel is assigned with a defined value, xor nondeterministically selects one channel whose value ist output as result.

Function xor can easily be extended to tuples of input channels. Channel tuples with equal type tuples are united to a single output tuple of corresponding tuple type. The functions output consists of the values of the input tuple that is assigned with defined values. If more than one input tuple is assigned with defined values, xor nondeterministically selects one of these channel tupels and outputs the corresponding values.

We unite our alternative process types of Figure 10 by introducing an input switch, as illustrated in Figure 11.

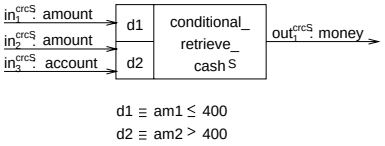

Figure 11. Input switch.

An input switch that is integrated in a process network unites different preceding process networks to a single succeeding process network.

The function that is associated with the input switch is of the same output functionality as each of the functions corresponding to the original process types. However, its input functionality is the cartesian product of input functionalities of the original functions, as described by

$$\textbf{fct } f^{conditional\_retrieve\_cash^S}:$$
$$amount \times amount \times account \rightarrow (money).$$

Function $f^{conditional\_retrieve\_cash^S}$ may be expressed in terms of the original alternative functions as follows.

$$f^{conditional\_retrieve\_cash}(in_1^{crc}, in_2^{crc}, in_3^{crc}) =$$
$$= \begin{cases} f^{conditional\_retrieve\_cash^1}(in_1^{crc}) \\ \quad \text{iff } in_1^{crc} \leq 400 \wedge in_2^{crc} \leq 400 \\ f^{conditional\_retrieve\_cash^2}(in_2^{crc}, in_3^{crc}) \\ \quad \text{iff } in_1^{crc} > 400 \wedge in_2^{crc} > 400 \end{cases}$$

When the different functions do not define disjunctive cases of parameter assignments, we split up the resulting nondeterministic behavior of the input switch into an equivalent set of deterministic functions.

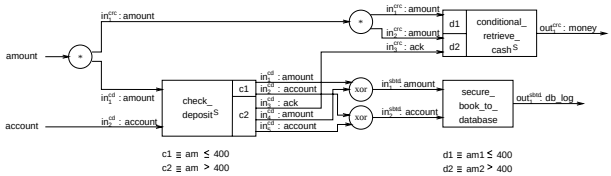

Figure 12. Process network with switches.

Figure 12 shows the second refinement level of our example process $withdraw\_money$ using input and output switches. On the level of semantics, this process network

corresponds to the following function definition.

$$
\begin{aligned}
&f^{ref(ref(withdraw\_money))^S}(in_1^{wm}, in_2^{wm}) = (\\
&f_1^{conditional\_retrieve\_cash^S}(in_1^{wm}, in_1^{wm},\\
&\qquad f_3^{check\_deposit^S}(in_1^{wm}, in_2^{wm})),\\
&f^{secure\_book\_to\_database}(\\
&\qquad xor(f_1^{check\_deposit^S}(in_1^{wm}, in_2^{wm}),\\
&\qquad\quad f_4^{check\_deposit^S}(in_1^{wm}, in_2^{wm})),\\
&\qquad xor(f_2^{check\_deposit^S}(in_1^{wm}, in_2^{wm}),\\
&\qquad\quad f_5^{check\_deposit^S}(in_1^{wm}, in_2^{wm}))))
\end{aligned}
$$

Process types with similar meaning but differing input and output functionality may be united into an IO-switch which combines input and output switch into a single uniting process type.

5. Conclusions and outlook

We presented a semantically well founded description technique for modeling typical system behavior in a way that is independant from organizational or geographical boundaries. Furthermore, we provided a refinement mechanism which supports behavior modeling across different levels of abstraction. Our modeling technique documents causal dependencies among process execution that are due to the communication of messages and events between processes, without introducing any additional artificial sequentialization. Thus we allow for a maximum of parallelism in process execution that conforms with the required causality of communication.

So far, we have provided a formally founded description technique for exemplaric system behavior. In a next step, we will move from a set of single process runs towards processes instances that are executed more than once within a single system run. Thus we need a notion of process state or memory, and consequently adapt our semantics to stream processing functions that work on histories of input and ouput messages (see, for example, [14] and [3]).

Finally, when assigning certain aspects of system behavior to the respective system modules for execution in later stages of the system development process, we leave the cross functional, exemplaric view of business process modeling and turn to modeling the complete behavior of single system components or objects. At this stage, we employ automata or state machines ([10] for modeling component behavior.

The methodic and semantic integration of these approaches is subject of ongoing research.

6. Acknowledgements

I thank Wolfgang Schwerin, Manfred Broy and Bernhard Rumpe for many fruitful discussions.

References

[1] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

[2] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Method Language – Notation Guide*. Rational Software Corporation, Santa Clara, CA., 1.1 c edition, July 1997.

[3] M. Broy. A theory for nondeterminism, parallelism, communication and concurrency. Technical report, Habilitationsschrift, Fakultät für Mathematik und Informatik, Technische Universität München, 1982.

[4] M. Broy. *Informatik – Eine grundlegende Einführung, Teil 1: Problemnahe Programmierung*, volume 1. Springer-Verlag, Berlin, 1992.

[5] M. Broy. (inter-)action refinement: The easy way. In F. Bauer, M. Broy, E. Dijkstra, D. Gries, and C. Hoare, editors, *Program Design Calculi, NATO ASI Series F: Computer and System Sciences, Vol. 118*, pages 121–158. Springer-Verlag, 1993.

[6] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stolen. The requirement and design specification language SPECTRUM – an informal introduction, part ii. Technical Report TUM-I9312, Technische Universität München, Institut für Informatik, München, May 1993.

[7] A. Davis. *Software Requirements – Objects, Functions, and States*. Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1993.

[8] T. DeMarco. *Structured Analysis and System Specification*. Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1979.

[9] O. Ferstl and E. Sinz. Ein vorgehensmodell zur objektmodellierung betrieblicher informationssysteme im semantischen objektmodell (som). In *Bamberger Beiträge zur Wirtschaftsinformatik, Nr. 5*. Universität Bamberg, July 1991.

[10] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State transition diagrams. Technical Report TUM-I9630, Technische Universität München, Institut für Informatik, München, June 1996.

[11] D. Harel. Statecharts – a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[12] C. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall International, Inc., Englewood Cliffs, New Jersey, 1985.

[13] ITU-T. *Z.120 – Message Sequence Chart (MSC)*. ITU-T, Geneva, 1996.

[14] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing, IFIP'74*. North-Holland, 1974.

[15] H. Österle. *Business Engineering – Prozeß- und Systementwicklung*, volume 1. Springer-Verlag, Berlin, 1995.

[16] W. Reisig. *Systementwurf mit Netzen*. Springer-Verlag, Berlin, 1985.

[17] Siemens Nixdorf Informationssysteme AG, München. *GRAPES V3 – Sprachbeschreibung*, Mar. 1995.