Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

else $\text{TRY}_{Async(Send)}(subj, state, m)$
if $ComAct = Receive$ **then**
if $Async(Receive)(m)$ **then** $\text{TRY}_{Async(Receive)}(subj, state, m)$
if $Sync(Receive)(m)$ **then** $\text{TRY}_{Sync(Receive)}(subj, state, m)$
where
$\text{MARKCHOICE}(m, subj, state) =$
$\text{DELETE}(m, MsgToBeHandled(subj, state))$
$currMsgKind(subj, state) := m$

The components $\text{TRY}_{Async(ComAct)}$ and $\text{TRY}_{Sync(ComAct)}$ check whether the *ComAct*ion can be done asynchronoysly resp. synchronously and in case of failure $\text{CONTINUEMULTIROUND}_{Fail}$. If a communication turns out to be possible they use components[16] ASYNCH(*ComAct*) and SYNC(*ComAct*) which carry out the actual *ComAct*ion and $\text{CONTINUEMULTIROUND}_{Success}$. They are defined below together with $PossibleAsync_{ComAct}(subj, m)$ and $PossibleSync_{ComAct}(subj, m)$ by which they are guarded.

$\text{TRY}_{Async(ComAct)}(subj, state, m) =$
if $PossibleAsync_{ComAct}(subj, m)$ // async communication possible
then $\text{ASYNC}(ComAct)(subj, state, m)$
else
if $ComAct = Receive$ **then**
$\text{CONTINUEMULTIROUND}_{Fail}(subj, state, m)$
if $ComAct = Send$ **then** $\text{TRY}_{Sync(ComAct)}(subj, state, m)$
$\text{TRY}_{Sync(ComAct)}(subj, state, m) =$
if $PossibleSync_{ComAct}(subj, m)$ // sync communication possible
then $\text{SYNC}(ComAct)(subj, state, m)$
else $\text{CONTINUEMULTIROUND}_{Fail}(subj, state, m)$

3.3 How to Actually Send a Message

In this section we define the ASYNCH(*Send*) and SYNC(*Send*) components which if the condition $PossibleAsync_{Send}$ resp. $PossibleSync_{Send}$ is true asynchronously or synchronously carry out the actual *Send* and $\text{CONTINUEMULTIROUND}_{Success}$.

$PossibleAsync_{Send}(subj, m)$ means that m is not *Blocked* by the receiver's input pool so that in ASYNCH(*Send*) *subj*ect can send m asynchronously:[17] PASSMSG to the input pool and $\text{CONTINUEMULTIROUND}_{Success}$.[18]

$PossibleSync_{Send}(subj, m)$ means that a *RendezvousWithReceiver* is possible for the *subj*ect whereby it can definitely send m synchronously via $Sync_{Send}$. For the sender *subj*ect this comes up to simply $\text{CONTINUEMULTIROUND}_{Success}$.

[16] The parameter *ComAct* plays here the role of an index.
[17] The reader will notice that for *Send* actions the *PossibleAsync* predicate depends only on messages. We have included the *subj*ect parameter for reasons of uniformity, since it is needed for $PossibleAsync_{Receive}$.
[18] In case of a single send action the subject will directly $\text{COMPLETENORMALLY}_{Send}$.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

The prepared message becomes available through the *RendezvousWithReceiver* so that the receiver can RECORDLOCALLY it (see the definitions in Sect. 3.4).

In ASYNC(*Send*) the component PASSMSG(*msg*) is called[19] if the *msg* is not *Blocked*. Therefore *msg* insertion must take place in two cases: either *msg* violates no constraint row or it violates one and the action of the first row it violates is not DropIncoming; in the second case also a DROP action has to be done to create in the input pool a place for the incoming *msg*.

ASYNC(*Send*)(*subj*, *state*, *msg*) =
  PASSMSG(*msg*)
  CONTINUEMULTIROUND$_{Success}$(*subj*, *state*, *msg*)
where
  PASSMSG(*msg*) =
    let *pool* = *inputPool*(*receiver*(*msg*))
      row = *first*({*r* ∈ *constraintTable*(*pool*) |
        ConstraintViolation(*msg*, *r*)})
          if *row* ≠ **undef and** *action*(*row*) ≠ *DropIncoming*
            then DROP(*action*)
          if *row* = **undef or** *action*(*row*) ≠ *DropIncoming* **then**
            INSERT(*msg*, *pool*)
            insertionTime(*msg*, *pool*) := *now*
  DROP(*action*) =
    if *action* = *DropYoungest* **then** DELETE(*youngestMsg*(*pool*), *pool*)
    if *action* = *DropOldest* **then** DELETE(*oldestMsg*(*pool*), *pool*)
  PossibleAsync$_{Send}$(*subj*, *msg*) iff **not** *Blocked*(*msg*)
  Blocked(*msg*) iff
    let *row* = *first*({*r* ∈ *constraintTable*(*inputPool*(*receiver*(*msg*))) |
      ConstraintViolation(*msg*, *r*)})
        row ≠ **undef and** *action*(*row*) = *Blocking*

In SYNC(*Send*)(*subj*, *state*, *msg*) the *subj*ect has nothing else to do than to CONTINUEMULTIROUND$_{Success}$ because through the *RendezvousWithReceiver* the elaborated *msg* becomes available to the receiver which will RECORDLOCALLY it during its *RendezvousWithSender* (see Sect. 3.4).

SYNC(*Send*)(*subj*, *state*, *msg*) =
  CONTINUEMULTIROUND$_{Success}$(*subj*, *state*, *msg*)
PossibleSync$_{Send}$(*subj*, *msg*) iff *RendezvousWithReceiver*(*subj*, *msg*)

Necessarily the following description of *RendezvousWithReceiver* refers to some details of the definitions for receive actions described in Sect. 3.4. Upon the first reading this definition may be skipped to come back to it after having read Sect. 3.4.

Copyright 2011 Carl Hanser Verlag München

[19] Typically an implementation will charge the input pool manager to execute PASSMSG, even if here the machine appears as component of a *subj*-rule.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

For a $RendezvousWithReceiver(subj, msg)$ the receiver has to $tryToReceive$ (see Fig. 4) synchronously (i.e. the receiver has chosen a $currMsgKind$[20] which requests a synchronous message transfer, described in $Sync(Receive)$ (see Sect. 3.4) as $commMode(currMsgKind) = sync$ and *subj*ect itself has to try a synchronous message transfer, i.e. the msg it wants to send has to be *Blocked* by the first synchronization requiring row which concerns msg (i.e. where $Match(msg, row)$ holds) in the $constraintTable$ of the receiver's input pool. Furthermore the msg the sender offers to send must *Match* the $currMsgKind$ the receiver has currently chosen in its current SID_state.

$RendezvousWithReceiver(subj, msg)$ iff
  $tryMode(rec) = tryToReceive$ **and** $Sync(Receive)(currMsgKind)$
    and $SyncSend(msg)$ **and** $Match(msg, currMsgKind)$
where
  $rec = receiver(msg),\ recstate = SID_state(rec)$
  $currMsgKind = currMsgKind(rec, recstate)$
  $blockingRow =$
    $first(\{r \in constraintTable(rec) \mid ConstraintViolation(msg, r)\})$
  $SyncSend(msg)$ iff $size(blockingRow) = 0$

Remark. The definition of *RendezvousWithReceiver* makes crucial use of the fact that for each subject its SID_state is uniquely determined so that for a subject in *tryMode tryToReceive* the selected receive alternative can be determined.

3.4 How to Actually Receive a Message

In this section we define the two ASYNCH(*Receive*) and SYNC(*Receive*) components which asynchronously or synchronously carry out the actual *Receive* action and $\text{CONTINUEMULTIROUND}_{Success}$ if the conditions $PossibleAsync_{Receive}$ resp. $PossibleSync_{Receive}$ is satisfied.

There are four kinds of basic receive action, depending on whether the receiver for the currently chosen kind of expected messages in its current *alt*ernative is ready to receive ('expects') *any* message or a message from a particular *sender* or a message of a particular *type* or a message of a particular type from a particular sender. We describe such receive conditions by the set *ExpectedMsgKind* of triples describing the combinations of sender and message type from which the receiver may choose $mult(alt)$ many for messages it will accept (see the definition of $\text{PREPAREMSG}_{Receive}$ in Sect. 3.1).

$ExpectedMsgKind(subj, state, alt)$ yields a set of 3-tuples of form:
  $s\ t\ commMode$
where
  $s \in Sender \cup \{any\}$ **and** $t \in MsgType \cup \{any\}$
  $commMode \in \{async, sync\}$ // accepted communication mode

[20] This MultiRound location is updated in MARKCHOICE.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

The communication mode decides upon whether the receiver will try to ASYNC(*Receive*) or to SYNC(*Receive*) a message of a chosen expected message kind.

$Async(Receive)(m)$ holds if $commMode(m) = async$. If a *subj*ect is called to ASYNC(*Receive*)$(subj, state, m)$ it knows that a message satisfying the asynchronous receive condition $PossibleAsync_{Receive}(subj, m)$ is *Present* in its input pool. It can then $\text{CONTINUEMULTIROUND}_{Success}$ and ACCEPT a message matching m. Since the input pool may contain at a given moment more than one message which matches m, to ACCEPT a message one needs another selection function $select_{ReceiveOfKind(m)}$ to determine the one message which will be received.

$\text{ASYNC}(Receive)(subj, state, msg) =$
  $\text{ACCEPT}(subj, msg)$
  $\text{CONTINUEMULTIROUND}_{Success}(subj, state, msg)$
where
  $\text{ACCEPT}(subj, m) =$
    let $receivedMsg =$
      $select_{ReceiveOfKind(m)}(\{msg \in inputPool(subj) \mid Match(msg, m)\})$
        $\text{RECORDLOCALLY}(subj, receivedMsg)$
        $\text{DELETE}(receivedMsg, inputPool(subj))$
  $Async(Receive)(m)$ iff $commMode(m) = async$
  $PossibleAsync_{Receive}(subj, m)$ iff $Present(m, inputPool(subj))$
  $Present(m, pool)$ iff **forsome** $msg \in pool$ $Match(msg, m)$

Copyright: 2011 Carl Hanser Verlag München

When SYNC(*Receive*)$(subj, state)$ is called, the receiver knows that there is a *sender* for a *RendezvousWithSender* (a subject which right now via a TRY_{Send} action tries to and *CanAccess* the receiver's input pool with a matching message, see Sect. 3.3) to receive its *msgToBeSent*. The synchronization then succeeds: *subj*ect can RECORDLOCALLY the *msgToBeSent*, bypassing the input pool,[21] and $\text{CONTINUEMULTIROUND}_{Success}(subj, state, currMsgKind(subj, state))$.

$\text{SYNC}(Receive)(subj, state, msgKind) =$
  let $P = inputPool(subj),\ sender = \iota s(CanAccess(s, P))$
    $\text{RECORDLOCALLY}(subj, msgToBeSent(sender, SID_state(sender))$
    $\text{CONTINUEMULTIROUND}_{Success}(subj, state, msgKind)$
$Sync(Receive)(msgKind)$ iff $commMode(msgKind) = sync$
$PossibleSync_{Receive}(subj, msgKind)$ iff
  $RendezvousWithSender(subj, msgKind)$

[21] The input pool is bypassed only concerning the act of passing the message from sender to receiver during the rendezvous. It is addressed however to determine the synchronization partner as the unique subject which in the given state can communicate with the receiver (whether synchronously or asynchronously), as mentioned in the footnote to the definition of TRY_{Send} in Sect. 3.3.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

$RendezvousWithSender(subj, msgKind)$ iff
  $Sync(Receive)(msgKind)$ **and**
    let $sender = \iota s(CanAccess(s, inputPool(subj))$
    let $msgToBeSent = msgToBeSent(sender, SID_state(sender))$
      $tryMode(sender) = tryToSend$ **and** $SyncSend(msgToBeSent)$
        and $Match(msgToBeSent, msgKind)$

Remark. The definition of *RendezvousWithSender* makes crucial use of the fact that for each subject its *SID_state* is uniquely determined and therefore for a subject in *tryMode tryToSend* also the *msgToBeSent*. Thus through the rendezvous this message becomes available to the receiver to RecordLocally it.

The subcomponent structure of Behavior(*subj*, *state* for *state*s whose associated *service* is a *ComAct* (Send or Receive) is illustrated in Fig. 5.

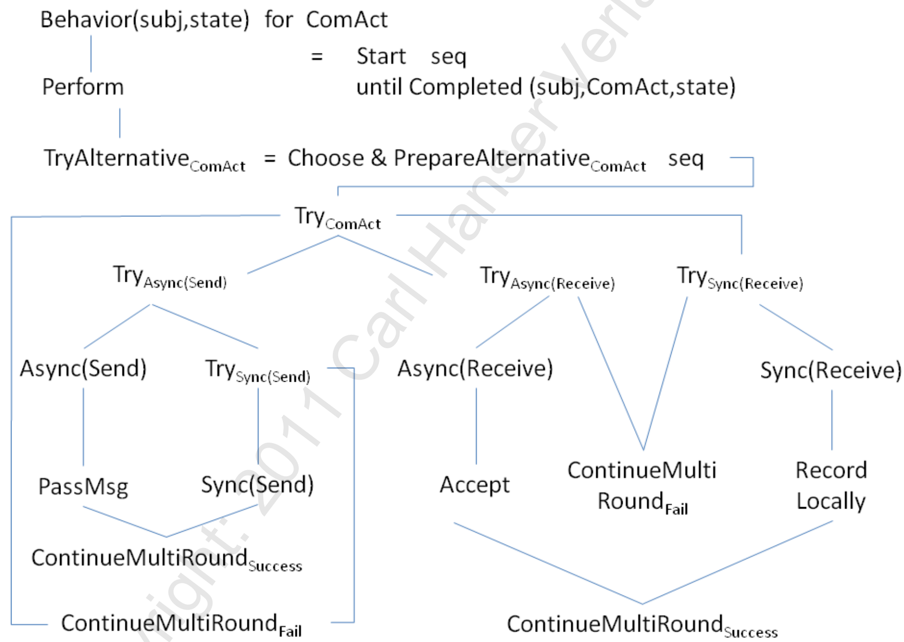


Fig. 5. Subcomponent Structure for Communication Behavior

3.5 Internal Functions

A detailed internal Behavior of a *subject* in a *state* with internal function A can be defined in terms of the submachines Start and Perform together with the

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

completion predicate *Completed* for the parameters $(subj, A, state)$ in the same manner as has been done for communication actions in Sect. 3.3-3.4—but only once it is known how to start, to perform and to complete A. For example, for Java coded functions A Start$(subj, A, state)$ could mean to call the (multi-threaded) Java interpreter *execJavaThread* defined in terms of ASMs in [4, p.101], Perform$(subj, A, state)$ means to execute it step by step and the completion predicate coincides with the termination condition of *execJavaThread*. A still more detailed description, one step closer to executed code, can be obtained by a refinement which replaces the computation of *execJavaThread* for A by a (in [4, Ch.14] proven to be equivalent) computation of the Java Virtual Machine model (called $diligentVM_D$ in [4, p.303]) on $compile(A)$.

For internal *state*s with uninterpreted internal functions A the two submachines of Behavior$(state)$ and the completion predicate remain abstract and the semantics of the SBD where they occur derives from the semantics of ASMs [2] for which the only requirement is that in an ASM state every function is interpreted even if the specification does not define the interpretation. The only requirement is that Performing an internal action is guarded by an interrupt mechanism. This comes up to further specify the SID-transition scheme for internal actions by detailing its **else**-clause as follows:

if $Timeout(subj, state, timeout(state))$ **then**
  $\text{Interrupt}_{service(state)}(subj, state)$
elseif $UserAbruption(subj, state)$
  then $\text{Abrupt}_{service(state)}(subj, state)$
  else Perform$(subj, state)$

Remark. An internal function is not permitted to represent a nested subject behavior diagram so that the SID-level normalized behavior view, the one defined by the subject behavior diagrams of a process (see Sect. 5.2), is clearly separated from the local subject behavior view for the execution of a single internal function by a subject. At present the tool permits as internal functions only self-services, no delegated service.

Copyright: 2011 Carl Hanser Verlag München

4 A Structured Behavioral Concept: Alternative Actions

Additional structural constructs can be introduced building upon the definitions for the core constructs of subject behavior diagrams: internal function, send and receive. The goal is to permit compact structured representations of processes which make use of common reuse, abstraction and modularization techniques. Such constructs can be defined by further refinements of the ASMs defined in Sect. 3 to accurately capture the semantics of the core SBD-constituents. The refined machines represent each a conservative (i.e. purely incremental) extension of the previous machines in the sense that on the core actions the two machines have the same behavior, whereas the refined version can also interprete additional constructs.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

In this section we deal with a structural extension concerning the general behavior of subjects, namely alternative actions. In Sect. 5 extensions concerning the communication constructs will be explained.

The concept of alternative actions allows the designer to express the order independence of certain actions of a subject. This abstraction from the sequential execution order for specific segments in a subject behavior diagram run is realized by introducing so-called *alternative action* (also called alternative path) states, a structured version of SID-states which is added to communication and internal action states.

At an alternative action *state* the computation of a subject splits into finitely many interleaved subcomputations of that subject, each following a (so-called alternative) subject behavior diagram $altBehDgm(state, i)$ of that subject ($1 \leq i \leq m$ for some natural number m determined by the *state*). For this reason such SID-*state*s are also called *altSplit* states.

$$AltBehDgm(altSplit) = \{altBehDgm(altSplit, i) \mid 1 \leq i \leq m\}$$

Stated more precisely, to PERFORM ALTACTION—the *service* associated to an alternative action *state*—means to perform for some subset of these alternative SBDs the behavior of each subdiagram in this set, executed step by step in an arbitrarily interleaved manner.[22] Some of these subdiagram computations may be declared to be compulsory with respect to their being started respectively terminated before the ALTACTION can be *Completed*.

To guarantee for computations of alternative action states a conceptually clear termination criterion in the presence of compulsory and optional interleaved subcomputations each altSplit *state* comes in pair with a unique *alternative action join* state $altJoin(state)$. The split and join states are decorated for each subdiagram D in $AltBehDgm(state)$ with an $entryBox(D)$ and an $exitBox(D)$ where in the pictorial representation (see Fig. 6) an x is put to denote the compulsory nature of entering resp. exiting the D-subcomputation via its unique $altEntry(D)$ resp. $altExit(D)$ state linked to the corresponding box. Declaring $altEntry(D)$ and/or $altExit(D)$ as *Compulsory* expresses the following constraint on the run associated to the ALTACTION split state:

- A compulsory $altEntry(D)$ state must be entered during the run so that the D-subcomputation must have been started before the run can be *Completed*. It is required that every alternative action split state has at least one subdiagram with compulsory *altEntry* state.
- A compulsory $altExit(D)$ state must be reached in the run, for the run to be *Completed*, if during the run a D-subcomputation has been entered at $altEntry(D)$ (whether the $altEntry(D)$ state is compulsory or not). It is required that every alternative action join state has at least one subdiagram with compulsory *altExit* state.[23]

[22] It is natural to apply the interleaving policy to alternative steps of one subject. The model needs no interleaving assumption on steps of different subjects.

[23] This condition implies that if an alternative action node is entered where no subdiagram with compulsory *altExit* has a compulsory *altEntry*, the subcomputation

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

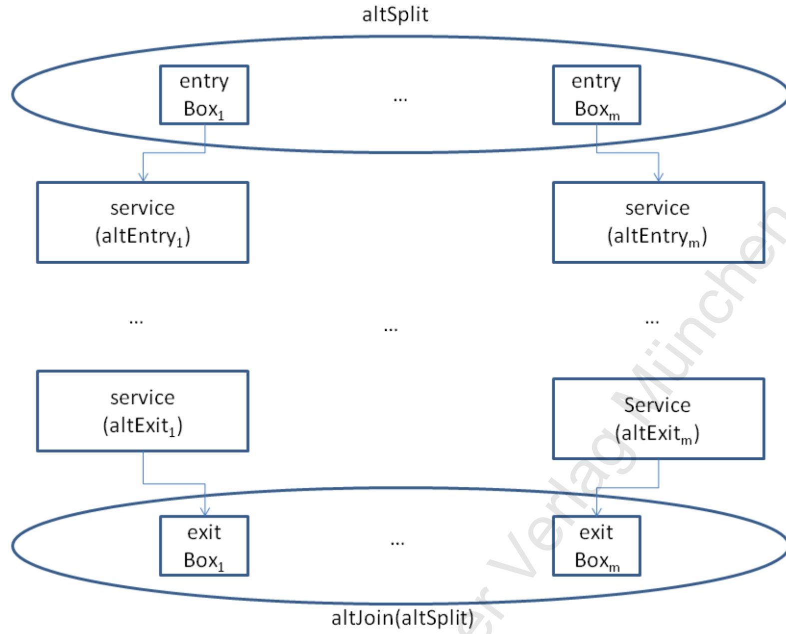


Fig. 6. Structure of Alternative Action Nodes

Copyright 2011 Carl Hanser Verlag München

When PROCEED takes the edge which leads out of $altExit(D)$ to its successor state $exitBox(D)$ (see Fig. 6), the computation of the service associated to $altExit(D)$ and therefore the entire D-subcomputation is completed. This does not mean yet that the entire computation of the ALTACTION state is *Completed*: $exitBox(D)$ is a wait state to wait for all other to-be-exited subcomputations of the ALTACTION state to be completed too. Formally the *service* ALTACTIONWAIT associated to a wait state is empty and there is no isolated exit from a wait state (read: no wait action is ever *Completed* in isolation) but only a common EXITALTACTION from all relevant wait states once ALTACTION is *Completed* (see below). This is formalized by the following definition.

START($subj$, ALTACTIONWAIT, $exitBox$) =
  INITIALIZECOMPLETIONPREDICATE$_{AltActionWait}$($subj$, $exitBox$)
PERFORM($subj$, ALTACTIONWAIT, $exitBox$) = **skip**

It is then stipulated that an ALTACTION—read: the run STARTed when entering an alternative action SID-state—is *Completed* if and only if for each subdiagram D with compulsory $altExit(D)$ state the subject during the run has

of this alternative action is immediately *Completed*. Therefore it seems reasonable to require for alternative action nodes to have at least one subdiagram where both states *altEntry* and *altExit* are compulsory.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

reached the $exitBox(D)$ state—by construction of the diagram this can happen only through the $altExit(D)$ after having *Completed* the *service* associated to this state and therefore the entire D-subcomputation—if in the run a subdiagram computation has been started at all at $altEntry(D)$ of D

Therefore from the SID-level point-of-view the BEHAVIOR($subj, node$) for an alternative action *node* is defined exactly as for standard nodes (with or without multiple (condition) exits); what is specific is the definition of STARTing and PERFORMing the steps of (read: the run defined by) an ALTACTION and the definition of when it is *Completed*. In other words we treat ALTACTION as the *service* associated to an alternative action state.

For the formal definition of what it means to START and to PERFORM the ALTACTION associated to an *altSplit* state the fact is used that SID-states of a subject are (implicitly) parameterized by the diagram in which the states occur. As a result one can keep track of whether the subject is active in a subcomputation of one of the alternative subject behavior diagrams in $AltBehDgm(altSplit)$ by checking whether the $SID_state(subj, D)$ has been entered by the subject (formally: whether it is defined) for any of these subdiagrams D. Therefore START($subj$, ALTACTION, $altSplit$) sets $SID_state(subj, D)$ to $altEntry(D)$ for each subdiagram D whose $altEntry(D)$ state is *Compulsory* and guarantees that the associated $service(altEntry(D))$ is STARTed. For the other subdiagrams $SID_state(subj, D)$ is initialized to **undef**.[24]

START($subj$, ALTACTION, $altSplit$) =
  forall $D \in AltBehDgm(altSplit)$
    if $Compulsory(altEntry(D))$ **then**
      $SID_state(subj, D) := altEntry(D)$
      START($subj, service(altEntry(D)), altEntry(D)$)
    else $SID_state(subj, D)$:=**undef**

Copyright: 2011 Carl Hanser Verlag München

As a consequence the computation of *subj*ect in a subdiagram D becomes active by defining the $SID_state(subj, D)$ so that the formal definition of the completion condition for alternative actions nodes described above reads as follows:[25]

Completed($subj$, ALTACTION, $altSplit$) iff
  forall $D \in AltBehDgm(altSplit)$
    if $Compulsory(altExit(D))$ **and** $Active(subj, D)$
      then $SID_state(subj, D) = exitBox(D)$
  where
    $Active(subj, D)$ iff $SID_state(subj, D) \neq$ **undef**

[24] This definition of START implies that $entryBox(D)$ is only a placeholder for the *Compulsory* attribute of D, whereas $exitBox(D)$ is treated as a diagram state for ALTACTIONWAITing that the entire ALTACTION action is *Completed*.

[25] The completion predicate for alternative action nodes is a derived predicate, in contrast to its controlled nature for communication actions.

Thus from the *altSplit* state the *subj*ect reaches its unique SID-successor state *altJoin*(*altSplit*),[26] where *subj*ect performs as ExitAltAction action (with empyt Start) to reset $SID_state(subj, D)$ for each alternative diagram $D \in AltBehDgm(altSplit)$ and to SetCompletionPredicate$_{ExitAltAction}$, so that *subj*ect in the next step from here will Proceed to a successor SID-state of the *altJoin*(*altSplit*) state.

Start(*subj*, ExitAltAction, *altJoin*(*altSplit*)) = **skip**
Perform(*subj*, ExitAltAction, *altJoin*) =
  forall $D \in AltBehDgm(altSplit)$ $SID_state(subj, D)$:= **undef**
  SetCompletionPredicate$_{ExitAltAction}$(*subj*, *altJoin*(*altSplit*))

To Perform a step of AltAction—a step in the subrun of an alternative action node—the subject either will PerformSubDgmStep, i.e. will execute the Behavior as defined for its current state in any of the subdiagrams where it is active, or it will StartNewSubDgm in one of the not yet active alternative behavior diagrams.

Perform(*subj*, AltAction, *state*) =
  PerformSubDgmStep(*subj*, *state*)
    or StartNewSubDgm(*subj*, *state*)}
where
  PerformSubDgmStep(*s*, *n*) =
    choose $D \in ActiveSubDgm(s, n)$ **in** Behavior(s, $SID_state(s, D)$)
  StartNewSubDgm(*s*, *n*) =
    choose $D \in AltBehDgm(n) \setminus ActiveSubDgm(s, n)$
      $SID_state(s, D) := altEntry(D)$
      Start(s, $service(altEntry(D))$, $altEntry(D)$)
  $ActiveSubDgm(s, n) = \{D \in AltBehDgm(n) \mid Active(s, D)\}$
  R **or** S = **choose** $X \in \{R, S\}$ **in** X

Remark. In each step of AltAction the underlying *SID_state* is uniquely determined by the interleaving scheme: it is either the alternative action state itself (when StartNewSubDgm is chosen) or the *SID_state* in the diagram chosen to PerformSubDgmStep, so that it can be computed recursively. Therefore its use in defining *RendezvousWith*... is correct also in the presence of alternative actions.

Remark. The understanding of alternative state computations is that once the alternative clause is *Completed* none of its possibly still non completed subcomputations will be continued. This is guaranteed by the fact that the submachine PerformSubDgmStep is executed (and thus performs a subdiagram step of *subj*ect) only when triggered by Perform in the *subj*ect's *altSplit* state, which however (by definition of Behavior(*subj*, *state*)) is not executed when *Completed* is true.

[26] In the diagram no direct edge connecting the two nodes is drawn, but it is implicit in the parenthesis structure formed by *altSplit* and *altJoin*(*altSplit*).

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011
Copyright 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

Remark. The tool at present does not allow nested alternative clauses, although the specification defined above also works for nested alternative clauses via the $SID_state(s, D)$ notation for subdiagrams D which guarantees that for each diagram D each *subj*ect at any moment is in at most one $SID_state(subj, D)$. If the subdiagrams are properly nested (a condition that is required for alternative behavior diagrams), it is guaranteed by the definition of PERFORM for an ALTACTION that *altSplit* controls the walk of *subj* through the subdiagrams until ALTACTION is *Completed* at *altSplit* so that *subj* can PROCEED to its unique successor state $altJoin(altSplit)$; if one of the behavior subdiagrams of *altSplit* contains an alternative split state $state_1$ with further alternative behavior subdiagrams, both *altSplit* and $state_1$ together control the walk of *subj* through the subsubdiagrams until ALTACTION is *Completed* at $state_1$, etc.[27]

Remark. The specification above makes no assumption neither on the nature or number of the states from where an alternative action node is entered nor on the number of edges leaving an alternative action node or the nature of their target states. For this reason Fig. 6 shows no edge entering *altSplit* and no edge leaving $altJoin(altSplit)$.

Remark. Alternative action nodes can be instantiated by natural constraints on which entry/exit states are compulsory to capture two common business process constructs, namely **and** (where each entry- and exitBox has an x) and **or** (where no entry- but every exitBox has an x). A case of interest for testing purposes is **skip** (where not exitBox has an x).

5 Notational Structuring Concepts

This section deals with notational concepts to structure processes. Some of them can be described by further ASM refinements of the basic constituents of SBDs.

5.1 Macros

The idea underlying the use of macros is to describe once and for all a behavior that can be replicated by insertion of the macro into multiple places. Macros represent a notational device supporting to define processes where instead of rewriting in various places copies of some same subprocess a short (possibly parameterized) name for this subprocess is used in the enclosing process description

Copyright 2011 Carl Hanser Verlag München

[27] Let SBDs $D, D_1, D_2, D_{11}, D_{12}$ be given where D is the main diagram with subdiagrams D_1, D_2 at an alternative action state *altSplit* and where D_1 contains another alternative action $state_1$ with subdiagrams D_{11}, D_{12}. Then the *SID_state* of *subj* first walks through states in D (read: assumes as values of $SID_state(subj) = SID_state(subj, D)$ nodes in D) until it reaches the D-node *altSplit*; *altSplit* controls the walks through $SID_state(subj, D_i)$ states (for $i = 1, 2$), in D_1 until $SID_state(subj, D_1)$ reaches $state_1$. Then *altSplit* and $state_1$ together control the walk through $SID_state(subj, D_{1j})$ (for $j = 1, 2$) until the ALTACTION at node $state_1$ is *Completed*. Then *altSplit* continues to control the walk through $SID_state(subj, D_i)$ states (for $i = 1, 2$) until the ALTACTION at *altSplit* is *Completed*.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

and the subprocess is separately defined once and for all. In the S-BPM context it means to define SBD-macros which can be inserted into given SBDs of possibly different (types of) subjects (participating in one process or even in different processes). The insertion must be supported by a substitution mechanism to replace (some of) the parameters of the macro-SBD by subject types or by concrete subjects that can be assumed to be known in the context of the SBD where the macro-SBD is inserted.

An SBD-macro (which for brevity will be called simply a macro) is defined to be an SBD which is parameterized by finitely many subject types.[28] Usually the first parameter is used to specify the type of a subject into whose SBDs the macro can be inserted. The remaining parameters specify the type of possible communication partners of (subjects of the type of) the first parameter. Through these parameters what is called macro really is a scheme for various macro instances which are obtained by parameter substitution.

To increase the flexibility in the use of macros it is permitted to enter and exit an SBD-macro via finitely many *entryStates* resp. *exitEdges* which can be specified at design time and are pictorially represented by so-called macro tables decorating so-called *macro states* (see Fig. 7). They are required to satisfy some natural conditions (called *Macro Insertion Constraints*) to guarantee that if a *subject* during its walk through D reaches the macro state it will:

- walk via one of the *entryStates* into the macro,
- then walk through the diagram of the macro until it reaches one of the *exitEdges*,
- through the *exitEdge* PROCEED to a state in the enclosing diagram D.

Copyright: 2011 Carl Hanser Verlag München

$entryState_1$	...	$entryState_n$
	M	
$exitEdge_1$	...	$exitEdge_m$

Fig. 7. Macro Table associated to a Macro State

The macro insertion constraints are therefore about how the *entryStates* and *exitEdges* are connected to states of the surrounding *subject* behavior diagram D

[28] This macro definition deliberately privileges the role of subjects, hiding the underlying data structure parameters of an SBD-macro.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

if the macro name is inserted there. We formulate them as constraints for (implicitly) transforming an SBD D where a macro state appears by insertion of the macro SBD at the place of the macro state.

Macro Insertion Constraints When a *macroState* node with SBD-macro M occurs in a subject behavior diagram D, D is (implicitly) transformed into a diagram $D[macroState/M]$ by inserting M for the *macroState* and redirecting the edges entering and exiting *macroState* such that the following conditions are satisfied:

1. Each D-edge targeting the *macroState* must point to exactly one *entryState* in the macro table and is redirected to target in $D[macroState/M]$ this *entryState*, i.e. the state in the subject behavior diagram M where the subject has to PROCEED to upon entering the *macroState* at this *entryState*.
   - There is no other way to enter M than via its *entryStates*, i.e. in the diagram $D[macroState/M]$ each edge leading into M is one of those redirected by constraint 1.
2. Each *exitEdge* in the macro table must be connected in $D[macroState/M]$ to exactly one D-successor state *succ* of the *macroState*, i.e. the state in the enclosing diagram D where to PROCEED to upon exiting the macro SBD M through the *exitEdge*.
   - There is no other way to exit M than via its *exitEdges*, i.e. in the diagram $D[macroState/M]$ each edge leaving the *macroState* node is one of those redirected to satisfy constraint 2.
3. Each *macro exit state* and no other state[29] appears in the macro table as source of one of the *exitEdges*. A state in a macro diagram M is called macro exit state if in M there is no edge leaving that state.

As a consequence of the macro insertion constraints the behavior of an SBD-macro at the place of a *macroState* in an SBD is defined, namely as behavior of the inserted macro diagram.[30] This definition provides a well-defined semantics also to SBDs with well nested macros.

Remark. For defining the abstract meaning of macro behavior it is not necessary to also consider the substitution of some macro parameters by names which are assumed to be known in the enclosing diagram where the macro is inserted. These substitutions, which often are simply renamings, only instantiate the abstract behavior to something (often still abstract but somehow) closer to the to-be-modeled reality.

Copyright 2011 Carl Hanser Verlag München

[29] The second conjunct permits to avoid a global control of when a macro subrun terminates.

[30] Different occurences of the same SBD-macro M at different *macroStates* in an SBD may lead to different executions, due to the possibly different macro tables in those states.

5.2 Interaction View Normalization of Subject Behavior Diagrams

Focus on communication behavior with maximal hiding of internal actions is obtained by the *interaction view* of SBDs (also called *normalized behavior view*) where not only every detail of a function state is hidden (read: its internal PERFORM steps), but also subpaths constituted by sequences of consecutive internal function nodes are compressed into one abstract internal function step. In the resulting *InteractionView*(D) of an SBD D (also called normalized SBD or function compression *FctCompression*(D)) every communication step together with each entry into and exit out of any alternative action state is kept,[31] but every sequence of consecutive function steps appears as compressed into one abstract function step. Thus an interaction view SBD shows only the following items:

- the initial state,
- transitions from internal function states to communication and/or alternative action states,
- transitions from communication or alternative action states,
- the end states.

Since interaction view SBDs are SBDs, their semantics is well-defined by the ASM-interpreter described in the preceding sections. The resulting *interaction view runs*, i.e. runs of a normalized SBD, are distinguished from the standard runs of an SBD by the fact that each time the *subject* PERFORMs an internal action in a state, in the next state it PERFORMs a communication or alternative action (unless the run terminates).

For later use we outline here a normalization algorithm which transforms any SBD D by function compression into a normalized SBD *InteractionView*(D). The idea is to walk through the diagram, beginning at the start node, along any path leading to an end node until all possible paths have been covered and to compress along the way every sequence of consecutive internal function computation steps into one internal function step. Roughly speaking in each step, say m, whenever from a given non-internal *state* through a sequence of internal function nodes a non-internal action or end state $state'$ is reached, an edge from *state* to one internal function *node*—with an appropriately compressed semantically equivalent associated *service*(*node*)—and from there an edge to $state'$ are added to *InteractionView*(D) and the algorithm proceeds in step $m+1$ starting from every node in the set $Frontier_m$ of all such non-internal action or end nodes $state'$ which have not been encountered before—until $Frontier_m$ becomes empty. Some special cases have to be considered due to the presence of alternative action nodes and to the fact that it is permitted that end nodes may have outgoing

[31] Alternative action nodes must remain visible in the interaction view of an SBD because some of their alternative behavior subdiagrams may contain communication states and others not. The other structured states need no special treatment here: multi-process communication states remain untouched by the normalization and macros are considered to have their defining SBD to be inserted when the normalization process starts.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011
Copyright 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

edges, so that the procedure will have to consider also paths starting from end nodes or *altEntry* or *altJoin* states of alternative action subdiagrams.

Start Step. This step starts at the initial *start* state of D. *start* goes as initial state into $InteractionView(D)$. There are two cases to consider.

Case 1. *start* is not an internal function node (read: a communication or alternative action *altSplit* state[32]) or it is an end node of D. Then *start* will not be compressed with other states and therefore will be a starting point for compression rounds in the iteration step. We set $Frontier_1 := \{start\}$ for the iteration steps. If an edge from *start* to *start* is present in D, it is put into $InteractionView(D)$ leaving the service associated to the *start* node in the normalized diagram unchanged.

Case 2. *start* is an internal function node. Then its function may have to be compressed with functions of successive function states. Let $Path_1$ be the set of all paths $state_1, \dots, state_{n+1}$ in D such that $state_1 = start$ and the following **MaximalFunctionSequence** property holds for the path $state_1, \dots, state_{n+1}$:

- for all $1 \leq i \leq n$ $state_i$ is an internal function node with associated service f_i and not an end state of D
- $state_{n+1}$ is an end state of D or not an internal action state.[33]

Then each subpath $state_1, \dots, state_n$ of a path in $Path_1$ (if there are any) is compressed into the *start* node[34] with associated service $(f_1 \circ \dots \circ f_n)$ and put into $InteractionView(D)$ with one edge leading from *start* (which is then also denoted $state_{(1,\dots,n)}$) to $state_{n+1}$. All final nodes $state_{n+1}$ of $Path_1$ elements are put into $Frontier_1$ and thus will be a starting point for iteration steps.

Iteration Step. If $Frontier_m$ is empty, the normalization procedure terminates and the obtained set $InteractionView(D)$ is what is called the interaction view or normalized behavior diagram of D and denoted $InteractionView(D)$.

If $Frontier_m$ is not empty, let $state_0, \dots, state_{n+1}$ be any element in the set $Path_{m+1}$ of all paths in D such that $state_0 \in Frontier_m$ and for the subsequence $state_1, \dots, state_{n+1}$ the MaximalFunctionSequence property holds. In case of an alternative action *altSplit* state in $Frontier_m$, as $state_0$ the $altEntry_i$ state of any alternative behavior subdiagram is taken, so that upon entering an alternative action node the normalization proceeds within the subdiagrams. The auxiliary wait action states $exitBox_i$ are considered as candidates for final nodes $state_{n+1}$ of to-be-compressed subsequences (read: not internal action nodes) so that they survive the compression and can play their role for determining the completion predicate for the alternative action node also in $InteractionView(D)$. The $altJoin(altSplit)$ state is considered like a diagram start node so that it too survives the compression. This realizes that alternative action nodes remain

[32] A start state cannot be an $altJoin(altSplit)$ state because otherwise the diagram would not be well-formed.

[33] The end node clauses in these two conditions guarantee that end nodes survive the normalization.

[34] This guarantees that initial internal function states survive the compression procedure.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

untouched by the normalization procedure, though their subdiagrams are normalized.[35]

If the to-be-compressed internal functions subsequence contains cycles, these cycles are eliminated by replacing recursively every subcycle-free subcycle from $state_i$ to $state_i$ by one node $state_i$ and associated service $(f_i \circ \ldots \circ f_i)$. Then each cycle-free subsequence $state_1, \ldots, state_n$ obtained in this way from a path in $Path_{m+1}$ is further compressed into one node, say $state_{(1,\ldots,n)}$ with associated service $(f_1 \circ \ldots \circ f_n)$ and is put into $InteractionView(D)$ together with two edges, one leading from $state_0$ to $state_{(1,\ldots,n)}$ and one from there to $state_{n+1}$.

All final nodes $state_{n+1}$ of such compressed $Path_{m+1}$ elements which are not in $Frontier_k$ for some $k \leq m$ (so that they have not been visited before by the algorithm) are put into $Frontier_{m+1}$ and thus may become a starting point for another iteration step. In the special case of an alternative action node: if $state_{n+1}$ is an $exitBox_i$ state, $exitBox_i$ is not placed into $Frontier_{m+1}$ because the subdiagram compression stops here. The normalization continues in the enclosing diagram by putting instead $altJoin(altSplit)$ into $Frontier_{m+1}$.

5.3 Process Networks

This section explains a concept which permits to structure processes into hierarchies via communication structure and visibility and access right criteria for processes and/or subprocesses.

Process Networks and their Interaction Diagrams An *S-BPM process network* (shortly called process network) is defined as a set of S-BPM processes. Usually the constituent processes of a process network are focussed on the communication between partner processes and are what we call S-BPM component processes. An *S-BPM component process* (or shortly component) is defined as a pair of an S-BPM process P and a set $ExternalPartnerProc$ of external partner processes which can be addressed from within P. More precisely $ExternalPartnerProc$ consists of pairs $(caller, (P', externalSubj))$ of a *caller*—a distinguished P-subject—and an S-BPM process P' with a distinguished P'-subject $externalSubj$, the communication partner in P' which is addressed from within P by the *caller* and thus for the *caller* appears as external subject whose process typically is not known to the *caller*.

We define that two process network components $(P, (caller, (P', extSubj')))$ and $(P_1, (caller_1, (P'_1, extSubj'_1)))$ (or the corresponding subjects $caller, extSubj'$) are *communication partners* or simply partners (in the network) if the external subject which can be called by the caller in the first process is the one which can call back this caller, formally:

$$P' = P_1 \textbf{ and } extSubj' = caller_1 \textbf{ and } P'_1 = P \textbf{ and } extSubj'_1 = caller$$

[35] The compression algorithm can be further sharpened for alternative action nodes by compressing into one node certain groups of subdiagrams without communication or alternative action nodes.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

A *service process* in a process network is a component process which is communication partner of multiple components in the network, i.e. which can be called from and call back to multiple other component processes in the network. Thus the *ExternalSubj*ect referenced in and representing a service process S for its clients represents a set of external subjects[36], namely the (usually disjoint) union of sets $ExternalSubj(P, S)$, namely the *extSubj*ects of the partner subjects in $caller(P, S)$ which from within their process P call the partner process S by referencing *extSubj*, formally:

$$ExternalSubj(S) = \bigcup_{P \in Partner(S)} ExternalSubj(P, S)$$

Each communication between a client process P and a service process S implies a substitution (usually a renaming) at the service process side of its $ExternalSubj(S)$ by a dedicated element *extSubj* of $ExternalSubj(S, P)$ which is the *extSubj* of an element of the set $caller(P, S)$ of concrete subjects calling S from the client process P.

A special class of S-BPM process networks is obtained by the decomposition of processes into a set of subprocesses. As usual various decomposition layers can be defined, leading to the concepts of horizontal subjects (those which communicate on the same layer) and vertical subjects (those which communicate with subjects in other layers) and to the application of various data sharing disciplines along a layer hierarchy.

An S-BPM process network comes with a graphical representation of its communication partner signature by the so-called *process interaction diagram* (PID), which is an analogue of a SID-diagram lifted from subjects to processes to which the communicating subjects belong. A PID for a process network is defined as a directed graph whose nodes are (names of) network components and whose arcs connect communication partners. The arcs may be labeled with the name of the message type through which the partner is addressed by the caller. A further abstraction of PIDs results if the indication of the communicating subjects is omitted and only the process names are shown.

Observer View Normalization of Subject Behavior Diagrams The interaction view normalization of SBDs defined in Sect. 5.2 can be pushed further by defining an *observer*'s *ObserverView* of the SBD of an observed *subj*ect, where not only internal functions are compressed, but also communication actions of the observed *subj*ect with other partners than the *observer* subject. In defining the normalization of an SBD D into the $ObserverView(observer, D_{subj})$ some attention has to be paid to structured states, namely those with communication alternatives or multiple communication actions and states with alternative actions. To further explain the concept we outline in the following a normalization algorithm which defines this $ObserverView(observer, D_{subj})$.

Copyright: 2011 Carl Hanser Verlag München

[36] For this reason it is called a general external subject.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

In a first step we construct a $CommunicationHiding(observer, D_{subj})$ diagram, also written $D_{subj} \downarrow observer$. It is semantically equivalent to but appears to be more abstract than D. Roughly speaking each communication action in D between the *subj*ect and other partners than the *observer* is hidden as an abstract pseudo-internal function, whose specification hides the original content of the communication action. Then to the resulting SBD the interaction view normalization defined in Sect. 5.2 is applied (where pseudo-internal functions are treated as internal functions). The final result is the *ObserverView* of the original SBD:

$$ObserverView(observer, D_{subj}) = \\ InteractionView(D_{subj} \downarrow observer)$$

The idea for the construction of $D_{subj} \downarrow observer$ is to visit every node in the SBD of *subj*ect once, beginning at the start node and following all possible paths in D, and to hide every encountered not *observer*-related communication action of *subj*ect as a (semantically equivalent) pseudo-internal function step. Since internal function states are not affected by this, it suffices to explain what the algorithm does at (single or multi-) communication nodes or at alternative action nodes. The symmetry in the model between send and receive actions permits to treat communication nodes uniformly as one case.

Case 1. The visited *state* has a send or receive action.

If the *observer* is not a possible communication partner of the *subj*ect in any communication $Alternative(subj, state)$ (Case 1.1), then the entire action in *state* is declared as pseuo-internal function (with its original but hidden semantical effect). If *observer* is a possible communication partner in every communication $Alternative(subj, state)$ (Case 1.2), then the communication action in *state* remains untouched with all its communication alternatives. In both cases the algorithm visits the next state.

We explain below how to compute the property of being a possible communication partner via the type structure of the elements of $Alternative(subj, state)$.

Otherwise (Case 1.3.) split $Alternative(subj, state)$ following the *priority* order into alternating successive segments $alt_i(observer)$ of communication alternatives with *observer* as possible partner and $alt_{i+1}(other)$ of communication alternatives with only *other* possible partners than *observer*. Keep in a *priority* preserving way[37] the *observer* relevant elements of any $alt_i(observer)$ untouched and declare each segment $alt_{i+1}(other)$ as one pseudo-internal function (with the

Copyright Carl Hanser Verlag München

[37] In case different elements are allowed to have the same *priority* there is a further technical complication. For the *priority* preservation one has then to split each $alt_j(other)$ further into three segments of alternatives which have a) the same priority as the last element in the preceding segment $alt_{j-1}(observer)$ (if there is any) resp. b) a higher priority than the last element in the preceding segment $alt_{j-1}(observer)$ and a lower one than the first element in the successor segment $alt_{j+1}(observer)$ (if there is any) resp. c) the same priority as the first element in $alt_{j+1}(observer)$ (if it exists). Each of these three segments must be declared as a pseudo-internal function with corresponding priority.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

original but hidden semantical effect of its elements) which constitutes one alternative of the *subj*ect in this *state* as observable by the *observer* (read: alternative in $CommunicationHiding(observer, D_{subj})$). If an $alt_{i+1}(other)$ segment contains a multi-communication action, the iteration due to the *MultiAction* character of this action remains hidden to the *observer* (read: the pseudo-internal function it will belong to is defined not to be a *MultiAction* in $D_{subj} \downarrow observer$). The function $select_{Alt}$ (and in the *MultiAction* case also the respective constraints) used in this *state* have to be redefined correspondingly to maintain the semantical equivalence of the transformation.

Case 2. The visited *state* is an alternative action state *altSplit*.

Split *AltBehDgm*(*altSplit*) into two subsets Alt_1 of those alternative subdiagrams which contain a communication state with *observer* as possible communication partner and Alt_2 of the other alternative subdiagrams. If Alt_1 is empty (Case 2.1), then the entire alternative action structure between *altSplit* and *altJoin*(*altSplit*) (comprising the alternative subdiagrams corresponding to this *state*) is collapsed into one *state* with a pseudo-internal function, which is specified to have its original semantical effect. All edges into any *entryBox* or out of any *exitBox* become an edge into resp. out of *state* and the algorithm visits the next state. If Alt_2 is empty (Case 2.2), then the alternative action *state* remains untouched with all its alternative subdiagrams and the algorithm visits each *altEntry* state. Once the algorithm has visited each node in each subdiagram, it proceeds from the *altJoin*(*altSplit*) state to any of its successor states.

Otherwise (Case 1.3.) the alternative action node structure formed by *altSplit* and the corresponding *altJoin*(*altSplit*) state remains, but the entire set Alt_2 of subdiagrams without communication with the *observer* is compressed into one new state: it is entered from an *entryBox* and exited from an *exitBox* (where all edges into resp. out of the boxes of Alt_2 elements are redirected) and has as associated service a pseudo-internal function, which is specified to have its original semantical effect. Then the algorithm visits each *altEntry* state of each Alt_1 element. Once the algorithm has visited each node in the subdiagram of each Alt_1 element, it proceeds from the *altJoin*(*altSplit*) state to any of its successor states.

It remains to explain how to compute whether *observer* is a possible communication partner in a communication *state* of the observed *subj*ect behavior diagram D_{subj}.

Case 1: *state* is a send state (whether canceling or blocking, synchronous or asynchronous, *Send*(*Single*) or *Send*(*Multi*)). Then *observer* is a possible communication partner of *subj* in this *state* if and only if $observer = receiver(alt)$ for some $alt \in alternative(subj, state)$.

Case 2: *state* is a receive state. Then *observer* is a possible communication partner of *subj* in this *state* if and only if the following property holds, where D_o denotes the SBD of the *observer*:

forsome $alt \in alternative(subj, state)$
forsome send state $state' \in D_o$

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

forsome $alt' \in alternative(observer, state')$
$alt \in \{any, observer\}$ **and** $subj \in PossibleReceiver(alt')$[38]
or forsome $type$ $alt = type = alt'$ **and** $subj \in PossibleReceiver(alt')$
or forsome $type$ $alt = (type, observer)$ **and**
$alt' \in \{type, (type, subj)\}$ **and** $subj \in PossibleReceiver(alt')$
where
$subj \in PossibleReceiver(alt')$ if and only if
$alt' = any$ **or** $receiver(alt') = subj$

Remark. The above algorithm makes clear that different observers may have a different view of a same diagram.

6 Two model extension disciplines

In this section we define two composition schemes for S-BPM processes which build upon the simple logical foundation of the semantics of S-BPM exposed in the preceding sections. They support the S-BPM discipline for controlled stepwise development of complex processes out of basic modular components and offer in particular a clean methodological separation of normal and exceptional behavior. More precisely they come as rigorous methods to enrich a given S-BPM process by new features in a purely incremental manner, typically by extending a given SBD D by an SBD D' with some desired additional process behavior without withdrawing or otherwise contradicting the original $\text{Behavior}_{subj}(D)$. This conservative model extension approach permits a separate analysis of the original and the extended system behavior and thus contributes to split a complex system into a manageable composition of manageable components. The separation of given and added (possibly exception) behavior allows one also to change the implementation of the two independently of each other.

The difference between the two model extension methods is of pragmatic nature. The so-called *Interrupt Extension* has its roots in and is used like the interrupt handling mechanism known from operating systems and the exception handling pendant in high-level programmming languages. The so-called *Behavior Extension* is used to stepwise extend (what is considered as) 'normal' behavior by additional features. Correspondingly the two extension methods act at different levels of the S-BPM interpreter; the Interrupt Extension conditions at the SID-level the 'normal' execution of $\text{Behavior}(subj, state)$ by the absence of interrupting events and calls an interrupt handler if an interruption is triggered whereas the Behavior Extension enriches the 'normal' execution of $\text{Behavior}_{subj}(D)$ by new ways to Proceed from $\text{Behavior}(subj, state)$ to the next state.

Copyright 2011 Carl Hanser Verlag München

[38] The second conjunct implies that *observer* is not considered to be a possible communication partner of *subj* in *state* if *subj* in this *state* is ready to receive a message from the *observer* but the *observer*'s SBD has no send state with a send alternative where the *subj*ect could be the receiver of the *msgToBeSent*.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

6.1 Interrupt Extension

The Interrupt Extension method introduces a conservative form of exception handling in the sense that it transforms any given SBD D in such a way that the behavior of the transformed diagram remains unchanged as long as no exceptions occur (read: as long as there are no interrupts), adding exception handling in case an exception event happens. To specify how exceptions are thrown (read: how interrupts are triggered) it suffices to consider here externally triggered interrupts because internal interrupt triggers concerning actions to-be-executed by a subject are explicitly modeled for communication actions Send/Receive in blocking Alternative Rounds (see Fig. 2 in Sect. 3.1) and are treated for internal functions through the specification of their PERFORM component. External interrupt triggers concerning the action currently PERFORMed by a *subj*ect are naturally integrated into the S-BPM model via a set *InterruptKind* of kinds (pairs of sender and message type) of *InterruptMsg*s arriving in *inputPool*(*subj*) independently of whether *subj*ect currently is ready to receive a message. It suffices to

- guarantee that elements of *InterruptMsg* are never *Blocked* in any input pool, so that at each moment every potential *interruptOriginator*—the sender of an *interruptMsg*—can PASS(*interruptMsg*) to the input pool of the receiving subject,[39]
- give priority to the execution of the interrupt handling procedure by the receiver *subj*ect, interrupting the PERFORMance of its current action when an *interruptMsg* arrives in the *inputPool*(*subj*). This is achieved through the INTERRUPTBEHAVIOR(*subj*, *state*) rule defined below which is a conservative extension of the BEHAVIOR(*subj*, *state*) rule defined in Sect. 2.2. This means that we can locally confine the extension, namely to an incremental modification of the interpreter rule for the new kind of interruptable SBD-states.

Thus the SBD-transformation *InterruptExtension* defined below has the following three arguments:

- A to be transformed SBD D with a set *InterruptState* of D-states s_i ($1 \leq i \leq n$) where an interrupt may happen so that for such states a new rule INTERRUPTBEHAVIOR(*subj*, *state*) must be defined which incrementally extends the rule BEHAVIOR(*subj*, *state*).
- A set *InterruptKind*(s_i) of indexed pairs $interrupt_j$ ($1 \leq j \leq m$) of sender and message type of interrupt messages to which *subj*ect has to react when in state s_i.
- An interrupt handling SBD D' the *subj*ect is required to execute immediately when an *interruptMsg* appears in its input pool, together with a set

[39] In the presence of the input pool default row *any any maxSize Blocking* it suffices to require that every input pool constraint table has a penultimate default interrupt msg row of form *interruptOriginator type*(*interruptMsg*) *maxSize Drop* with associated *Drop* action *DropYoungest* or *DropOldest*.

InterruptProcEntry of edges $arc_{i,j}$ without source node, with target node in D' and with associated $ExitCond_{i,j}$.[40]

InterruptExtension when applied to $(D, InterruptState)$, *InterruptKind* and the exception procedure $(D', InterruptProcEntry)$ joins the two SBDs into one graph D^*:

$$D^* = D \cup D' \cup Edges_{D,D'}$$

where $Edges_{D,D'}$ is defined as set of edges (called again) $arc_{i,j}$ connecting in D^* the source node s_i in D with the $target(arc_{i,j})$ node in D' where $j = indexOf(e, InterruptKind(s_i))$ for any $e \in InterruptKind$. $\text{BEHAVIOR}(D^*)$ is defined as in Sect. 2.2 from $\text{BEHAVIOR}_D(subj, state)$ with the following extension $\text{INTERRUPTBEHAVIOR}_{D^*}$ of $\text{BEHAVIOR}_D(subj, s_i)$ for *InterruptStates* s_i of D, whereas $\text{BEHAVIOR}(subj, state)$ remains unchanged for the other D *states* and for *states* of D'—which are assumed to be disjoint from those of D:[41]

$$\text{BEHAVIOR}_{D^*}(subj, state) = \ // \text{ Case of InterruptExtension}(D, D')$$
$$\begin{cases} \text{BEHAVIOR}_D(subj, state) & \textbf{if } state \in D \setminus InterruptState \\ \text{BEHAVIOR}_{D'}(subj, state) & \textbf{if } state \in D' \\ \text{INTERRUPTBEHAVIOR}(subj, state) & \textbf{if } state \in InterruptState \end{cases}$$

$\text{INTERRUPTBEHAVIOR}(subj, s_i) =$ // at InterruptState s_i
  if $SID_state(subj) = s_i$ **then**
    if $InterruptEvent(subj, s_i)$ **then**
      choose $msg \in InterruptMsg(s_i) \cap inputPool(subj)$[42]
      let $j = indexOf(interruptKind(msg), InterruptKind(s_i))$
        $handleState = target(arc_{i,j})$
          $\text{PROCEED}(subj, service(handleState), handleState)$
          $\text{DELETE}(msg, inputPool(subj))$
    else $\text{BEHAVIOR}_D(subj, s_i)$
where
  $InterruptEvent(subj, s_i)$ iff
    forsome $m \in InterruptMsg(s_i)$ $m \in inputPool(subj)$

[40] This includes the special case $m = 1$ where the (entry into the) interrupt handling procedure depends only on the happening of an interrupt regardless of its kind. The general case with multiple entries (or equivalently multiple exception handling procedures each with one entry) prepare the ground for an easy integration of compensation procedures as part of exception handling, which typically depend on the state where the exception happens and on the kind of interrupt (pair of originator and type of the interrupt message).

[41] This does not exclude the possibility that some edges in D' have as target a node in D, as is the case when the exception handling procedure upon termination leads back to normal execution.

[42] Note that in each step *subj* can react only to one out of possibly multiple interrupt messages present in its *inputPool*(*subj*). If one wants to establish a hierarchy among those a priority function is needed to regulate the selection procedure.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011
Copyright 2011 Carl Hanser Verlag München

When no confusion is to be feared we write again BEHAVIOR$(subj, s_i)$ also for INTERRUPTBEHAVIOR$(subj, s_i)$.

Remark. The definition of INTERRUPTBEHAVIOR implies that if during the execution of the exception handling procedure described by D' *subj*ect encounters an interrupt event inD', it will start to execute the handling procedure D'' for the new exception, similar to the exception handling mechanism in Java [4, Fig.6.2].

6.2 Behavior Extension

The SBD-transformation method *BehaviorExtension* has the following two arguments:

- A to be transformed SBD D with a set *ExtensionState* of D-states s_i $(1 \leq i \leq n)$ where a new behavior is added to be possibly executed if selected by $select_{Edge}$ in BEHAVIOR$(subj, s_i)$ when exiting s_i upon completion of its associated service.
- An SBD D' (assumed to be disjoint from D) which describes the new behavior the *subj*ect will execute when the new behavior is selected to be executed next. To enter D' from extension states in D we use (in analogy to *InterruptProcEntry*) a set *AddedDgmEntry* of edges arc_i without source node and with target node in D' and associated $ExitCond_i$.

BehaviorExtension applied to $(D, ExtensionState)$ and $(D', AddedDgmEntry)$ joins the two SBDs into one graph D^+:

$$D^+ = D \cup D' \cup Edges_{D,D'}$$

where $Edges_{D,D'}$ is defined as set of edges (called again) arc_i connecting in D^+ the source node s_i in D with the $target(arc_i)$ node in D'.

BEHAVIOR(D^+) can be defined as in Sect. 2.2 from BEHAVIOR$(subj, state)$ for *state*s in D resp. D' but with the selection function $select_{Edge}$ extended for *ExtensionState* nodes s_i to include in its domain arc_i with the associated $ExitCond_i$. In this way new D'-behavior becomes possible which can be analyzed separately from the original D-behavior.

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

Copyright 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

7 S-BPM Interpreter in a Nutshell

Collection of the ASM rules for the high-level subject-oriented interpreter model for the semantics of the S-BPM constructs.

7.1 Subject Behavior Diagram Interpretation

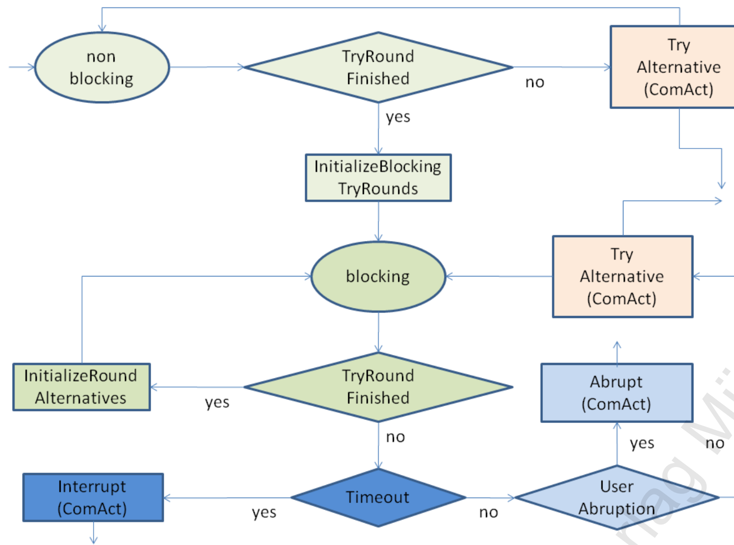
$\textsc{Behavior}_{subj}(D) = \{\textsc{Behavior}(subj, node) \mid node \in Node(D)\}$

$\textsc{Behavior}(subj, state) =$
  if $SID_state(subj) = state$ **then**
    if $Completed(subj, service(state), state)$ **then**
      let $edge =$
        $select_{Edge}(\{e \in OutEdge(state) \mid ExitCond(e)(subj, state)\})$
          $\textsc{Proceed}(subj, service(target(edge)), target(edge))$
    else $\textsc{Perform}(subj, service(state), state)$
  where
    $\textsc{Proceed}(subj, X, node) =$
      $SID_state(subj) := node$
      $\textsc{Start}(subj, X, node)$

7.2 Alternative Send/Receive Round Interpretation

$\textsc{Perform}(subj, \textsc{ComAct}, state) =$
  if $NonBlockingTryRound(subj, state)$ **then**
    if $TryRoundFinished(subj, state)$ **then**
      $\textsc{InitializeBlockingTryRounds}(subj, state)$
    else $\textsc{TryAlternative}_{ComAct}(subj, state)$
  if $BlockingTryRound(subj, state)$ **then**
    if $TryRoundFinished(subj, state)$
      then $\textsc{InitializeRoundAlternatives}(subj, state)$
      else
        if $Timeout(subj, state, timeout(state))$ **then**
          $\textsc{Interrupt}_{ComAct}(subj, state)$
        elseif $UserAbruption(subj, state)$
          then $\textsc{Abrupt}_{ComAct}(subj, state)$
          else $\textsc{TryAlternative}_{ComAct}(subj, state)$

Copyright 2011 Carl Hanser Verlag München



Interpretation of Auxiliary Macros

$\textsc{Start}(subj, \textsc{ComAct}, state) =$
  $\textsc{InitializeRoundAlternatives}(subj, state)$
  $\textsc{InitializeExit\&CompletionPredicates}_{ComAct}(subj, state)$
  $\textsc{EnterNonBlockingTryRound}(subj, state)$
where
  $\textsc{InitializeRoundAlternatives}(subj, state) =$
    $RoundAlternative(subj, state) := Alternative(subj, state)$
  $\textsc{InitializeExit\&CompletionPredicates}_{ComAct}(subj, state) =$
    $\textsc{InitializeExitPredicates}_{ComAct}(subj, state)$
    $\textsc{InitializeCompletionPredicate}_{ComAct}(subj, state)$
  $\textsc{InitializeExitPredicates}_{ComAct}(subj, state) =$
    $NormalExitCond(subj, \textsc{ComAct}, state) := false$
    $TimeoutExitCond(subj, \textsc{ComAct}, state) := false$
    $AbruptionExitCond(subj, \textsc{ComAct}, state) := false$
  $\textsc{InitializeCompletionPredicate}_{ComAct}(subj, state) =$
    $Completed(subj, \textsc{ComAct}, state) := false$

$[Non]BlockingTryRound(subj, state) =$
  $tryMode(subj, state) = [non]blocking$
$\textsc{Enter}[\textsc{Non}]\textsc{BlockingTryRound}(subj, state) =$
  $tryMode(subj, state) := [non]blocking$
$TryRoundFinished(subj, state) =$
  $RoundAlternatives(subj, state) = \emptyset$
$\textsc{InitializeBlockingTryRounds}(subj, state) =$
  $\textsc{EnterBlockingTryRound}(subj, state)$

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011
Copyright 2011 Carl Hanser Verlag München

$\quad$ INITIALIZEROUNDALTERNATIVES$(subj, state)$
$\quad$ SETTIMEOUTCLOCK$(subj, state)$
SETTIMEOUTCLOCK$(subj, state) =$
$\quad blockingStartTime(subj, state) := now$
$Timeout(subj, state, time) =$
$\quad now \geq blockingStartTime(subj, state) + time$

INTERRUPT$_{ComAct}(subj, state) =$
$\quad$ SETCOMPLETIONPREDICATE$_{ComAct}(subj, state)$
$\quad$ SETTIMEOUTEXIT$_{ComAct}(subj, state)$
SETCOMPLETIONPREDICATE$_{ComAct}(subj, state) =$
$\quad Completed(subj,$ COMACT$, state) := true$
SETTIMEOUTEXIT$_{ComAct}(subj, state) =$
$\quad TimeoutExitCond(subj,$ COMACT$, state) := true$
ABRUPT$_{ComAct}(subj, state) =$
$\quad$ SETCOMPLETIONPREDICATE$_{ComAct}(subj, state)$
$\quad$ SETABRUPTIONEXIT$_{ComAct}(subj, state)$

7.3 MsgElaboration Interpretation for Multi Send/Receive

TRYALTERNATIVE$_{ComAct}(subj, state) =$
$\quad$ CHOOSE&PREPAREALTERNATIVE$_{ComAct}(subj, state)$
$\quad\quad$ **seq** TRY$_{ComAct}(subj, state)$

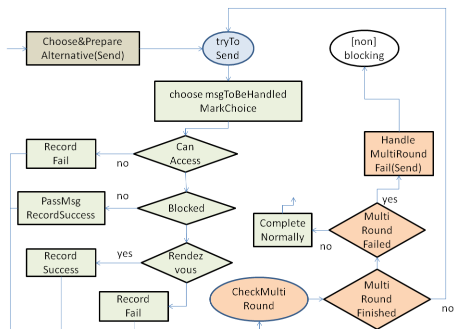
CHOOSE&PREPAREALTERNATIVE$_{ComAct}(subj, state) =$
$\quad$ **let** $alt = select_{Alt}(RoundAlternative(subj, state), priority(state))$
$\quad\quad$ PREPAREMSG$_{ComAct}(subj, state, alt)$
$\quad\quad$ MANAGEALTERNATIVEROUND$(alt, subj, state)$
$\quad\quad$ **where**
$\quad\quad\quad$ MANAGEALTERNATIVEROUND$(alt, subj, state) =$
$\quad\quad\quad\quad$ MARKSELECTION$(subj, state, alt)$
$\quad\quad\quad\quad$ INITIALIZEMULTIROUND$_{ComAct}(subj, state)$
$\quad\quad\quad$ MARKSELECTION$(subj, state, alt) =$
$\quad\quad\quad\quad$ DELETE$(alt, RoundAlternative(subj, state))$

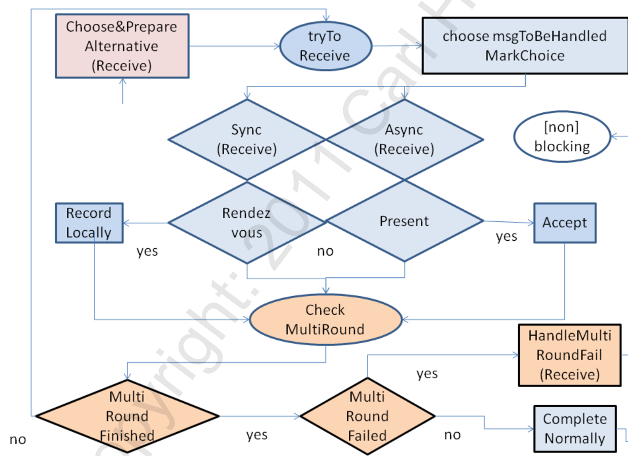
PREPAREMSG$_{ComAct}(subj, state, alt) =$
$\quad$ **forall** $1 \leq i \leq mult(alt)$
$\quad$ **if** $ComAct = Send$ **then**
$\quad\quad$ **let** $m_i = composeMsg(subj, msgData(subj, state, alt), i)$
$\quad\quad\quad MsgToBeHandled(subj, state) := \{m_1, \ldots, m_{mult(alt)}\}$
$\quad$ **if** $ComAct = Receive$ **then**
$\quad\quad$ **let** $m_i = select_{MsgKind(subj,state,alt,i)}(ExpectedMsgKind(subj, state, alt))$
$\quad\quad\quad MsgToBeHandled(subj, state) := \{m_1, \ldots, m_{mult(alt)}\}$

Copyright 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

7.4 Multi Send/Receive Round Interpretation





Copyright: 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

```
TRY_ComAct(subj, state) =
  choose m ∈ MsgToBeHandled(subj, state)
    MARKCHOICE(m, subj, state)
    if ComAct = Send then
      let receiver = receiver(m), pool = inputPool(receiver)
        if not CanAccess(subj, pool) then
          CONTINUEMULTIROUND_Fail(subj, state, m)
        else TRY_Async(Send)(subj, state, m)
    if ComAct = Receive then
      if Async(Receive)(m) then TRY_Async(Receive)(subj, state, m)
      if Sync(Receive)(m) then TRY_Sync(Receive)(subj, state, m)
where
  MARKCHOICE(m, subj, state) =
    DELETE(m, MsgToBeHandled(subj, state))
    currMsgKind(subj, state) := m


TRY_Async(ComAct)(subj, state, m) =
  if PossibleAsync_ComAct(subj, m) // async communication possible
    then ASYNC(ComAct)(subj, state, m)
    else
      if ComAct = Receive then
        CONTINUEMULTIROUND_Fail(subj, state, m)
      if ComAct = Send then TRY_Sync(ComAct)(subj, state, m)
TRY_Sync(ComAct)(subj, state, m) =
  if PossibleSync_ComAct(subj, m) // sync communication possible
    then SYNC(ComAct)(subj, state, m)
    else CONTINUEMULTIROUND_Fail(subj, state, m)
```

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

Copyright: 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

7.5 Actual Send Interpretation

$\text{ASYNC}(\mathit{Send})(\mathit{subj}, \mathit{state}, \mathit{msg}) =$
  $\text{PASSMSG}(\mathit{msg})$
  $\text{CONTINUEMULTIROUND}_{\mathit{Success}}(\mathit{subj}, \mathit{state}, \mathit{msg})$
where
  $\text{PASSMSG}(\mathit{msg}) =$
    let $\mathit{pool} = \mathit{inputPool}(\mathit{receiver}(\mathit{msg}))$
      $\mathit{row} = \mathit{first}(\{r \in \mathit{constraintTable}(\mathit{pool}) \mid$
        $\mathit{ConstraintViolation}(\mathit{msg}, r)\})$
          if $\mathit{row} \neq$ **undef and** $\mathit{action}(\mathit{row}) \neq \mathit{DropIncoming}$
            then $\text{DROP}(\mathit{action})$
          if $\mathit{row} =$ **undef or** $\mathit{action}(\mathit{row}) \neq \mathit{DropIncoming}$ **then**
            $\text{INSERT}(\mathit{msg}, \mathit{pool})$
            $\mathit{insertionTime}(\mathit{msg}, \mathit{pool}) := \mathit{now}$
  $\text{DROP}(\mathit{action}) =$
    if $\mathit{action} = \mathit{DropYoungest}$ **then** $\text{DELETE}(\mathit{youngestMsg}(\mathit{pool}), \mathit{pool})$
    if $\mathit{action} = \mathit{DropOldest}$ **then** $\text{DELETE}(\mathit{oldestMsg}(\mathit{pool}), \mathit{pool})$
  $\mathit{PossibleAsync}_{\mathit{Send}}(\mathit{subj}, \mathit{msg})$ iff **not** $\mathit{Blocked}(\mathit{msg})$
  $\mathit{Blocked}(\mathit{msg})$ iff
    let $\mathit{row} = \mathit{first}(\{r \in \mathit{constraintTable}(\mathit{inputPool}(\mathit{receiver}(\mathit{msg}))) \mid$
      $\mathit{ConstraintViolation}(\mathit{msg}, r)\})$
        $\mathit{row} \neq$ **undef and** $\mathit{action}(\mathit{row}) = \mathit{Blocking}$

$\text{SYNC}(\mathit{Send})(\mathit{subj}, \mathit{state}, \mathit{msg}) =$
  $\text{CONTINUEMULTIROUND}_{\mathit{Success}}(\mathit{subj}, \mathit{state}, \mathit{msg})$
$\mathit{PossibleSync}_{\mathit{Send}}(\mathit{subj}, \mathit{msg})$ iff $\mathit{RendezvousWithReceiver}(\mathit{subj}, \mathit{msg})$

$\mathit{RendezvousWithReceiver}(\mathit{subj}, \mathit{msg})$ iff
  $\mathit{tryMode}(\mathit{rec}) = \mathit{tryToReceive}$ **and** $\mathit{Sync}(\mathit{Receive})(\mathit{currMsgKind})$
    and $\mathit{SyncSend}(\mathit{msg})$ **and** $\mathit{Match}(\mathit{msg}, \mathit{currMsgKind})$
where
  $\mathit{rec} = \mathit{receiver}(\mathit{msg}),\ \mathit{recstate} = \mathit{SID_state}(\mathit{rec})$
  $\mathit{currMsgKind} = \mathit{currMsgKind}(\mathit{rec}, \mathit{recstate})$
  $\mathit{blockingRow} =$
    $\mathit{first}(\{r \in \mathit{constraintTable}(\mathit{rec}) \mid \mathit{ConstraintViolation}(\mathit{msg}, r)\})$
  $\mathit{SyncSend}(\mathit{msg})$ iff $\mathit{size}(\mathit{blockingRow}) = 0$

Copyright 2011 Carl Hanser Verlag München

7.6 Actual Receive Interpreation

$\textsc{Async}(\mathit{Receive})(\mathit{subj}, \mathit{state}, \mathit{msg}) =$
$\quad \textsc{Accept}(\mathit{subj}, \mathit{msg})$
$\quad \textsc{ContinueMultiRound}_{\mathit{Success}}(\mathit{subj}, \mathit{state}, \mathit{msg})$
where
$\quad \textsc{Accept}(\mathit{subj}, m) =$
$\quad\quad$ **let** $\mathit{receivedMsg} =$
$\quad\quad\quad \mathit{select}_{\mathit{ReceiveOfKind}(m)}(\{\mathit{msg} \in \mathit{inputPool}(\mathit{subj}) \mid \mathit{Match}(\mathit{msg}, m)\})$
$\quad\quad\quad\quad \textsc{RecordLocally}(\mathit{subj}, \mathit{receivedMsg})$
$\quad\quad\quad\quad \textsc{Delete}(\mathit{receivedMsg}, \mathit{inputPool}(\mathit{subj}))$
$\quad \mathit{Async}(\mathit{Receive})(m)$ iff $\mathit{commMode}(m) = \mathit{async}$
$\quad \mathit{PossibleAsync}_{\mathit{Receive}}(\mathit{subj}, m)$ iff $\mathit{Present}(m, \mathit{inputPool}(\mathit{subj}))$
$\quad \mathit{Present}(m, \mathit{pool})$ iff **forsome** $\mathit{msg} \in \mathit{pool}$ $\mathit{Match}(\mathit{msg}, m)$

$\textsc{Sync}(\mathit{Receive})(\mathit{subj}, \mathit{state}, \mathit{msgKind}) =$
$\quad$ **let** $P = \mathit{inputPool}(\mathit{subj}),\ \mathit{sender} = \iota s(\mathit{CanAccess}(s, P))$
$\quad\quad \textsc{RecordLocally}(\mathit{subj}, \mathit{msgToBeSent}(\mathit{sender}, \mathit{SID_state}(\mathit{sender}))$
$\quad\quad \textsc{ContinueMultiRound}_{\mathit{Success}}(\mathit{subj}, \mathit{state}, \mathit{msgKind})$
$\mathit{Sync}(\mathit{Receive})(\mathit{msgKind})$ iff $\mathit{commMode}(\mathit{msgKind}) = \mathit{sync}$
$\mathit{PossibleSync}_{\mathit{Receive}}(\mathit{subj}, \mathit{msgKind})$ iff
$\quad \mathit{RendezvousWithSender}(\mathit{subj}, \mathit{msgKind})$

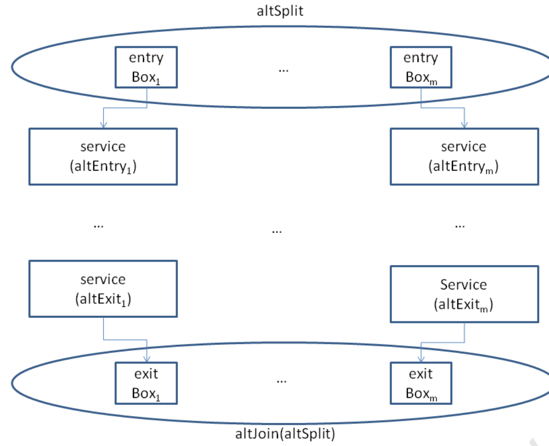
$\mathit{RendezvousWithSender}(\mathit{subj}, \mathit{msgKind})$ iff
$\quad \mathit{Sync}(\mathit{Receive})(\mathit{msgKind})$ **and**
$\quad\quad$ **let** $\mathit{sender} = \iota s(\mathit{CanAccess}(s, \mathit{inputPool}(\mathit{subj}))$
$\quad\quad$ **let** $\mathit{msgToBeSent} = \mathit{msgToBeSent}(\mathit{sender}, \mathit{SID_state}(\mathit{sender}))$
$\quad\quad\quad \mathit{tryMode}(\mathit{sender}) = \mathit{tryToSend}$ **and** $\mathit{SyncSend}(\mathit{msgToBeSent})$
$\quad\quad\quad\quad$ **and** $\mathit{Match}(\mathit{msgToBeSent}, \mathit{msgKind})$

Copyright 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

7.7 Alternative Action Interpretation



$\textsc{Start}(subj, \textsc{AltAction}, altSplit) =$
  forall $D \in AltBehDgm(altSplit)$
    if $Compulsory(altEntry(D))$ **then**
      $SID_state(subj, D) := altEntry(D)$
      $\textsc{Start}(subj, service(altEntry(D)), altEntry(D))$
    else $SID_state(subj, D) :=$ **undef**

$\textsc{Perform}(subj, \textsc{AltAction}, state) =$
  $\textsc{PerformSubDgmStep}(subj, state)$
    or $\textsc{StartNewSubDgm}(subj, state)\}$
where
  $\textsc{PerformSubDgmStep}(s, n) =$
    choose $D \in ActiveSubDgm(s, n)$ **in** $\textsc{Behavior}(s, SID_state(s, D))$
  $\textsc{StartNewSubDgm}(s, n) =$
    choose $D \in AltBehDgm(n) \setminus ActiveSubDgm(s, n)$
      $SID_state(s, D) := altEntry(D)$
      $\textsc{Start}(s, service(altEntry(D)), altEntry(D))$
  $ActiveSubDgm(s, n) = \{D \in AltBehDgm(n) \mid Active(s, D)\}$
  R **or** $S =$ **choose** $X \in \{R, S\}$ **in** X

$Completed(subj, \textsc{AltAction}, altSplit)$ iff
  forall $D \in AltBehDgm(altSplit)$
    if $Compulsory(altExit(D))$ **and** $Active(subj, D)$
      then $SID_state(subj, D) = exitBox(D)$
where
  $Active(subj, D)$ iff $SID_state(subj, D) \neq$ **undef**

Copyright 2011 Carl Hanser Verlag München

Copyright: 2011 Carl Hanser Verlag München
Auszug aus: Fleischmann, A.; Schmidt, W.; Stary, Ch.; Obermeier, S.; Börger, E.: Subjektorientiertes Prozessmanagement. Hanser, München 2011

Auxiliary Wait/Exit Rule Interpretation

$\text{Start}(subj, \text{AltActionWait}, exitBox) =$
$\quad \text{InitializeCompletionPredicate}_{AltActionWait}(subj, exitBox)$
$\text{Perform}(subj, \text{AltActionWait}, exitBox) = \textbf{skip}$

$\text{Start}(subj, \text{ExitAltAction}, altJoin(altSplit)) = \textbf{skip}$
$\text{Perform}(subj, \text{ExitAltAction}, altJoin) =$
$\quad \textbf{forall } D \in AltBehDgm(altSplit)\ SID_state(subj, D) := \textbf{undef}$
$\quad \text{SetCompletionPredicate}_{ExitAltAction}(subj, altJoin(altSplit))$

7.8 Interrupt Behavior

$$\text{Behavior}_{D^*}(subj, state) = \quad // \text{ Case of InterruptExtension}(D, D')$$
$$\begin{cases} \text{Behavior}_D(subj, state) & \textbf{if } state \in D \setminus InterruptState \\ \text{Behavior}_{D'}(subj, state) & \textbf{if } state \in D' \\ \text{InterruptBehavior}(subj, state) & \textbf{if } state \in InterruptState \end{cases}$$

$\text{InterruptBehavior}(subj, s_i) = \quad // \text{ at InterruptState } s_i$
$\quad \textbf{if } SID_state(subj) = s_i \textbf{ then}$
$\quad\quad \textbf{if } InterruptEvent(subj, s_i) \textbf{ then}$
$\quad\quad\quad \textbf{choose } msg \in InterruptMsg(s_i) \cap inputPool(subj)$[43]
$\quad\quad\quad \textbf{let } j = indexOf(interruptKind(msg), InterruptKind(s_i))$
$\quad\quad\quad\quad handleState = target(arc_{i,j})$
$\quad\quad\quad\quad\quad \text{Proceed}(subj, service(handleState), handleState)$
$\quad\quad\quad\quad\quad \text{Delete}(msg, inputPool(subj))$
$\quad\quad \textbf{else } \text{Behavior}_D(subj, s_i)$
$\textbf{where}$
$\quad InterruptEvent(subj, s_i) \text{ iff}$
$\quad\quad \textbf{forsome } m \in InterruptMsg(s_i)\ \ m \in inputPool(subj)$

References

1. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
2. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
3. D. E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information at Stanford/ California, 1992.
4. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

Appeared as appendix in:
A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, E. Börger:
Subjektorientiertes Prozessmanagement. Hanser-Verlag, München, 2011.

[43] Note that in each step *subj* can react only to one out of possibly multiple interrupt messages present in its *inputPool*(*subj*). If one wants to establish a hierarchy among those a priority function is needed to regulate the selection procedure.

Copyright: 2011 Carl Hanser Verlag München