A Formally Founded Description Technique for Business Processes*

Veronika Thurner
Department of Computer Science, Technical University of Munich
Arcisstr. 21, 80290 Munich, Germany
email: thurner@informatik.tu-muenchen.de

December 19, 1997

Abstract

As a means of modeling typical system behavior, we derive from data flow nets a description technique for business processes and provide it with a formal semantics based on functions and their composition. Our description formalism features black box and glass box view on system processes, as well as a concept of refinement which supports behavior modeling across several levels of abstraction. Thus we provide a modeling mechanism that is both easy to understand intuitively and formally well founded, and therefore equally adequate for the needs of application domain experts as well as system engineers in requirements engineering.

1 Introduction and motivation

Many approaches to requirements engineering involve a detailed modeling of characteristic system aspects such as structure, data or behavior. These models are a vital means of communication between expert users and system analysts. Also, they are the basis for system design and implementation taking place in later stages of the system development process. Consequently, the quality of requirements specifications is a decisive factor for software quality and correction costs [Dav93].

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 [IT96], interaction diagrams of Booch [Boo94], sequence diagrams of UML [BJR97], or process object schemes [FS91]). Thus, scenarios are often arranged according to structural system aspects, so that the behavior model is always intermingled with, and dominated by, the system architecture. Therefore, additional constraints are added to the behavior model which restrict the order of process execution and consequently the possible amount of parallelism, although they represent constraints that are not due to any causal dependencies

*This work was supported by the Bayerische Forschungsstiftung.

originating from the behavioral model itself. Other techniques, such as activity diagrams in [BJR97] or the process notion of [Kah74], already include aspects of system state in their models. However, although this integrated modeling of several different system aspects at a time might still work with small examples, it quickly turns to be difficult and hard to handle as system complexity increases.

In contrast to this, we apply a task oriented point of view in behavior modeling. Focussing on the major tasks of the system under consideration, 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 begin our behavior modeling by documenting single runs of exemplaric system behavior. As the 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 point on system behavior and the related requirements.

To document our model of typical system behavior, we introduce a description technique that supports behavior modeling in a way that is independant from organizational or geographical boundaries. This modeling technique documents causal dependencies between process and their execution that are due to the exchange of messages and events between processes. However, we do not introduce any additional artificial sequentialization or other constraints on the order of process execution, thus allowing for a maximum of possible parallelism in process execution.

Our modeling technique includes both a black box and a glass box view on business processes. Furthermore, we enhance our description formalism with a refinement mechanism which supports behavior modeling across different levels of abstraction. To reduce redundancy in our process model, we base our process documentation on the definition of process types.

Finally, to allow unique and unambiguous modeling and to precisely relate our description technique to models of other system views, we provide a formal semantics to our description technique, based on functions and their composition. This type of semantics is suitable for supporting our modeling intentions stated above, as it provides a flexible modeling and abstraction mechanism focusing on data dependencies rather than on partially ordered sequences of events that are exchanged between objects.

2 Concrete syntax of a description technique for business processes

We use business processes for modeling system behavior in an exemplaric way, focussing on sequences of 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. A process type defines the interface, internal behavior and refinement structure, which are common to all instances of a specific process type.

Each of these three aspects corresponds to a certain view on a process type. In the black box view, the interface describes the functionality of the process type. The internal behavior, i.e. 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, or, the other way round, the composition of processes types of a finer granularity into a network which realizes a process type of a coarser level of granularity.

Based on the set of defined process types, instances of these types can be composed into process networks which desribe sequences of system behavior in an exemplaric way. The identifiers of process instances are unique throughout the whole model of the system.

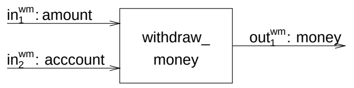
We provide a notation that consists of graphical as well as of textual elements. For the graphical aspects of our notation, we use a derivative of data flow nets which were introduced in [DeM79]. Moreover, we incorporate and enhance notation ideas taken from those parts of the modeling language GRAPES V3 [Sie95] that are relevant for business process modeling. Textual aspects of our notation are provided in extended Backus-Naur form as introduced in [BFG+93]. The non-terminals ⟨process-type⟩, ⟨function⟩ and ⟨predicate-expression⟩ are not specified any further within this work.

2.1 Black Box View

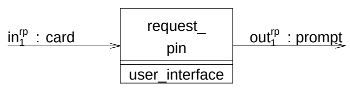
The definition of the black box view specifies the signature of a process type as evident and relevant on the current level of granularity. Here, a process type's name is determined as well as its bundles of typed input and output ports. In the case that a process type is refined into a process network within a subsequent step of development, the definition on the refined level may be supplemented by additional input and output ports. However, these additional ports do not have to be added to the hierarchically higher levels of granularity.

Optionally, a role may be associated with a process type. Roles are auxiliary concepts which link process types to physical actors carrying out instances of these process types. A role can be designed for realization by one or more human beings, a hardware/software system or a combination thereof. Roles group processes according to different, often pragmatic aspects such as qualification, or authorization for usage or decision taking that are necessary for process execution. Another aspect of grouping processes by roles is the encapsulation of data that are to be manipulated by the different processes that are associated with a role. Methodically, roles are usually introduced towards the end of requirements engineering and during design, preparatory to distributing the execution of process instances to the different system components.

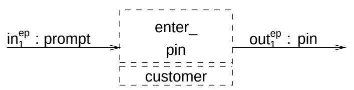
Another optional feature states whether a process is executed within the system or by the system's environment. Respectively, processes are marked as internal or external. Often, this binding is implicitly determined by the role that is associated with the process type. However, for methodical reasons, it is helpful to allow an explicit declaration whether the execution of instances of a certain process type takes place internally or externally to the system under consideration. By default, process types are assumed to be internal.



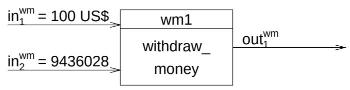
(a) process type *withdraw_money*



(b) process type *request_pin* with associated role *user_interface*



(c) process type *enter_pin* with associated role *customer* and external binding



(d) process instance *wm*1 of process type *withdraw_money*, and assigned input values

Figure 1: Black box definitions of process types and a process instance

Figure 1(a) shows an example of the graphical representation of a process type.

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 name of the role is designated at the lower border of the process type symbol, as shown in Figure 1(b).

External process types are executed outside of the system under consideration. As illustrated in Figure 1(c), we denote them by a dashed circumference of the process type symbol.

In the graphical representation of the black box view of a process instance, the name of the process type is preceded by the identifier of the process instance in a separate section of the process symbol (confer Figure 1(d)).

Whereas the black box view merely specifies the input/output behavior of a process type, the glass box view describes the internal manipulation of data within a process.

2.2 Glass Box View

The glass box view describes the internal manipulation of data during the execution of a process instance. The modeling of nondeterminism is supported. Furthermore, the glass box view documents pre- and postconditions of a process execution.

Thus, 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. Within a process type's computation scheme, input and output data are parameterized by the corresponding port names. If necessary, local variables may be introduced. Depending on the degree of knowledge that is available in the computation method, the scheme may be described informally by structured textual comments, or more formally in mathematical notation.

Moreover, pre- and postconditions of process execution are defined. An instance of a process type is executed only if its precondition is fulfilled, with the precondition being a predicate over the process instance's input parameters. Correspondingly, when the execution of a process instance is completed, the associated postcondition holds. The postcondition is given as a predicate over input and output parameters of the process instance.

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, using the specific values that are assigned to the input ports.

In our notation, we do not introduce any graphical symbols for defining the glass box view on process types, as we do not expect an adequate gain in readability and understandability at this point. Thus, we use a textual notation, where the manipulation of data may be described either mathematically by specifying a function, or as text which may be enhanced by mathematical elements. Pre- and postconditions are specified as predicate expressions.

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

The glass box view of our example process type *withdraw_money* may be given as follows, where we employ a textual representation with some mathematical elements for defining the computation method.

glass box process type *withdraw_money* = **{**
  computes $out_1^{wm} = f_1^{wm}(in_1^{wm}, in_2^{wm})$, with

$$f_1^{wm}(in_1^{wm}, in_2^{wm}) = \begin{cases} \text{requested money} & \text{if requested amount smaller than 400} \\ \text{requested money} & \text{if requested amount greater than 400} \\ & \quad \text{and account deposit greater than} \\ & \quad \text{or equal to requested amount} \\ \text{no money} & \text{if requested amount greater than 400} \\ & \quad \text{and account deposit smaller than} \\ & \quad \text{requested amount} \end{cases}$$

  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 [Bro93]. Such a process network is constructed from process types of finer granularity. They are connected via interfaces which were defined in the black box view, by connecting an output port of one process to an input port of another process, thus building an internal channel. A channel is denoted by the pair of its ports according to $(outport, inport)$. 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 2 illustrates the refinement of process type *withdraw_money* from our example in Figure 1(a).

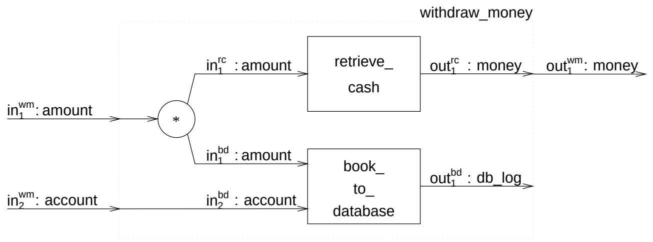


Figure 2: Refinement of process type *withdraw_money*

Operator $*$ symbolizes the duplication of the message assigned to a port, and the redirection of the copies to several subports on the refinement level.

Possibly, within a refining process network, a single process type may occur multiply. However, in our graphical representation these different occurances may easily be distinguished by their geometrical position within the diagram. Thus as well, the structure of connecting channels may be defined without ambiguities.

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 as well.

3 Semantics

The semantics of our description technique for business processes is based on functions and their composition. Compositionality is necessary for formalizing refinement, or, if seen from another angle, the composition of single processes to a process network. This usage of function composition is related to computation forms which are discussed e.g. in [Bro92].

In the definition of semantics, we assign a function with adequate input/output signature to each process type. This function formalizes the computation scheme associated with the process type.

Some existing approaches to process modeling define a semantics based on event traces (for example [Hoa85]). 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.

In the following, let

- PT denote a set of identifiers of process types,
- PI denote a set of identifiers of process instances,
- P denote a set of identifiers of ports,
- F denote 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 the typed input/output functionality of a process type. On the level of semantics, this aspect corresponds to the definition of 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

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

where, respectively, $s_{in_1^p}, \ldots, s_{in_{i_p}^p} \in S$ 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 computation scheme that corresponds to process type p is specified by the body of function f^p. The explicit documentation of the function body 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.

With our example process type *withdraw_money* from Figure 1(a), we associate a function $f^{withdraw\_money}$ whose functionality

$$\textbf{fct } f^{withdraw\_money} : \quad amount \times account \longrightarrow (money)$$

mirrors exactly the input/output situation of the corresponding process.

On the level of semantics, process execution is equivalent to the evaluation of the associated function on specific input values.

So far, we assumed our processes to be deterministic. However, the semantics can easily be generalized to cover nondeterministic processes as well. We achieve this 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 process network. Within this process network, process types of finer granularity are linked by connecting some of their input and output ports.

On the level of semantics, refinement of a process type to a process network corresponds to representing a function by the composition of other functions. When the refinement level contains supplementary input and output ports that were not relevant or not yet known on the coarser levels of refinement, a restriction of the input/output functionality of the composition of refining functions is necessary as well.

In Figure 2, our example process *withdraw_money* from Figure 1(a) 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

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

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

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

Here, indices at the left of resctrction operator $|_{.\to .}$ symbolize input restriction, whereas indices at the right denote a restriction of output.

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.

$$\begin{array}{llrcl} \textbf{fct} & f^{retrieve\_cash}: & amount & \longrightarrow & (money) \\ \textbf{fct} & f^{book\_to\_database}: & amount \times account & \longrightarrow & (db\_log) \end{array}$$

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 by using business processes, 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 3 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, respectively, 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 type *check_deposit* in Figure 3

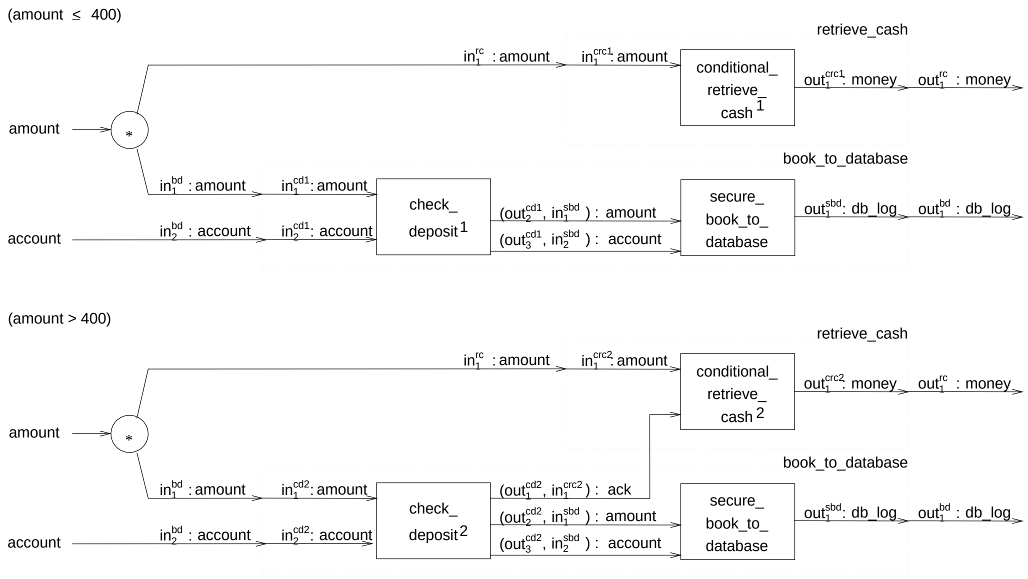


Figure 3: Alternative process networks

correspond to the following signatures.

$$
\begin{array}{llcl}
\textbf{fct } f^{check\_deposit^1}: & amount \times account & \longrightarrow & (amount \times account) \\
\textbf{fct } f^{check\_deposit^2}: & amount \times account & \longrightarrow & (ack \times amount \times account)
\end{array}
\tag{1}
$$

Process type *conditional_retrieve_cash* occurs in two variants with the following functionalities.

$$
\begin{array}{lrcl}
\textbf{fct } f^{conditional\_retrieve\_cash^1}: & amount & \longrightarrow & (money) \\
\textbf{fct } f^{conditional\_retrieve\_cash^2}: & amount \times ack & \longrightarrow & (money)
\end{array}
\tag{2}
$$

4.1 Uniting alternative process networks to their superset

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 4. 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.

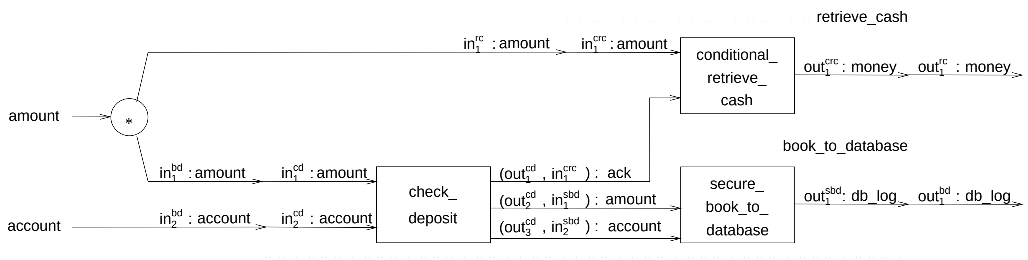


Figure 4: Uniting alternative process networks to their superset

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 4. 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.

$$\begin{array}{llrcl} \textbf{fct } f^{check\_deposit} & : & amount \times account & \longrightarrow & (ack \times amount \times account) \\ \textbf{fct } f^{conditional\_retrieve\_cash} & : & amount \times ack & \longrightarrow & (money) \end{array}$$

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.

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

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

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.

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

For the uniting superset (confer to Figure 4) of the similar process networks, we get the following function.

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

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 function models exactly the behavior of the nondeterminstic process type.

This decomposition of nondeterministic behavior into a set of functions is illustrated by the following example.

Let f_v, f_v^A and f_v^B be functions over a set v of typed variables. Furthermore, let $\mathcal{B}$ be the set of possible value combinations over this set of variables v. In addition, let $A \subseteq \mathcal{B}$ and $B \subseteq \mathcal{B}$ be subsets of the set of possible value combinations. Also, $A \cup B = \mathcal{B}$ and $AB := A \cap B \neq \emptyset$ holds. Finally, let $\beta(v) \in \mathcal{B}$ be one specific combination of values assigned to the set of variables v.

The process behavior is modeled by f_v^A if $\beta(v) \in A$ holds, and by f_v^B if $\beta(v) \in B$ holds.

As assumption $A \cap B \neq \emptyset$ holds, this behavior is nondeterministic. For resolving this nondeterminism, we describe this behavior in terms of a set of functions f_v as follows.

$$f_v \equiv \{f_v^{A'}, f_v^{B'}\}, \quad \text{where} \quad \begin{array}{l} f_v^{A'} = \begin{cases} f_v^A & \text{iff} \quad \beta(v) \in A \\ f_v^B & \text{iff} \quad \beta(v) \in B \backslash AB \end{cases} \\[2ex] f_v^{B'} = \begin{cases} f_v^A & \text{iff} \quad \beta(v) \in A \backslash AB \\ f_v^B & \text{iff} \quad \beta(v) \in B \end{cases} \end{array}$$

4.2 Encapsulating alternative processes by 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 3, depending on the variable assignments, different variants of *check_deposit* are 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.

4.2.1 Output Switch

Process types which coincide in their meaning and their input functionality, but which differ in their output functionality, may be united into an output switch.

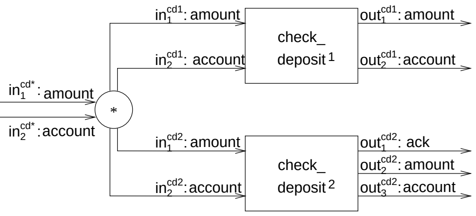


Figure 5: Similar process types with differing output functionality

As an example, Figure 5 illustrates similar process types with identical input functionality but differing output functionality, as described in equation 1. We unite these alternative process types into an output switch which is shown in Figure 6.

Note that the syntax of the glass box description of switch process types is identical to that of regular process types.

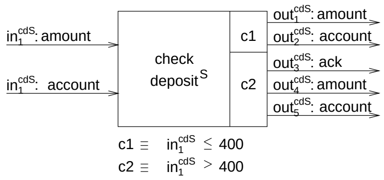


Figure 6: Uniting alternative process types to an output switch

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. Thus

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

holds for our example.

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

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

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.

In the following section, we introduce input switches as an analogon to the output switches we just presented.

4.2.2 Input Switch

Process types that correspond in their meaning and in their output functionality, but which differ in their input functionality may be united to form an input switch.

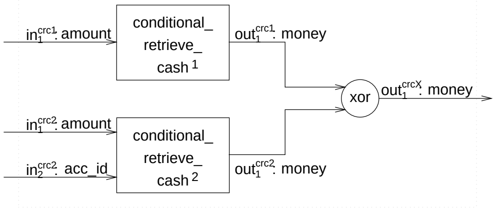


Figure 7: Similar process types with differing input functionality

Figure 7 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 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 7 by introducing an input switch, as illustrated in Figure 8.

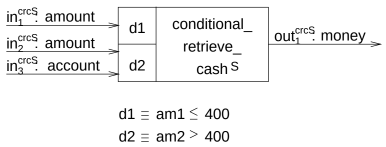


Figure 8: Uniting alternative process types by an input switch

Note that again, the syntax of the glass box description of switch process types is identical to that of regular process types.

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. Thus

$$\mathbf{fct}\ f^{conditional\_retrieve\_cash^S} : \quad amount \times amount \times account \longrightarrow (money)$$

holds.

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

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

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.

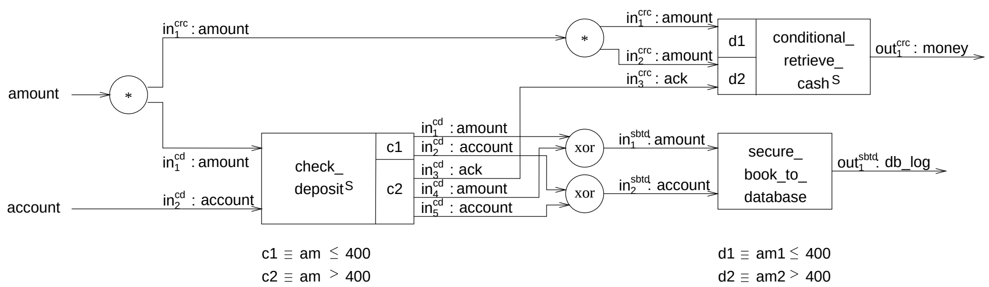


Figure 9: Process network with input and output switch

Figure 9 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.

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

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, [Kah74] and [Bro82]).

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 ([GKRB96] for modeling component behavior.

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

Acknowledgements

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

References

[BFG+93] 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.

[BJR97] G. Booch, I. Jacobson and J. Rumbaugh. *Unified Method Language – Notation Guide.* Rational Software Corporation, Santa Clara, CA., 1.1 c edition, July 1997.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 1994.

[Bro82] M. Broy. A Theory for Nondeterminism, Parallelism, Communication and Concurrency. Technical report, Habilitationsschrift, Fakultät für Mathematik und Informatik, Technische Universitä München, 1982.

[Bro92] M. Broy. *Informatik – Eine grundlegende Einführung, Teil 1: Problemnahe Programmierung*, volume 1. Springer-Verlag, Berlin, 1992.

[Bro93] M. Broy. (Inter-)Action Refinement: The Easy Way. In F.L. Bauer, M. Broy, E.W. Dijkstra, D. Gries and C.A.R Hoare, editors, *Program Design Calculi, NATO ASI Series F: Computer and System Sciences, Vol. 118*, pages 121–158. Springer-Verlag, 1993.

[Dav93] A.M. Davis. *Software Requirements – Objects, Functions, and States.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1993.

[DeM79] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1979.

[FS91] O.K. Ferstl and E.J. Sinz. Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM). In *Bamberger Beiträge zur Wirtschaftsinformatik, Nr. 5.* Universität Bamberg, July 1991.

[GKRB96] 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.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice Hall International, Inc., Englewood Cliffs, New Jersey, 1985.

[IT96] ITU-T. *Z.120 – Message Sequence Chart (MSC).* ITU-T, Geneva, 1996.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing, IFIP'74.* North-Holland, 1974.

[Sie95] Siemens Nixdorf Informationssysteme AG, München. *GRAPES V3 – Sprachbeschreibung*, March 1995.