Albert Fleischmann
Werner Schmidt
Christian Stary
Stefan Obermeier
Egon Börger

Subjektorientiertes Prozessmanagement

Mitarbeiter einbinden, Motivation und Prozessakzeptanz steigern

Aktualisierter Anhang:

A Subject-Oriented Interpreter Model for S-BPM

In der Buchversion hat sich leider der Fehlerteufel eingeschlichen.
Bitte verwenden Sie diese Version.

A Subject-Oriented Interpreter Model for S-BPM

We develop in this appendix a high-level subject-oriented interpreter model for the semantics of the S-BPM constructs presented in this book. To directly and faithfully reflect the basic constituents of S-BPM, namely *communicating agents* which can perform arbitrary *actions* on arbitrary *objects*, Abstract State Machines are used which explicitly contain these three conceptual ingredients.

1 Introduction

Subject-oriented Business Process Modeling (S-BPM) is characterized by the use of three fundamental natural language concepts to describe distributed processes: actors (called *subjects*) which perform arbitrary *actions* on arbitrary *objects* and in particular communicate with other subjects in the process, computationally speaking agents which perform abstract data type operations and send messages to and receive messages from other process agents. We provide here a mathematically precise definition for the semantics of S-BPM processes which directly and faithfully reflects these three constituent S-BPM concepts and supports the methodological goal pursued in this book to lead the reader through a precise natural language description to a reliable understanding of S-BPM concepts and techniques.

The challenge consists in building a scientifically solid S-BPM model which faithfully captures and links the understanding of S-BPM concepts by the different stakeholders and thus can serve as basis for the communication between them: analysts and operators on the process design and management side, IT technologists and programmers on the implementation side, users (suppliers and customers) on the application side. To make a transparent, sufficiently precise and easily maintainable documentation of the meaning of S-BPM concepts available which expresses a common understanding of the different stakeholders we have to *start from scratch*, explaining the S-BPM constructs as presented in this book without dwelling upon any extraneous (read: not business process specific) technicality of the underlying computational paradigm.

To brake unavoidable business process specific complexity into small units a human mind can grasp reliably we use a *feature-based* approach, where the meaning of the involved concepts is defined itemwise, construct by construct. For each investigated construct we provide a dedicated set of simple IF-THEN-descriptions (so-called behavior rules) which abstractly describe the operational interpretation of the construct.[1] The feature-based approach is enhanced by the systematic use of *stepwise refinement* of abstract operational descriptions.

[1] This rigorous operational character of the descriptions offers the possibility to use them as a reference model for both simulation (testing) and verification (logical analysis of properties of interest) of classes of S-BPM processes.

Last but not least, to cope with the distributed and heterogeneous character of the large variety of cooperating S-BPM processes, it is crucial that the model of computation which underlies the descriptions supports both *true concurrency* (most general scheduling schemes) and *heterogeneous state* (most general data structures covering the different application domain elements).

For these reasons we use the method of Abstract State Machines (ASMs) [2], which supports feature and refinement based descriptions[2] of heterogeneous distributed processes and in particular allows one to view interacting subjects as rule executing communicating agents (in software terms: multiple threads each executing specific actions), thus matching the fundamental view of the S-BPM approach to business processes.

Technically speaking the ASM method expects from the reader only some experience in process-oriented thinking which supports an understanding of so-called transition rules (also called ASM rules) of form

if *Condition* **then** ACTION

prescribing an ACTION to be undertaken if some event happens; happening of events is expressed by corresponding *Condition*s (also called rule *guards*) becoming true. Using ASMs guarantees the needed generality of the underlying data structures because the states which are modified by executing ASM rules are so-called *Tarski structures*, i.e. sets of arbitrary elements on which arbitrary updatable functions (operations) and predicates (properties and relations) are defined. In the case of business process objects the elements are placeholders for values of arbitrary types and the operations typically the creation, duplication, deletion, modification of objects. Views are projections (substructures) of Tarski structures

Using such rules we define a succinct high-level and easily extendable S-BPM behavior model the business process practitioner can understand directly, without further training, and use a) to reason about the design and b) to hand it over to a software engineer as a binding and clear specification for a reliable and justifiably correct implementation.

For the sake of quick understandability and to avoid having to require from the reader some formal method expertise we paraphrase the ASM rules by natural language explanations, adopting Knuth's literate programming [3] idea for the development of abstract behavior models. The reader who is interested in the details of the simple foundation of the semantics of ASM rule systems, which can also be viewed as a rigorous form of pseudo-code, is refered to the Asm-Book [2]. Here it should suffice to draw the reader's attention to the fact that for a given ASM with rules R_i $(1 \leq i \leq n)$ in each state all rules R_i whose guard is true in this state are executed simultaneously, in one step. This parallelism

[2] Since ASM models support an intuitive operational understanding at both high and lower levels of abstraction, the software developer can use them to introduce in a rigorously documentable and checkable way the crucial design decisions when implementing the abstract ASM models. Technically this can be achieved using the ASM refinement concept see [2, 3.2.1].

allows one to hide semantically irrelevant details of sequential implementations of independent actions.

The ASM interpeter model for the semantics of S-BPM we describe in the following sections is developed by stepwise refinement, following the gradually proceeding exposition in this book. Thus we start with an abstract interaction view model of subject behavior diagrams (Sect. 2, based upon Sect.2.2.3 in this book, which (based upon Sect.5.4.3 in this book) is refined in Sect. 3 by detailed descriptions of the communication actions (send, receive) in their various forms (canceling or blocking, synchronous or asynchronous and including their multi-process forms, based upon Sect.5.6.1.3 in this book) and further refined by stepwise introduced structuring concepts: structured actions—alternative actions (Sect. 4, based upon Sect.5.6.2.5 in this book)—and structured processes: macros (Sect. 5.1, based upon Sect.5.6.2.2-4 in this book), interaction view normalization (Sect. 5.2, based upon Sect.5.4.4.2 in this book), process networks and observer view normalization (Sect. 5.3, based upon Sect.5.6.1.1-2 in this book). Two concepts for model extension are defined in Sect. 6. They cover in particular the exception handling model proposed in Sect.5.6.2.6 in this book.

We try to keep this appendix on an S-BPM interpreter technically self-contained though all relevant definitions are supported by the explanations in the preceding chapters of the book.

2 Interaction View of Subject Behavior Diagrams

An S-BPM *process* (shortly called process) is defined by a set of subjects each equipped with a diagram, called the *subject behavior diagram* (SBD) and describing the behavior of its subject in the process. Such a process is of distributed nature and describes the overall behavior of its subjects which interact with each other by sending or receiving messages (so-called send/receive actions) and perform certain activities on their own (so-called internal actions or functions).

2.1 Signature of Core Subject Behavior Diagrams

Mathematically speaking a subject behavior diagram is a directed graph. Each node represents a state in which the underlying subject[3] can be in when executing an activity associated to the node in the diagram. We call these states *SID_state*s (Subject Interaction Diagram states) of the subject in the diagram because they represent the state a subject is in from the point of view of the other subjects it is interacting with in the underlying process, where it only matters whether the subject is communicating (sending or receiving a message) or busy with performing an internal function (whose details are usually not interesting for and hidden to the other subjects). The incoming and the outgoing edges represent (and are labeled by names of) the subject's SID-state transitions from *source*(*edge*) to *target*(*edge*). The *target*(*outEdge*) of an

[3] Where needed we call an SBD a *subject*-SBD and write also $\text{SBD}_{subject}$ to indicate that it is an SBD with this underlying *subject*.

outEdge $\in$ *OutEdge*(*node*) is also called a successor state of *node* (element of the set *Sucessor*(*node*)), the *source*(*inEdge*) of an *inEdge* $\in$ *InEdge*(*node*) a predecessor state (in the diagram an element of the set *Predecessor*(*node*)).

As distinguished from SID-states (and usually including them) the overall states of a subject are called *data state*s or simply *state*s. They are constituted by a set of interpreted (possibly abstract) data types, i.e. sets with functions and predicates defined over them, technically speaking Tarski structures, the states of Abstract State Machines. SID-states of a subject are implicitly parameterized by the diagram in which the states occur since a subject may have different diagrams belonging to different processes; if we want to make the parameter D explicit we write $SID\_state_D(subject)$ or $SID\_state(subject, D)$.

The SID-states of a subject in a diagram can be of three types, corresponding to three fundamental types of activity associated to a node to be performed there under the control of the subject: *function states* (also called internal function or action node states), *send states* and *receive states*. The activity (operation or method) associated to and performed under the control of the subject at a *node* (read: when the subject is in the corresponding SID-state) is called *service*(*node*). We explain in Sect. 3 the detailed behavioral meaning of these services for sending resp. receiving a message (interaction via communication) and for arbitrary internal activities (e.g. activities of a human or functions in the sense of programming). In a given function state a subject may go through many so-called internal (Finite State Machine like) control states to each of which a complex data structure may be associated, depending on the nature of the performed function. These *internal states* are hidden in the SID-level view of subject behavior in a process, also called *normalized behavior* view and described in Sect. 5.2. The semantics of the interaction view of SBDs is defined in this section by describing the meaning of the transitions between SID-states in terms of communication and abstract internal functions.

A transition from a source to a target SID-state is allowed to be taken by the subject only when the execution of the service associated to the source node has been *Completed* under the control of this subject. This completion requirement is called synchrony condition and reflects the sequential nature of the behavior of a single subject, which in the given subject behavior diagram performs a sequence of single steps. Correspondingly each arc exiting a node corresponds to a termination condition of the associated service, also called *ExitCond*ition of the transition represented by the arc and usually labeling the arc; in the wording used for labeling arcs often the *ExitCond*ition refers only to a special data state condition reached upon service completion, but it is assumed to always contain the completion requirement implicitly. In case more than one edge goes out of a node we often write $ExitCond_i$ for the *ExitCond*ition of the i-th outgoing arc.

The nodes (states) are graphically represented by rectangles and by a systematic notational abuse sometimes identified with (uniquely named) occurrences of their associated *service* whose names are written into the rectangle. It is implicit in the graphical representation that given a SID-state (i.e. a node in the graph),

the associated service and the incoming and outgoing edges are functions of the SID-state.

Each SBD is assumed to be finite and to have exactly one *initial state* and at least one (maybe more than one) *end state*. It is assumed that each path leads to at least one end state. It is permitted that end states have outgoing edges, which the executing subject may use to proceed from this to a successor state, but each such path is assumed to lead back to at least one end state. A *process* is considered to *terminate* if each of its subjects is in one of its end states.

2.2 Semantics of Core Subject Behavior Diagram Transitions

The semantics of subject behavior diagrams D can be characterized essentially by a set of instances of a single SID-transition scheme BEHAVIOR(*subj*, *state*) defined below for the transition depicted in Fig. 1. It expresses that when a *subj*ect in a given SID-*state* in D has *Completed* a given action (function, send or receive operation)—read: PERFORMing the action has been *Completed* while the *subj*ect was in the given SID-*state*, assuming that the action has been STARTed by the *subj*ect upon entering this *state*—then the *subj*ect PROCEEDs to START its next action in its successor SID-state, which is determined by an *ExitCond*ition whose value is defined by the just completed action. This simple and natural transition scheme is instantiated for the three kinds of SID-states with their corresponding action types, namely by giving the details of the meaning of STARTing an action and PERFORMing it until it is *Completed* for internal functions and for sending resp. receiving messages (see Sect. 3).

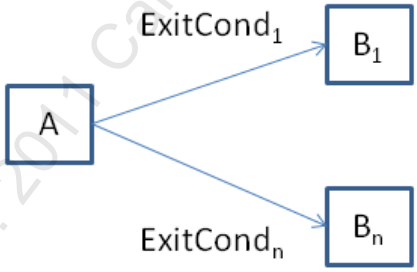


Fig. 1. SID-transition graph structure

Technically speaking the SID-transition scheme is an Abstract State Machine rule BEHAVIOR(*subj*, *state*) describing the transition of a *subj*ect from an SID-*state* with associated service A to a next SID-state with its associated service after (and only after) PERFORMing A has been *Completed* under the control of the subject. The successor state with its associated service to be STARTed next—in Fig. 1 one among B_i associated to the $target(outEdge(state, i))$ of the i-th

outEdge$(state, i)$ outgoing *state* for $1 \leq i \leq n$—is the target of an outgoing edge *outEdge* that satisfies its associated exit condition *ExitCond*(*outEdge*) when the *subj*ect has *Completed* to PERFORM its action A in the given *SID_state*. The outgoing edge to be taken is selected by a function $select_{Edge}$ which may be defined by the designer or at runtime by the user. In BEHAVIOR$(subj, state)$ the **else**-branch expresses that it may take an arbitrary a priori unknown number of steps until PERFORMing A is *Completed* by the *subj*ect.

BEHAVIOR$(subj, state)$ =
  if *SID_state*$(subj)$ = *state* **then**
    if *Completed*$(subj, service(state), state)$ **then**
      let *edge* =
        $select_{Edge}(\{e \in OutEdge(state) \mid ExitCond(e)(subj, state)\})$
          PROCEED$(subj, service(target(edge)), target(edge))$
    else PERFORM$(subj, service(state), state)$
  where
    PROCEED$(subj, X, node)$ =
      SID_state$(subj)$:= *node*
      START$(subj, X, node)$

Remark. Each SID-transition is implicitly parameterized via the SID-states by the diagram to which the transition parameters belong, given that a (concrete) subject may be simultaneously in SID-states of subject behavior diagrams of multiple processes.

We define the BEHAVIOR$_{subject}(D)$ of a *subject* behavior diagram D as the set of all ASM transition rules BEHAVIOR$(subject, node)$ for each $node \in Node(D)$.

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

When *subj*ect is known we write BEHAVIOR(D) instead of BEHAVIOR$_{subj}(D)$. BEHAVIOR(D) represents an interpreter of D.

This definition yields the traditional concept of (terminating) standard computations (also called *standard run*s) of a *subject* behavior diagram (from the point of view of subject interaction), namely sequences $S_0, \ldots, S_n$ of states of the subject behavior diagram where in the initial resp. final state S_0, S_n the *subject* is in the initial resp. a final SID-state and where for each intermediate S_i (with $i < n$) with SID-state say $state_i$ its successor state S_{i+1} is obtained by applying BEHAVIOR$(subject, state_i)$. Usually we only say "computation" or "run" omitting the "standard" attribute.

Remark. One can also spell out the SBD-BEHAVIOR rules as a general SBD-interpreter $Interpreter_{SBD}$ which given as input any SBD D of any *subject* walks through this diagram from the initial state to an end state, interpreting each diagram *node* as defined by BEHAVIOR$(subject, node)$.

Remark. BEHAVIOR$(subj, state)$ is a scheme which uses as basic constituents the abstract submachines PERFORM, START and the abstract completion predicate *Completed* to describe the pure interaction view for the three kinds of action in a subject behavior diagram: that an action is STARTed and PERFORMed by

a subject until it is *Completed* hiding the details of how START, PERFORM and *Completed* are defined. These constituents can be specialized further by defining a more detailed meaning for them to capture the semantics of specific internal functions and of particular send and receive patterns. Technically speaking such specializations represent ASM-refinements (as defined in [1]). We use examples of such ASM-refinements to specify the precise meaning of the basic S-BPM communication constructs (see Sect. 3) and of the additional S-BPM behavior constructs (see Sect. 4). The background concepts for communication actions are described in Sect. 3.1, Sect. 3.3-3.4 present refinements defining the details of send and receive actions.

3 Refinements for the Semantics of Core Actions

Actions in a core subject behavior diagram are either internal functions or communication acts. Internal functions can be arbitrary manual functions performed by a human subject or functions performed by machines (e.g. represented abstractly or by finite state machine diagrams or by executable code written in some programming language) and are discussed in Sect. 3.5.

3.1 How to Perform Alternative Communication Actions

For each communication node we refine in this section and Sect. 3.2-3.4 the abstract machines START, PERFORM and the abstract predicate *Completed* to the corresponding concepts of STARTing and PERFORMing the communication and the meaning of its being *Completed*. Since the alternative communication version naturally subsumes the corresponding 1-message version (i.e. without alternatives where exactly one message is present to be sent or received), we give the definitions for the general case with communication action alternatives and derive from it the special 1-message case as the one where the number of alternatives is 1. The symmetries shared by the two *ComAct*ion versions *Send* and *Receive* are made explicit by parameterizing machine components of the same structure with an index *ComAct*.

In this section three concepts are described which are common to and support the detailed definition of both communication actions send and receive in Sect. 3.2-3.4: subject interaction diagrams describing the process communication structure, input pool of subjects and the iterative structure of alternative send/receive actions.

Subject Interaction Diagram The communication structure (signature) of a process is defined by a *Subject Interaction Diagram* (SID-diagram). These diagrams are directed graphs consisting of one node for each subject in the process (so that without loss of generality nodes of an SID-diagram can be identified with subjects) and one directed arc from node $subject_1$ to node $subject_2$ for each type of message which may be sent in the process from $subject_1$ to $subject_2$ (and thereby received by $subject_2$ from $subject_1$). Thus SID-edges define

the communication connections between their source and target subjects and are labeled with the message type they represent. There may be multiple edges from $subject_1$ to $subject_2$, one for each type of possibly exchanged message.

Input Pools To support the asynchronous understanding of communication, which is typical for distributed computations, each subject is assumed to be equipped with an *inputPool* where messages sent to this subject (called *receiver*) are placed by any other subject (called *sender*) and where the receiver looks for a message when it 'expects' it (i.e. is ready to receive it).

An *inputPool* can be configured by the following size restricions:

- restricting the overall capacity of *inputPool*, i.e. the maximal number of messages of any type and from any sender which are allowed to be *Present* at any moment in *inputPool*,
- restricting the maximal number of messages coming from an indicated *sender* which are allowed to be *Present* at any moment in the *inputPool*,
- restricting the maximal number of messages of an indicated *type* which are allowed to be *Present* at any moment in *inputPool*,
- restricting the maximal number of messages of an indicated *type* and coming from an indicated *sender* which are allowed to be *Present* at any moment in the *inputPool*.

For a uniform description of synchronous communication 0 is admitted as value for input pool size parameters. It is interpreted as imposing that the *receiver* accepts messages from the indicated sender and/or of the indicated type only via a rendezvous with the *sender*.

Asynchronous communication is characterized by positive natural numbers for the input pool size parameters. In the presence of such size limits it may happen that a sender tries to place a message of some type into an input pool which has reached the corresponding size limit (i.e. its total capacity or its capacity for messages of this type and/or from that sender). The following two strategies are foreseen to handle this situation:

- *canceling send* where either a) a forced message deletion reduces the actual size of the input pool and frees a slot to insert the arriving message or b) the incoming message is dropped (i.e. not inserted into the input pool),
- *blocking send* where the sending is blocked and the sender repeats the attempt to send its message until either a) the input pool becomes free for the message to be inserted or b) a timeout has been reached triggering an interrupt of this send action or c) the sender manually abrupts its send action.

Three canceling disciplines are considered, namely to drop the incoming message or to delete the oldest resp. the youngest message m in P, determined in terms of the $insertionTime(m, P)$ of m into P.[4]

[4] We use Hilbert's ι-operator to express by $\iota\ x\ P(x)$ the unique element satisfying property P.

$youngestMsg(P) =$
  $\iota m(m \in P$ **and forall** $m' \in P$ **if** $m' \neq m$ **then**
    $insertionTime(m, P) > insertionTime(m', P))$ //m came later
$oldestMsg(P) =$
  $\iota m(m \in P$ **and forall** $m' \in P$ **if** $m' \neq m$ **then**
    $insertionTime(m, P) < insertionTime(m', P))$ //m came earlier

Whether a send action is handled by the targeted input pool P as canceling or blocking depends on whether in the given state the pool satisfies the size parameter constraints which are formulated in a pool *constraintTable*. Each row of $constraintTable(P)$ indicates for a combination of *sender* and *msgType* the allowed maximal *size* together with an *action* to be taken in case of a constraint violation:

$constraintTable(inputPool) =$
  ...
  $sender_i\ msgType_i\ size_i\ action_i\ (1 \leq i \leq n)$
  ...
where
  $action_i \in \{Blocking, DropYoungest, DropOldest, DropIncoming\}$
  $size_i \in \{0, 1, 2, \ldots, \infty\}$
  $sender_i \in Subject$
  $msgType_i \in MsgType$

When a sender tries to send a message *msg* to the owner of an input pool P the first $row = \ s\ t\ n\ a$ in the $constraintTable(P)$ is identified whose size constraint concerns *msg* and would be violated by inserting *msg*:

$ConstraintViolation(msg, row)$ iff [5]
  $Match(msg, row) \wedge size(\{m \in P \mid Match(m, row)\}) + 1 \not< n$
where
  $Match(m, row)$ iff
    $(sender(m) = s$ **or** $s = any)$ **and** $(type(m) = t$ **or** $t = any)$

If there is no such row—so that the first such element in $constraintTable(P)$ is **undef**—the message can be inserted into the pool; otherwise the action indicated in the identified row is taken, thus either blocking the sender or accepting the message (by either dropping it or inserting it into the pool at the price of deleting another pool element).

It is required that in each row r with $size = 0$ the *action* is *Blocking* and that in case $maxSize(P) < \infty$ the *constraintTable* has the following last (the default) row:

any any maxSize Blocking

[5] iff stands for: if and only if.

Similarly a (possibly blocking) receive action tries to receive a message, 'expected' to be of a given kind (i.e. of a given type and/or from a given sender) and chosen out of finitely many alternatives (again either nondeterministically or respecting a given priority scheme), with possible timeout to abort unsuccessful receives (i.e. when no message of the expected kind is in the input pool) or a manual abort chosen by the subject.

Since in a distributed computation more than one subject may simultaneously try to place a message to the input pool P of a same receiver, a selection mechanism is needed (which in general will depend on P and therefore is denoted $select_P$) to determine among those subjects that are *TryingToAccess* P the one which *CanAccess* it to place the message to be sent.[6]

$CanAccess(sender, P)$ if and only if
$sender = select_P(\{subject \mid TryingToAccess(subject, P)\})$

Alternative Send/Receive Iteration Structure S-BPM forsees so-called *alternative* send/receive states where to perform a communication action *ComAct* (*Send* or *Receive*) the subject can do three things in order:

- choose an *alt*ernative among finitely many *Alternative*s,[7] i.e. message kinds associated to the send/receive state,
- prepare a corresponding *msgToBeHandled*: for a send action a *msgToBeSent* and for a receive action an *expectedMsg* kind,
- TryAlternative$_{ComAct}$, i.e. try to actually send the *msgToBeSent* resp. receive a message *Match*ing the kind of *expectedMsg*.

The choice and preparation of an alternative is defined below by a component Choose&PrepareAlternative$_{ComAct}$ of TryAlternative$_{ComAct}$.

[6] One can formally define the *TryingToAccess* predicate, but the $select_P$ function is deliberately kept abstract. There are various criteria one could use for its further specification and various mechanisms for its implementation. A widely used interpretation of such functions in a distributed environment is that of a nondeterministic choice, which can be implemented using some locking mechanism to guarantee that at each moment at most one subject can insert a message into the input pool in question. The negative side of this interpretation is that proofs of properties of systems exhibiting nondeterministic phenomena are known to be difficult. Attempts to further specify the selection (e.g. by considering a maximal waiting time) introduce a form of global control for computing the selection function that contradicts the desired decentralized nature of an asynchronous communication mechanism (and still does not solve the problem of simultaneity in case different senders have the same waiting time). One can avoid infinite waiting of a subject (for a moment where it *CanAccess* a pool) by governing the waiting through a timeout mechanism.

[7] We consider *Alternative* as dependent on two parameters, *subj*ect and *state*, to prepare the ground for service processes where the choice of *Alternative*s in a *state* may depend on the subject type the client belongs to. Otherwise *Alternative* depends only on the *state*. In the currently implemented diagram notation the *Alternative*s appear as pairs of a receiver and a message type, each labeling in the form (*to receiver*, *msgType*) an arc leaving the alternative send *state* in question.

If the selected *alt*ernative fails (read: could not be be communicated neither asynchronously nor in a synchronous manner between sender and receiver), the subject chooses the next *alt*ernative until:

- either one of them succeeds, implying that the send/receive action in the given state can be *Completed* normally,
- or all *Alternative*s have been tried out but the *TryRoundFinished* unsuccessfully.

After such a first (so-called *nonblocking* because non interruptable) TryRound a second one can be started, this time of *blocking* character in the sense that it may be interrupted by a *Timeout* or *UserAbruption*.

This implies iterations through a runtime set *RoundAlternative* of alternatives remaining to be tried out in both the first (*nonblocking*) and the other (*blocking*) TryRounds in which the subject for its present *ComAct* action has to TryAlternative$_{ComAct}$. *RoundAlternative* is initialized for the first round in Start, namely to the set *Alternative*(*subj*, *node*) of all alternatives of the *subj*ect at the *node*, and reinitialized at the beginning of each blocking round.

Since the blocking TryRound can be interrupted by a *Timeout*-triggered Interrupt or by a ('manually') *UserAbruption*-triggered Abruption, there are three outgoing edges to Proceed from a communication *node*. We use three predicates *NormalExitCond*, *TimeoutExitCond*, *AbruptionExitCond* to determine the correct *node* exit when the ComAct completes normally or due to the *Timeout* condition[8] or due to a *UserAbruption*. One of these three cases will eventually occur so that the corresponding exit condition then determines the next SID-state where the subject has to Proceed with its run. To guarantee a correct behavior these three exit conditions and the completion predicate are initialized in Start to false. Since the machines are the same for the two *ComAct*ion cases (*Send* or *Receive*) we parameterize them in the definition below by an index *ComAct*.

Since the actual blocking presents itself only if none of the possible alternatives succeeds in a first run, *blockingStartTime*(*subject*, *node*)—the timeout clock which depends on the subject and the state *node*, not on the messages—is set only after a first round of unsuccessful sending attempts, namely in the submachine InitializeBlockingTryRounds. As a consequence the *Timeout* condition guards TryAlternative$_{ComAct}$ only in the blocking rounds. Timeouts are considered as of higher priority than user abruptions.

This explains the following refinement of the abstract machine Perform to Perform(*subj*, ComAct, *state*). The flowchart in Fig. 2 visualizes the structure of Perform(*subj*, ComAct, *state*).[9] The symmetry between non-blocking and

[8] *TimeoutExitCond* is only a name for the timeout condition we define below, namely *Timeout*(*msg*, *timeout*(*state*)); in the diagram it is written as edge label of the form *Timeout* : *timeout*.

[9] These flowcharts represent so-called control-statel ASMs which come with a precise semantics, see [2, p.44]. Using the flowchart representation of control-state ASMs allow one to save some control-state guards and updates. To make this exposition

blocking TryRounds is illustrated by a similar coloring of the respective components, whereas the components for the timeout and user abruption extensions are colored differently. Outgoing edges without target node denote possible exits from the flowchart. The equivalent textual definition (where we define also the components) reads as follows.

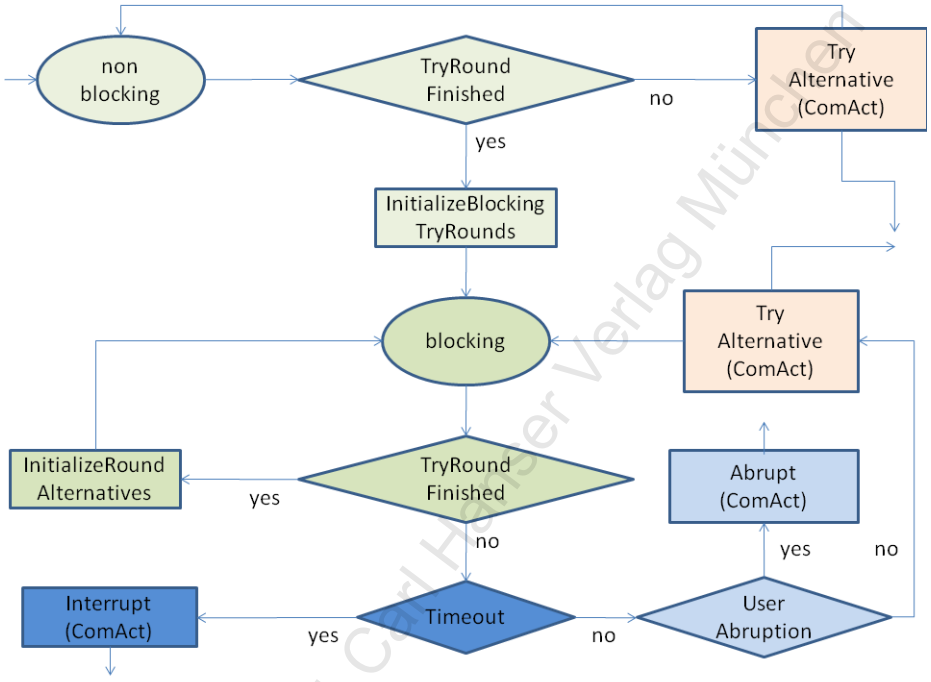


Fig. 2. PERFORM(*subj*, COMACT, *state*)

PERFORM(*subj*, COMACT, *state*) =
  if *NonBlockingTryRound*(*subj*, *state*) **then**
    if *TryRoundFinished*(*subj*, *state*) **then**
      INITIALIZEBLOCKINGTRYROUNDS(*subj*, *state*)
    else TRYALTERNATIVE$_{ComAct}$(*subj*, *state*)
  if *BlockingTryRound*(*subj*, *state*) **then**
    if *TryRoundFinished*(*subj*, *state*)
      then INITIALIZEROUNDALTERNATIVES(*subj*, *state*)
      else
        if *Timeout*(*subj*, *state*, *timeout*(*state*)) **then**

self-contained we provide however the full textual definition and as a consequence allow us to suppress in the flowchart some of the parameters.

INTERRUPT$_{ComAct}(subj, state)$
elseif *UserAbruption*$(subj, state)$
then ABRUPT$_{ComAct}(subj, state)$
else TRYALTERNATIVE$_{ComAct}(subj, state)$

Macros and Components of PERFORM**(***subj***,** COMACT**,** *state***)** We define here also the START$(subj,$ COMACT$,$ *state*$)$ machine. The function *now* used in SETTIMEOUTCLOCK is a monitored function denoting the current system time.

START$(subj,$ COMACT$,$ *state*$)$ =
INITIALIZEROUNDALTERNATIVES$(subj, state)$
INITIALIZEEXIT&COMPLETIONPREDICATES$_{ComAct}(subj, state)$
ENTERNONBLOCKINGTRYROUND$(subj, state)$
where
INITIALIZEROUNDALTERNATIVES$(subj, state)$ =
RoundAlternative$(subj, state)$:= *Alternative*$(subj, state)$
INITIALIZEEXIT&COMPLETIONPREDICATES$_{ComAct}(subj, state)$ =
INITIALIZEEXITPREDICATES$_{ComAct}(subj, state)$
INITIALIZECOMPLETIONPREDICATE$_{ComAct}(subj, state)$
INITIALIZEEXITPREDICATES$_{ComAct}(subj, state)$ =
NormalExitCond$(subj,$ COMACT$,$ *state*$)$:= *false*
TimeoutExitCond$(subj,$ COMACT$,$ *state*$)$:= *false*
AbruptionExitCond$(subj,$ COMACT$,$ *state*$)$:= *false*
INITIALIZECOMPLETIONPREDICATE$_{ComAct}(subj, state)$ =
Completed$(subj,$ COMACT$,$ *state*$)$:= *false*

$[Non]BlockingTryRound(subj, state)$ =
tryMode$(subj, state) = [non]blocking$
ENTER[NON]BLOCKINGTRYROUND$(subj, state)$ =
tryMode$(subj, state)$:= $[non]blocking$
TryRoundFinished$(subj, state)$ =
RoundAlternatives$(subj, state) = \emptyset$
INITIALIZEBLOCKINGTRYROUNDS$(subj, state)$ =
ENTERBLOCKINGTRYROUND$(subj, state)$
INITIALIZEROUNDALTERNATIVES$(subj, state)$
SETTIMEOUTCLOCK$(subj, state)$
SETTIMEOUTCLOCK$(subj, state)$ =
blockingStartTime$(subj, state)$:= *now*
Timeout$(subj, state, time)$ =
$now \geq blockingStartTime(subj, state) + time$

INTERRUPT$_{ComAct}(subj, state)$ =
SETCOMPLETIONPREDICATE$_{ComAct}(subj, state)$
SETTIMEOUTEXIT$_{ComAct}(subj, state)$

SetCompletionPredicate$_{ComAct}$(*subj*, *state*) =
  Completed(*subj*, ComAct, *state*) := *true*
SetTimeoutExit$_{ComAct}$(*subj*, *state*) =
  TimeoutExitCond(*subj*, ComAct, *state*) := *true*
Abrupt$_{ComAct}$(*subj*, *state*) =
  SetCompletionPredicate$_{ComAct}$(*subj*, *state*)
  SetAbruptionExit$_{ComAct}$(*subj*, *state*)

To conclude this section: an attempt to TryAlternative$_{ComAct}$ comes in two phases: the first phase serves to Choose&PrepareAlternative and is followed by a second phase where the subject as we are going to explain in the next section will try to actually carry out the communication. If this attempt succeeds, the *ComAct* is *Completed*; otherwise the subject will try out the next send/receive alternative.

3.2 How to Try a Specific Communication Action

As explained in Sect. 3.1 subject's first step to TryAlternative$_{ComAct}$ in [*non*]*blocking tryMode* is to Choose&PrepareAlternative$_{ComAct}$. Then it will Try$_{ComAct}$ for the prepared message(s).[10]

TryAlternative$_{ComAct}$(*subj*, *state*) =
  Choose&PrepareAlternative$_{ComAct}$(*subj*, *state*)
    seq Try$_{ComAct}$(*subj*, *state*)

We first explain the Choose&PrepareAlternative$_{ComAct}$ component for the elaboration of messages and then define the machines Try$_{ComAct}$.

Elaboration of Messages Messages are objects which need to be prepared. The PrepareMsg component of Choose&PrepareAlternative does this for each selected communication *alt*ernative. To describe the selection, which can be done either nondeterministically or following a priority scheme, we use abstract functions *select*$_{Alt}$ and *priority*. They can and will be further specified once concrete send *state*s are given in a concrete diagram.

Choose&PrepareAlternative also must ManageAlternativeRound, essentially meaning to MarkSelection—typically by deleting the selected alternative from *RoundAlternative*, to exclude the chosen candidate from a possible

[10] Such a sequential structure is usually described using an FSM-like control state, say *tryMode*, as we will do in the flowcharts below. For a succinct textual description we will use sometimes the ASM **seq** operator (see the definition in [2]) which allows one to hide control state guards and updates. For example in the definition of Choose&PrepareAlternative we could skip an EnterTryAlternative$_{ComAct}$ update because the machine is used only as composed by **seq** (with Try$_{ComAct}$ in TryAlternative$_{ComAct}$).

next AlternativeRound step which may happen if sending/receiving the selected message is blocked.

There is one more feature to be prepared for due to the fact that S-BPM deals also with multi-processes in the form of multiple send/receive actions, which extend single send/receive actions where only one message is sent resp. received to complete the communication act instead of *mult* many messages belonging to the chosen *alt*ernative.

In the S-BPM framework a multi-process is either a multiple send action (where a subject iterates finitely many times sending a message of some given kind) or a multiple receive action (where a subject expects to receive finitely many messages of a given kind). In the diagram notation the (design-time determined) *mult*itude in question, which adds a new kind of message to communicate, appears as number of messages of some kind to be sent or to be received during a Multi Send or MultiReceive. It is assumed that $mult \geq 2$. The principle of multiple send and receive actions in the presence of communication alternatives which is adopted for S-BPM is that once in a state a subject has chosen a MultiSend or MultiReceive alternative, to complete this multi-action it must send resp. receive the indicated multitude of messages of the kind defined for the chosen alternative and in between will not pursue any other communication. Therefore the alternative send/receive TryRound structure (see Fig. 2) and its START component are not affected by the multi-process feature, but only the TRY_{ComAct} component which has to provide a nested MultiRound. For MultiSend actions it is also required that first all specimens of a *msgToBeHandled* are elaborated by the subject, as to-be-contemplated for the definition of $\text{CHOOSE\&PREPAREALTERNATIVE}_{Send}$, and then they are tried to be sent one after the other.

Thus one needs a MultiRound to guarantee that if a multi-communication action has been chosen as communication *alt*ernative, then:

- each of the $mult(alt)$ many specimens belonging to the chosen message *alt*ernative is tried out exactly once,
- if for at least one of these specimens the attempt to communicate fails the chosen *alt*ernative is considered to be failed,
- no other communication takes place within a MultiRound.

Thus each MultiRound constitutes one iteration step of the current AlternativeRound where the multi-communication action has been selected as *alt*ernative. Since single send/receive steps are the special case of multi steps where $mult(alt) = 1$ we treat single/multi communication actions uniformly instead of introducing them separately.[11]

[11] The price to pay is a small MultiRound overhead (which can later be optimized away for the single action case $mult(alt) = 1$). In an alternative model one could introduce first single communication actions (as they are present in the current implementation) and then extend them in a purely incremental way by the multi-process feature. Both ways to specify S-BPM clearly show that the extension of S-BPM from SingleActions to MultiActions (for both Send and Receive actions) is a *purely incremental* (in logic also called conservative) *extension*, which does only

In the presence of multi-communication actions for each alternative one has to InitializeMultiRound, as done in the ManageAlternativeRound component of Choose&PrepareAlternative defined below.

This explains the following *ComAct*ion preparation machine a *subj*ect will execute in every communication *state* as first step of TryAlternative$_{ComAct}$. As before the *ComAct* parameter stands for *Send* or *Receive*.

Choose&PrepareAlternative$_{ComAct}(subj, state) =$
  let $alt = select_{Alt}(RoundAlternative(subj, state), priority(state))$
    PrepareMsg$_{ComAct}(subj, state, alt)$
    ManageAlternativeRound$(alt, subj, state)$
    where
      ManageAlternativeRound$(alt, subj, state) =$
        MarkSelection$(subj, state, alt)$
        InitializeMultiRound$_{ComAct}(subj, state)$
      MarkSelection$(subj, state, alt) =$
        Delete$(alt, RoundAlternative(subj, state))$

A subject to PrepareMsg$_{Send}$ will *composeMsg*s out of *msgData* (the values of the relevant data structure parameters) and make the result available in *MsgToBeHandled*.[12] Similarly a receiver to PrepareMsg$_{Receive}$ may select $mult(alt)$ elements from a set of $ExpectedMsgKind(alt)$ using some choice function $select_{MsgKind}$.[13]

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

The functions *composeMsg* and *msgData* must be left abstract in this high-level model, playing the role of interfaces to the underlying data structure manipulations, because they can be further refined only once the concrete data struc-

add new behavior without retracting behavior that was possible before. It supports a modular design discipline and compositional proofs of properties of the system. Notably all the other extensions defined in S-BPM are of this kind. See Sect. 6 for further explanations.

[12] For a *Send*(*Multi*) *alt*ernative $mult(alt)$ message specimens of the selected alternative will be composed, whereas for a *Send*(*Single*) action *MsgToBeHandled* will be a singleton set containing a unique element which we then denote *msgToBeSent*.

[13] In analogy to *msgToBeSent* we write also *msgKindToBeReceived* if there is a unique chosen kind of *MsgToBeHandled* by a receive action. This case is currently implemented.

tures are known which are used by the subject in the send state under consideration. It is however assumed that there are functions $sender(msg)$, $type(msg)$ and $receiver(msg)$ to extract the corresponding information from a message, so that *composeMsg* is required to put this information into a message. Similarly for the *expectedMsgKind* and $select_{MsgKind}$ functions.

TRY$_{ComAct}$ Components The structure of the machines TRY$_{ComAct}$ we are going to explain now is visualized by Fig. 3 and Fig. 4.

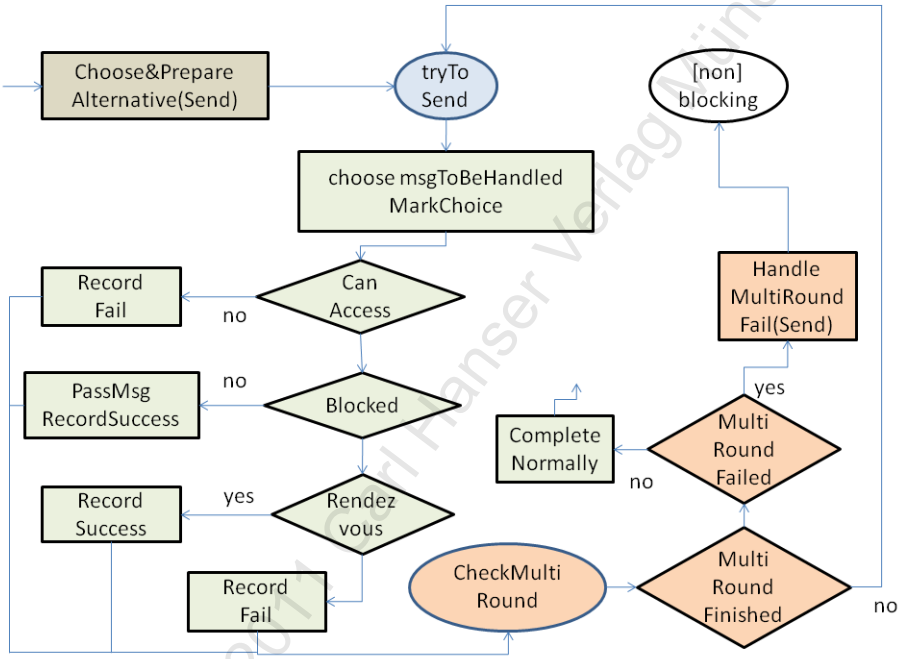


Fig. 3. TRYALTERNATIVE$_{Send}$

In TRY$_{ComAct}$ the subject first chooses from *MsgToBeHandled* a message m (to send) or kind m of message (to receive) and—to exclude it from further choices—will MARKCHOICE of m.[14] Then the subject does the following:

- For *Send* it checks whether it *CanAccess* the input pool of the $receiver(m)$ to TRY$_{Async(Send)}$ing m (otherwise it will CONTINUEMULTIROUND$_{Fail}$, which includes to RECORDFAILure of this send attempt).

[14] MARKCHOICE is the MultiRound pendant of MARKSELECTION defined in Sect. 3.1 for *AlternativeRound*s. We include into it a record of the current choice because this information is needed to describe the Rendezvous predicate for synchronous communication.

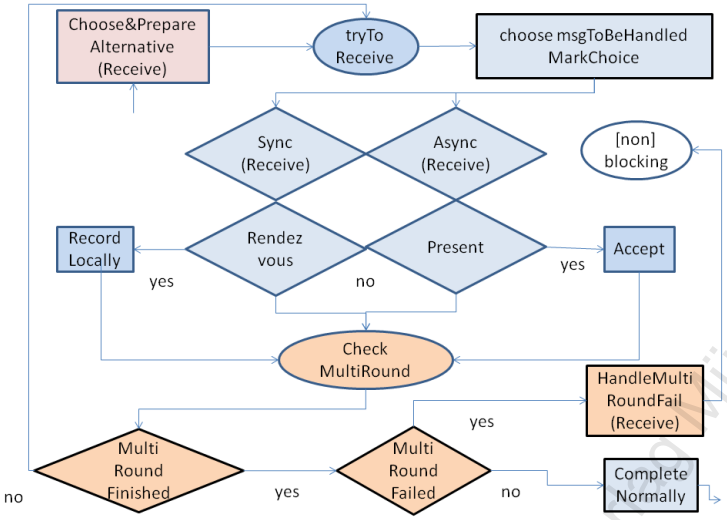


Fig. 4. TRYALTERNATIVE$_{Receive}$

- For *Receive* it goes directly to TRY$_{Async(Receive)}$ or TRY$_{Sync(Receive)}$ a message of kind m depending on whether the $commMode(m)$ is asynchronous (as expressed by the guard $Async(Receive)(m)$ or synchronous (as expressed by the guard $Sync(Receive)(m)$), without the *CanAccess* condition.[15]

Another slight asymmetry between send/receive actions derives from the fact that the sender tries a synchronous action only if the asynchronous one failed.

CONTINUEMULTIROUND$_{Fail}$ has a pendant CONTINUEMULTIROUND$_{Success}$ for successful communication. They record success resp. failure of the current MultiRound communication step and check whether to continue with the MultiRound or go back to the AlternativeRound.

TRY$_{ComAct}(subj, state)$ =
  choose $m \in MsgToBeHandled(subj, state)$
    MARKCHOICE$(m, subj, state)$
    if $ComAct = Send$ **then**
      let $receiver = receiver(m)$, $pool = inputPool(receiver)$
        if not $CanAccess(subj, pool)$ **then**
          CONTINUEMULTIROUND$_{Fail}(subj, state, m)$

[15] Thus the access of a *receiver* to its input *pool* (which comes up to read the pool and to possibly delete an expected message) can happen at the same time as an INSERT of a sender. One INSERT and one DELETE operation can be assumed to be executed consistently in parallel by the pool manager. An alternative would be to include the receiver into the *CanAccess* mechanism—at the price of complicating the definition of *RendezvousWithSender*.

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] $\text{ASYNCH}(ComAct)$ and $\text{SYNC}(ComAct)$ 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 $\text{ASYNCH}(Send)$ and $\text{SYNC}(Send)$ 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 $\text{ASYNCH}(Send)$ *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 $\text{COMPLETENORMALLY}_{Send}$.

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.

[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.

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* ∈ *constraintTable*(*rec*) | *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 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 PREPAREMSG$_{Receive}$ in Sect. 3.1).

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

[20] This MultiRound location is updated in MARKCHOICE.

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.

ASYNC(*Receive*)(*subj*, *state*, *msg*) =
  ACCEPT(*subj*, *msg*)
  $\text{CONTINUEMULTIROUND}_{Success}(subj, state, msg)$
where
  ACCEPT(*subj*, *m*) =
    let *receivedMsg* =
      $select_{ReceiveOfKind(m)}(\{msg \in inputPool(subj) \mid Match(msg, m)\})$
      RECORDLOCALLY(*subj*, *receivedMsg*)
      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* ∈ *pool* *Match*(*msg*, *m*)

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 $\text{CONTINUEMULTIROUND}_{Success}(subj, state, currMsgKind(subj, state))$.

SYNC(*Receive*)(*subj*, *state*, *msgKind*) =
  let *P* = *inputPool*(*subj*), *sender* = ι*s*(*CanAccess*(*s*, *P*))
    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.

$RendezvousWithSender(subj, msgKind)$ 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.

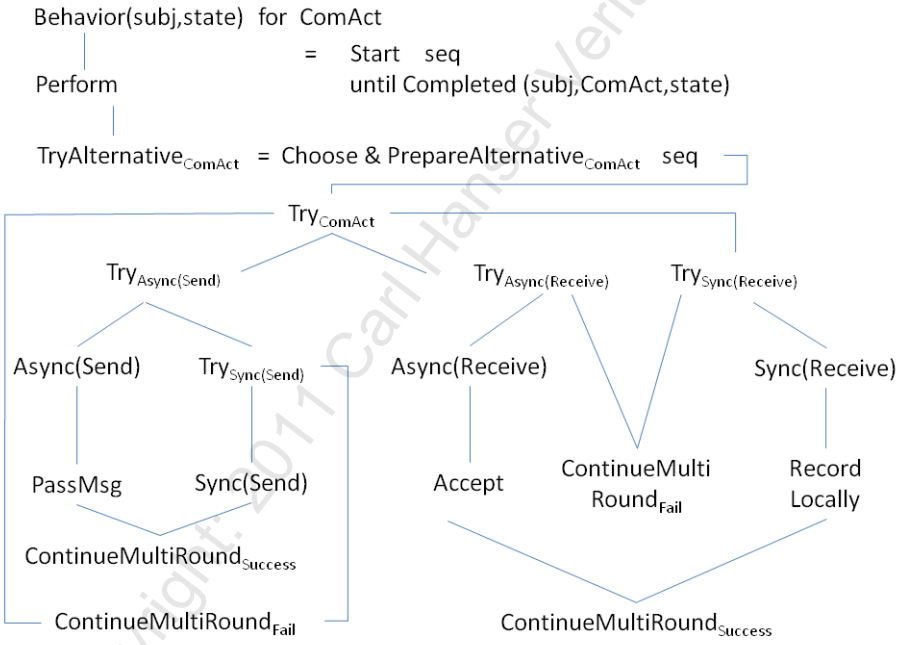


Fig. 5. Subcomponent Structure for Communication Behavior

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

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**
  INTERRUPT$_{service(state)}(subj, state)$
elseif $UserAbruption(subj, state)$
  then 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.

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.

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

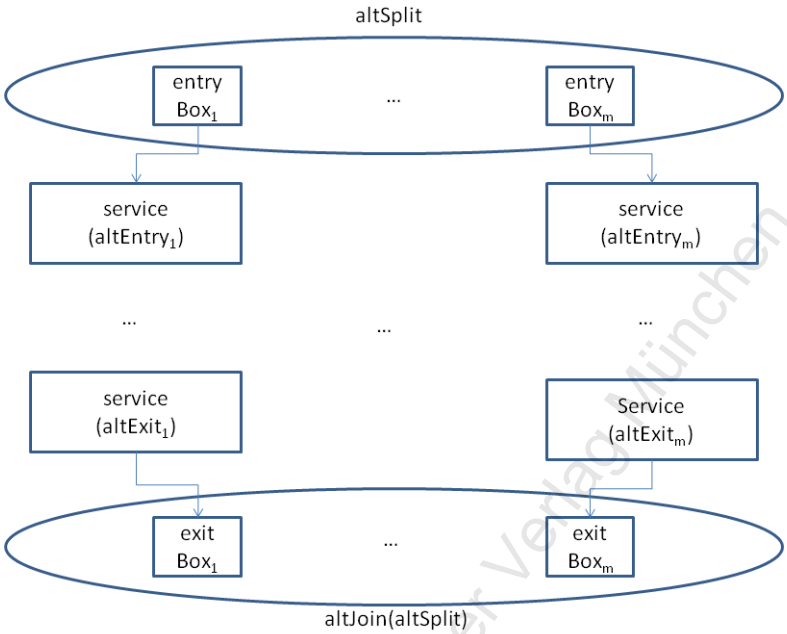


Fig. 6. Structure of Alternative Action Nodes

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.

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**

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*).

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

[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*.

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

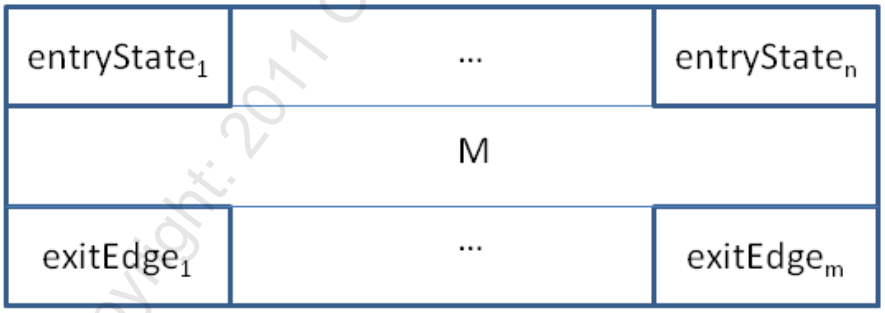


Fig. 7. Macro Table associated to a Macro State

The macro insertion constraints are therefore about how the *entryState*s and *exitEdge*s 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.

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 *entryState*s, 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 *exitEdge*s, 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 *exitEdge*s. 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.

[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 *macroState*s 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 run*s, 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.

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 $state_1, \ldots, state_{n+1}$ in D such that $state_1 = start$ and the following **MaximalFunctionSequence** property holds for the path $state_1, \ldots, state_{n+1}$:

- 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 $state_1, \ldots, state_n$ of a path in $Path_1$ (if there are any) is compressed into the *start* node[34] with associated service $(f_1 \circ \ldots \circ f_n)$ and put into $InteractionView(D)$ with one edge leading from *start* (which is then also denoted $state_{(1,\ldots,n)}$) 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 $state_0, \ldots, state_{n+1}$ 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 $state_1, \ldots, state_{n+1}$ 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.

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.

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 $ObserverView(observer, D_{subj})$.

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

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

[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.

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 $state' \in D_o$

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 $\textsc{Behavior}_{subj}(D)$. 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 $\textsc{Behavior}(subj, state)$ 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 $\textsc{Behavior}_{subj}(D)$ by new ways to $\textsc{Proceed}$ from $\textsc{Behavior}(subj, state)$ to the next state.

[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*.

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 $e \in InterruptKind$. $\textsc{Behavior}(D^*)$ is defined as in Sect. 2.2 from $\textsc{Behavior}_D(subj, state)$ with the following extension $\textsc{InterruptBehavior}_{D^*}$ of $\textsc{Behavior}_D(subj, s_i)$ for *InterruptState*s s_i of D, whereas $\textsc{Behavior}(subj, state)$ remains unchanged for the other D *state*s and for *state*s of D'—which are assumed to be disjoint from those of D:[41]

$\textsc{Behavior}_{D^*}(subj, state) =$ // Case of InterruptExtension(D, D')

$$\begin{cases} \textsc{Behavior}_D(subj, state) & \textbf{if } state \in D \setminus InterruptState \\ \textsc{Behavior}_{D'}(subj, state) & \textbf{if } state \in D' \\ \textsc{InterruptBehavior}(subj, state) & \textbf{if } state \in InterruptState \end{cases}$$

$\textsc{InterruptBehavior}(subj, s_i) =$ // 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 $j = indexOf(interruptKind(msg), InterruptKind(s_i))$
      $handleState = target(arc_{i,j})$
        $\textsc{Proceed}(subj, service(handleState), handleState)$
        $\textsc{Delete}(msg, inputPool(subj))$
  else $\textsc{Behavior}_D(subj, s_i)$
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.

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.

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.

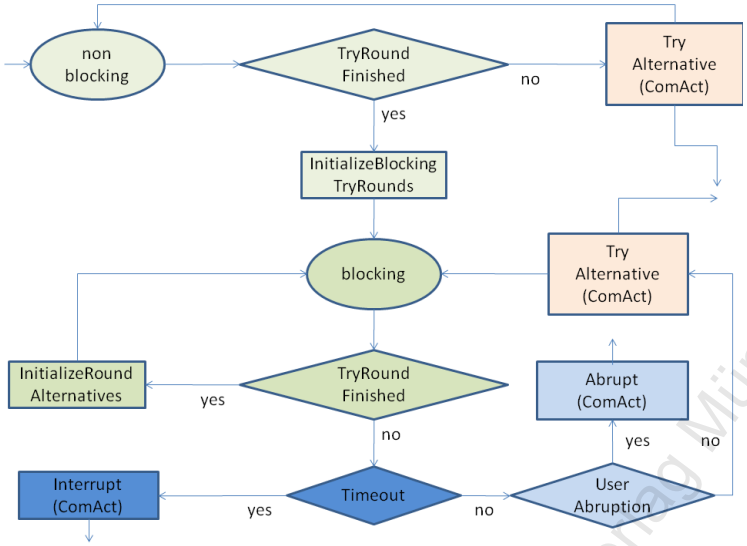
7.1 Subject Behavior Diagram Interpretation

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

$\textsc{Behavior}(subj, state) =$
  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 $\textsc{TryAlternative}_{ComAct}(subj, state)$



Interpretation of Auxiliary Macros

START(*subj*, COMACT, *state*) =
  INITIALIZEROUNDALTERNATIVES(*subj*, *state*)
  INITIALIZEEXIT&COMPLETIONPREDICATES$_{ComAct}$(*subj*, *state*)
  ENTERNONBLOCKINGTRYROUND(*subj*, *state*)
where
  INITIALIZEROUNDALTERNATIVES(*subj*, *state*) =
    RoundAlternative(*subj*, *state*) := *Alternative*(*subj*, *state*)
  INITIALIZEEXIT&COMPLETIONPREDICATES$_{ComAct}$(*subj*, *state*) =
    INITIALIZEEXITPREDICATES$_{ComAct}$(*subj*, *state*)
    INITIALIZECOMPLETIONPREDICATE$_{ComAct}$(*subj*, *state*)
  INITIALIZEEXITPREDICATES$_{ComAct}$(*subj*, *state*) =
    NormalExitCond(*subj*, COMACT, *state*) := *false*
    TimeoutExitCond(*subj*, COMACT, *state*) := *false*
    AbruptionExitCond(*subj*, COMACT, *state*) := *false*
  INITIALIZECOMPLETIONPREDICATE$_{ComAct}$(*subj*, *state*) =
    Completed(*subj*, COMACT, *state*) := *false*

[*Non*]*BlockingTryRound*(*subj*, *state*) =
  tryMode(*subj*, *state*) = [*non*]*blocking*
ENTER[NON]BLOCKINGTRYROUND(*subj*, *state*) =
  tryMode(*subj*, *state*) := [*non*]*blocking*
TryRoundFinished(*subj*, *state*) =
  RoundAlternatives(*subj*, *state*) = ∅
INITIALIZEBLOCKINGTRYROUNDS(*subj*, *state*) =
  ENTERBLOCKINGTRYROUND(*subj*, *state*)

InitializeRoundAlternatives($subj$, $state$)
SetTimeoutClock($subj$, $state$)
SetTimeoutClock($subj$, $state$) =
$blockingStartTime(subj, state) := now$
$Timeout(subj, state, time)$ =
$now \geq blockingStartTime(subj, state) + time$

Interrupt$_{ComAct}$($subj$, $state$) =
SetCompletionPredicate$_{ComAct}$($subj$, $state$)
SetTimeoutExit$_{ComAct}$($subj$, $state$)
SetCompletionPredicate$_{ComAct}$($subj$, $state$) =
$Completed(subj, \text{ComAct}, state) := true$
SetTimeoutExit$_{ComAct}$($subj$, $state$) =
$TimeoutExitCond(subj, \text{ComAct}, state) := true$
Abrupt$_{ComAct}$($subj$, $state$) =
SetCompletionPredicate$_{ComAct}$($subj$, $state$)
SetAbruptionExit$_{ComAct}$($subj$, $state$)

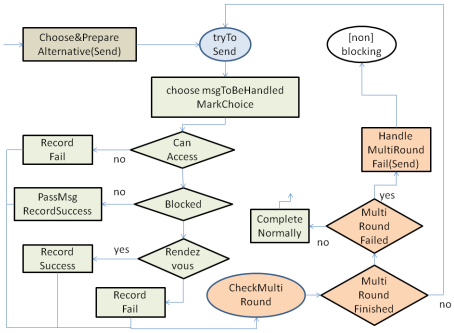
7.3 MsgElaboration Interpretation for Multi Send/Receive

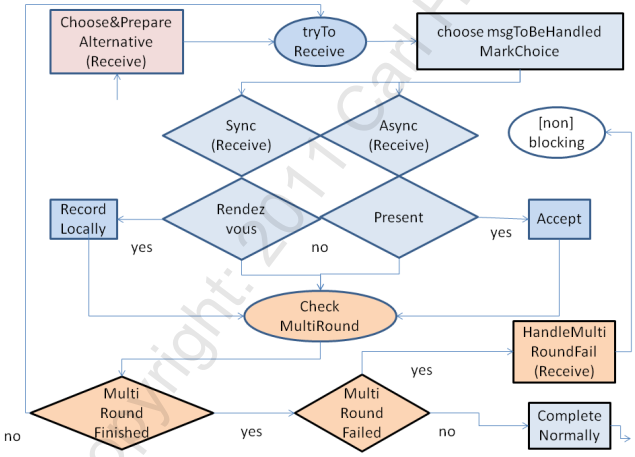
TryAlternative$_{ComAct}$($subj$, $state$) =
Choose&PrepareAlternative$_{ComAct}$($subj$, $state$)
seq Try$_{ComAct}$($subj$, $state$)

Choose&PrepareAlternative$_{ComAct}$($subj$, $state$) =
let $alt = select_{Alt}(RoundAlternative(subj, state), priority(state))$
PrepareMsg$_{ComAct}$($subj$, $state$, alt)
ManageAlternativeRound(alt, $subj$, $state$)
where
ManageAlternativeRound(alt, $subj$, $state$) =
MarkSelection($subj$, $state$, alt)
InitializeMultiRound$_{ComAct}$($subj$, $state$)
MarkSelection($subj$, $state$, alt) =
Delete(alt, $RoundAlternative(subj, state)$)

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

7.4 Multi Send/Receive Round Interpretation





$\text{TRY}_{ComAct}(subj, state) =$
  choose $m \in MsgToBeHandled(subj, state)$
    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 $\text{TRY}_{Async(Send)}(subj, state, m)$
    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
  MARKCHOICE$(m, subj, state) =$
    DELETE$(m, MsgToBeHandled(subj, state))$
    $currMsgKind(subj, state) := m$

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

7.5 Actual Send Interpretation

$\textsc{Async}(\mathit{Send})(subj, state, msg) =$
  $\textsc{PassMsg}(msg)$
  $\textsc{ContinueMultiRound}_{Success}(subj, state, msg)$
where
  $\textsc{PassMsg}(msg) =$
    let $pool = inputPool(receiver(msg))$
      $row = first(\{r \in constraintTable(pool) \mid$
        $ConstraintViolation(msg, r)\})$
          if $row \neq$ **undef and** $action(row) \neq DropIncoming$
            then $\textsc{Drop}(action)$
          if $row =$ **undef or** $action(row) \neq DropIncoming$ **then**
            $\textsc{Insert}(msg, pool)$
            $insertionTime(msg, pool) := now$
  $\textsc{Drop}(action) =$
    if $action = DropYoungest$ **then** $\textsc{Delete}(youngestMsg(pool), pool)$
    if $action = DropOldest$ **then** $\textsc{Delete}(oldestMsg(pool), pool)$
  $PossibleAsync_{Send}(subj, msg)$ iff **not** $Blocked(msg)$
  $Blocked(msg)$ iff
    let $row = first(\{r \in constraintTable(inputPool(receiver(msg))) \mid$
      $ConstraintViolation(msg, r)\})$
        $row \neq$ **undef and** $action(row) = Blocking$

$\textsc{Sync}(\mathit{Send})(subj, state, msg) =$
  $\textsc{ContinueMultiRound}_{Success}(subj, state, msg)$
$PossibleSync_{Send}(subj, msg)$ iff $RendezvousWithReceiver(subj, msg)$

$RendezvousWithReceiver(subj, msg)$ 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 $size(blockingRow) = 0$

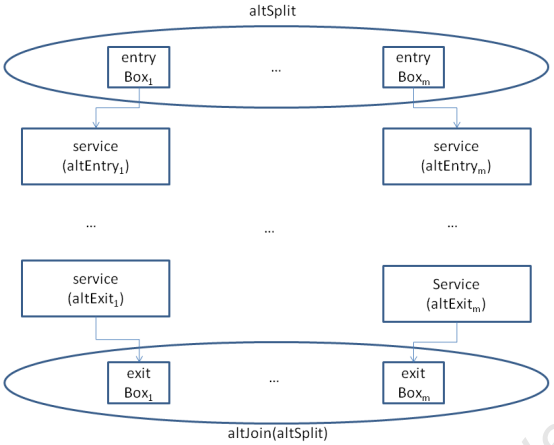
7.6 Actual Receive Interpreation

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

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

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

7.7 Alternative Action Interpretation



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**

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

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**

Auxiliary Wait/Exit Rule Interpretation

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

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

7.8 Interrupt Behavior

$\textsc{Behavior}_{D^*}(subj, state) =$ // Case of InterruptExtension(D, D')

$$\begin{cases} \textsc{Behavior}_{D}(subj, state) & \textbf{if } state \in D \setminus InterruptState \\ \textsc{Behavior}_{D'}(subj, state) & \textbf{if } state \in D' \\ \textsc{InterruptBehavior}(subj, state) & \textbf{if } state \in InterruptState \end{cases}$$

$\textsc{InterruptBehavior}(subj, s_i) =$ // at InterruptState s_i
  if $SID\_state(subj) = s_i$ **then**
    if $InterruptEvent(subj, s_i)$ **then**
      choose $msg \in InterruptMsg(s_i) \cap inputPool(subj)$[43]
      let $j = indexOf(interruptKind(msg), InterruptKind(s_i))$
        $handleState = target(arc_{i,j})$
          $\textsc{Proceed}(subj, service(handleState), handleState)$
          $\textsc{Delete}(msg, inputPool(subj))$
    else $\textsc{Behavior}_{D}(subj, s_i)$
where
  $InterruptEvent(subj, s_i)$ iff
    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.