# A Precise Description of the S-BPM Modeling Method

# 12

## 12.1 To Go

Now, I understand the benefit of seamless modeling and execution. It is a prerequisite for process development and optimization. However, I still struggle with the flow of communication. For instance, what happens in case I receive several messages simultaneously? Where and how is this handled?



For clarifying those issues, we have prepared a dedicated specification. It is also written in a special symbol language, which details the handling of the symbols of the S-BPM models or their diagrams.

Applying the method of Abstract State Machines allows us to eliminate all ambiguities when executing models and to precisely describe the behavior of an implementation of the models. The development of the formal language has also been guided by an orientation to natural language which should make it easy to understand after just a short briefing.


A. Fleischmann et al., *Subject-Oriented Business Process Management*,
DOI 10.1007/978-3-642-32392-8_12, © The Author(s) 2012

Well, when ambiguities occur, just look them up and you should find the answers in the precise specification. The starting issue is how input pools of subjects handle messages. Using formal S-BPM, we are able to check and specify the constraints for communication to be considered successful, namely if nothing that has been communicated gets lost.

This chapter presents a precise formulation of the S-BPM constructs discussed in the preceding chapters. We express them in the form of an abstract SBD-interpreter,[1] which yields a precise, controllable definition of the subject behavior in SBDs, the so-called semantics of SBDs. Furthermore, this definition establishes a solid scientific foundation for the S-BPM method to support a guarantee of the implementation correctness of the interpreter by the Metasonic modeling tool.[2] The correctness of the interpreter model concerns two levels: correctness of the interpreter with respect to the intended meaning of the modeling constructs (*ground model correctness*) and correctness of the interpreter implementation by the tool with respect to the interpreter (*refinement correctness*). Thus, the interpreter model represents a blueprint of the system and the double-faced correctness guarantees that the user understanding of processes and the result of their machine executions match, a feature that is crucial for reliable computer supported modeling systems.

Due to the survey character of this chapter, we only review here the main S-BPM modeling constructs and refer for a complete version of the interpreter model to the appendix.

## 12.2 Abstract State Machines

A precise definition of the meaning of business process modeling constructs provides a reliable basis for successful communication between the different stakeholders, namely designers and analysts on the management, development, and evaluation level, IT-specialists and programmers on the implementation level, and users on the application level. This needs a language that is common to the involved parties and allows to avoid the well-known problems of ambiguity of natural languages. This holds in particular for the S-BPM approach whose fundamental concepts—*actors*, which perform arbitrary *actions* on arbitrary *objects* and communicate with other actors—require most general heterogeneous data structures: sets of various elements with various operations and predicates (properties and relations) defined for them and agents, which execute those operations.

[1] SBD stands for subject behavior diagram.

[2] Such a guarantee must come in the form of a mathematical verification of appropriate interpreter and implementation properties, which is made possible by the precise character of the interpreter. This issue is not treated in this book.

The language of the so-called *Abstract State Machines* (ASMs) represents such a language. It uses only elementary If-Then-Else-rules, which are typical also for rule systems formulated in natural language, i.e., rules of the (symbolic) form

**if** *Condition* **then** ACTION

with arbitrary *Condition* and ACTION. The latter is usually a finite set of assignments of form $f(t_1, \ldots, t_n) := t$. The meaning of such a rule is to perform in any given state the indicated action if the indicated condition holds in this state.[3]

The unrestricted generality of the used notion of *Condition* and ACTION is guaranteed by using as ASM-*states* the so-called *Tarski structures,* i.e., arbitrary sets of arbitrary elements with arbitrary functions and relations defined on them. These structures are updatable by rules of the form above. In the case of business processes, the elements are placeholders for values of arbitrary type and the operations are typically the creation, duplication, deletion, or manipulation (value change) of objects. The so-called views are conceptually nothing else than projections (read: substructures) of such Tarski structures.

An (asynchronous, also called distributed) ASM consists of a set of agents each of which is equipped with a set of rules of the above form, called its program. Every agent can execute in an arbitrary state in one step all its rules which are executable, i.e., whose condition is true in the indicated state. For this reason, such an ASM, if it has only one agent, is also called sequential ASM. In general, each agent has its own "time" to execute a step, in particular if its step is independent of the steps of other agents;[4] in special cases multiple agents can also execute their steps simultaneously (in a synchronous manner).

This intuitive understanding of ASMs suffices to understand the definition of an SBD-interpreter given in this chapter. The subjects acting in an SBD are interpreted as agents, which at each diagram node execute their associated rules.

Without further explanations, we adopt usual notations, abbreviations, etc., for example:

**if** *Cond* **then** $M_1$ **else** $M_2$

instead of the equivalent ASM with two rules:

**if** *Cond* **then** $M_1$

**if not** *Cond* **then** $M_2$

Another notation used below is

**let** $x = t$ **in** $M$

[3] Usually, we write ASMs in capital letters as in ACTION, predicates beginning with capital followed by lower case letters as in *Condition*, and functions and terms with lower case letters as in $f$, $t_i$, $t$.

[4] This means that technically speaking a run of an asynchronous ASM is not a sequence of steps of an agent, but a set of such sequences defined by the involved agents, where steps $m$ of an agent which depend on steps $m'$ of another agent are in an order relation $m$ *before* $m'$ or $m$ *after* $m'$.

for *M(x/a),* where *a* denotes the value of *t* in the given state and *M(x/a)* is obtained from *M* by substitution of each (free) occurrence of *x* in *M* by *a*.

For details of a mathematical definition of the semantics of ASMs which justifies their intuitive (rule-based or pseudo-code) understanding, we refer the reader to the AsmBook (Börger and Stärk 2003). It contains also an explanation of the so-called refinement method which we use here to define the components of the SBD-interpreter in steps—a didactical concern adopted already in the preceding chapters of this book.

## 12.3 Interaction View of SBD-Behavior

An S-BPM *process* (short process) is defined as set of agents each of which is equipped with an SBD so that the process behavior can be defined by the SBD-behavior of its subjects (see Sect. 5.5.5). Thus, the definition of an S-BPM process interpreter as asynchronous ASM is reduced to the definition of a sequential ASM, which represents the interpreter BEHAVIOR$_{subj}$ *(D)* of an arbitrary subject *subj* in an arbitrary SBD-diagram *D*. For the interpretation of a process, this interpreter can then be replicated (read: instantiated) with each corresponding SBD.

A subject walks from node to node along the edges of *D,* beginning at the start node, and executes at each node the associated *service* until it reaches an end state. Therefore, the total behavior of the *subj*ect in *D* can be defined as set of each local BEHAVIOR*(subj, node)* of the *subj*ect at this *node* of *D*:

BEHAVIOR$_{\text{subj}}$ *(D)* = {BEHAVIOR(*subj, node)* | *node* ∈ *Node(D)*}

In this way, one can define SBD-computations of *subj* in the usual way as sequences $S_0, \ldots, S_n$ of (data*)* states of *subj* in the diagram which begin with an initial state $S_0$, i.e., a data state which has an initial SID-state,[5] lead to a state $S_n$ with a final SID-state and where each state $S_{i+1}$ is obtained from $S_i$ with SID-state $state_i$ by a step of BEHAVIOR*(subj,* $state_i$*)*.

Thus, the construction of an interpreter is decomposed into the definition of the behavior of a subject in a given state, represented in the diagram by a node, for each type of state. This directly supports the intuitive operational understanding of the single S-BPM constructs and simplifies the interpreter definition. Before proceeding to this definition in Sect. 12.3.2, we list in Sect. 12.3.1 the assumptions we make for the diagrams.

### 12.3.1 Diagrams

An SBD is a directed graph. Each *node* represents a state where a *subj*ect which is in this state performs the associated action *service(node)*. We call such a state an

[5] SID stands for Subject Interaction Diagram.

SID-state (Subject Interaction Diagram state) and denote it by *SID_state (subj)* since the abstract interpretation of *service (node)* refers only to the role the state plays with respect to other subjects with which *subj*ect communicates from within *D*. We speak without distinction about states as nodes.

Each SID-state has one of three types corresponding to the type of the associated *service*: *function state* (also called internal function or action state), *send state*, or *receive state*. Each SID-state is implicitly parameterized with the SBD in which it occurs, sometimes denoted by an index as in $SID\_state_D$ *(subject)* and *SID_state (subject, D)*. Each SID-state is part of the encompassing so-called *data state* or simply *state* (read: the underlying Tarski structure of the SBD).

The *edge*s which enter or exit a *node* represent the SID-state transitions from the source node *source(edge)* to *node* resp. from *node* to the target node *target(edge)*. Therefore, we call the *target(outEdge)* of an *outEdge* (an element of *OutEdge (node)*) also a successor state of *node* (in the diagram an element of the set *Successor (node)*) and *source (inEdge)* of an *inEdge* $\in$ *InEdge (node)* a predecessor state (an element of the set *Predecessor(node)*). A transition from a source to a target node is permitted only if the execution of the *service* associated to the source node is *Completed* so that each outgoing edge corresponds to a termination condition of the *service* and is typically indicated on the edge as *ExitCond*. We write $ExitCond_i$ for the *ExitCond* of the *i*-th outgoing edge if there is more than one.

Each SBD is finite and has exactly one initial and one end state. Each path is required to lead to at least one end state. It is permitted that an end state may have outgoing edges; a process terminates only if each of its subjects is in an end state.

### 12.3.2 SID-View of State Behavior

For the definition (of the SID-view) of BEHAVIOR *(subject, state)*, see Fig. 12.1. It describes the transition *subject* has to perform from a *SID_state* with associated *service A* to a next *SID_state* with associated *service* $B_i$ once the execution of *A* (using an abstract machine PERFORM) is *Completed*, where *subject* upon entering a state must START the associated *service*. The successor state *target(outEdge (state, i))* with its associated *service* $B_i$ is determined via a function $select_{Edge}$; it can be defined by the designer or at runtime by the executing subject.

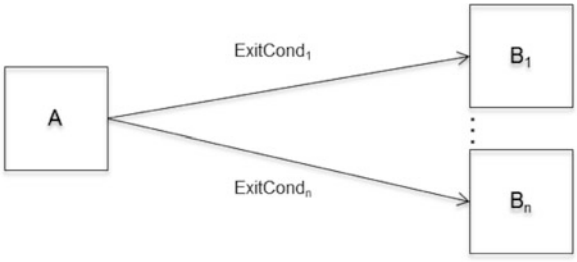


**Fig. 12.1** SID-transition graph structure

The following ASM-rule provides a compact textual description where the **else**-case expresses that it may take many steps until the execution of PERFORM for *A* by the executing subject is terminated.

```
BEHAVIOR(subj, state) =
  if SID_state (subj) = state then
    if Completed, (subj, service (state), state) then
      let edge =
        select_Edge ({e ∈ OutEdge (state) | 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.** BEHAVIOR *(subj, state)* is a scheme which comes with abstract machines PERFORM, START, and an abstract termination criterion *Completed* as components. It describes the interaction view of an SBD—that a subject upon entering a node STARTS the associated action and PERFORMS its steps until *Completed* becomes true—without providing details on how the component machines work and how they satisfy the termination criterion. The three constituents can and must be specified further to make the meaning of the performed action concrete. We do this in the next two sections for the S-BPM communication actions. The extension for the additional behavioral S-BPM constructs is given in the appendix.

## 12.4 Choice of Alternative Communication Steps

In this section, we define what it means to bring one step out of a set of so-called alternative communication steps to its execution. In this description, the meaning of a single such step still remains abstract and is refined in Sect. 12.5 by details of their multiprocess communication capabilities. In Sect. 12.4.1, we define the elements of the characteristic S-BPM input pool concept and formulate in Sect. 12.4.2 the first refinement of START, PERFORM, and *Completed* for sending and receiving; here the multiprocess communication capability still remains abstract. Since many definitions are symmetric in sending and receiving, we formulate them using a parameter *ComAct* for the corresponding *Com*munication *Act*ion.

### 12.4.1 Basics of the Input Pool Concept

To support asynchronous communication, which is typical for distributed systems, each *subject* has an *inputPool(subj)* where other subjects in the sender role may deposit messages and where *subj*ect in the receiver rule "expects" messages (i.e., looks for messages when it is ready to receive some).

Each *inputPool* can be configured by capacity bounds for the maximal number of messages it may contain of a specific or an arbitrary type and/or from a specific or arbitrary sender. All four possible cases (read: parameter pairs of arbitrary or specific type and sender) are considered (see Sect. 5.5.5.2).

To obtain a uniform description also for synchronous communication, 0 is allowed as value for the capacity parameters of an input pool. It is interpreted as requiring that the receiver expects to receive messages of the indicated type and/or from the indicated sender only via a rendezvous with the sender.

Asynchronous communication is determined by positive natural numbers for the input pool capacity parameters. Two strategies are contemplated for the case that a sender tries to deposit a message in an input pool that has reached already its corresponding capacity:

- *Canceling send* where either (a) a message is deleted from the input pool to enable the insertion of the incoming message or (b) the incoming message is thrown away (not inserted into the input pool).
- *Blocking send* where sending the message is blocked and the sender must repeat the attempt to send this message until either (a) an appropriate place has become free in the input pool, or (b) a timeout interrupts the attempt to send the message, or (c) the sender decides to abrupt the attempt to send the message.

For the first case, two versions to cancel are contemplated, namely to delete from the input pool the message which is *Present* there for the longest resp. shortest time, as described by two functions *oldestMsg* and *youngestMsg* defined in the appendix.

Whether an attempt to send is treated by an input pool $P$ of the receiver as canceling or blocking is a question of whether in the given state the capacity condition of $P$ would be violated by inserting the incoming message. These conditions are given by a *constraintTable(P)* in which the $i$-th row indicates for a combination of $sender_i$ and $msgType_i$ the allowed maximal number $size_i$ of messages of this kind, together with the $action_i$ to be performed in case of a capacity 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 attempts to deposit a *msg* in $P$ the first $row = s\ t\ n\ a$ in *constraintTable(P)* is identified (if there is one) whose capacity bound is relevant for *msg* and would be violated by inserting *msg*:

*ConstraintViolation(msg, row)* iff[6]
*Match (msg, row)* $\wedge$ *size* ({*m* $\in$ *P* | *Match (m, row)*}) + 1 > *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, the message can be inserted into *P*. Otherwise the action indicated in the identified row is performed so that either this attempt to send is blocked or the message is accepted via a cancellation action (possibly by directly throwing away the message).

It is required that each row with $size_i = 0$ satisfies $action_i = Blocking$ and that if $maxSize(P) < \infty$ holds, then the *constraintTable* contains the following default-row:
any any maxSize Blocking

Similarly, a receiver tries to transfer from its input pool into its data space an "expected" message (i.e., a message of the indicated *(msgType, sender)*) as we will see when interpreting a receive step.

In a distributed process at a given moment, multiple subjects may try to deposit a message in the input pool *P* of a same receiver, but only one subject can obtain the access to the resource *P*. Therefore, a selection mechanism is needed to determine this subject. We use a function $select_P$ which allows one to define the access predicate as follows:

*CanAccess(sender, P)* iff
$sender = select_P$ ({*subject* | *TryingToAccess(subject, P)*})

### 12.4.2 Iteration Structure of Alternative Communication Steps

In an *alternative communication state*, a subject performs the requested communication action *ComAct* by executing, until the communication succeeds (see Sects. 5.5.4.3 and 5.5.4.4), the following three steps, where *Alternative(subj, node)* is the set of all *ComAct*-alternatives the *subj*ect finds in the given state *node*:

- Selection: Choose from *Alternative (subj, node)* an *alt*ernative communication kind.
- Preparation: Prepare a *msgToBeHandled* which corresponds to the chosen *alt*ernative, that is in case of *ComAct* = *Send* a concrete *msgToBeSent* and otherwise a concrete *expectedMsg* kind.
- *ComAct*-attempt: $\text{TRYALTERNATIVE}_{ComAct}$, i.e., try—synchronously or involving the input pool—to send the concrete *msgToBeSent* resp. to accept a message that *Match*es the *expectedMsg* kind.

The first two steps (choice and preparation of the alternative) are done by a component $\text{CHOOSE\&PREPAREALTERNATIVE}_{\text{ComAct}}$ which represents the first step of $\text{TRYALTERNATIVE}_{\text{ComAct}}$ and is defined in Sect. 12.5.1.

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

If the third step fails for the chosen *alt*ernative, that is if *msgToBeHandled* cannot be sent resp. received neither asynchronously nor synchronously, the subject repeats the three steps for the next *alt*ernative until:

- Either *ComAct* succeeds for some alternative and the subject can set the predicate *Completed* for the *ComAct* (i.e., the *service*) in the given state *node* to true.
- Or *TryRoundFinished* holds, that is all *alt*ernatives have been tried without success.

In the second case, after this first so-called *nonblocking* round, further rounds of *ComAct*-attempts are started which are *blocking* in the sense that they can be terminated, besides by being normally *Completed*, also by a *Timeout* or by a *UserAbruption*. *Timeout* has higher priority than *UserAbruption*.

The set *RoundAlternative* of still to be tried alternatives must be initialized for each round to *Alternative (subj, node)*. This happens:

- For the *nonblocking*-round in START.
- For the first *blocking*-round in INITIALIZEBLOCKINGTRYROUNDS, where also the *Timeout*-clock is set.
- For each further round in InitializeRoundAlternatives.

Since the blocking rounds can be interrupted, to continue the computation via PROCEED the SBD must contain at least three edges leaving *node* to be taken after a normal or a forced *ComAct*-termination. Three predicates *NormalExitCond*, *TimeoutExitCond*, and *AbruptionExitCond* determine the outgoing edge which must be taken to reach the next SID-state if COMACT is normally *Completed* or ends by a *Timeout* or a *UserAbruption*. These three predicates are initialized in START, namely to *false*.

The following definition of PERFORM *(subj*, *ComAct*, *state)* synthesizes the preceding explanations in symbolic form. We write it down in the form of a traditional flowchart in Fig. 12.2. Such diagrams represent ASMs and thus have a precise semantics [see Börger et al. (2003, p. 44) and the equivalent textual definition in the appendix, where also the other more or less obvious and therefore here not listed component machines are defined].

**Macros and Components** of PERFORM*(subj,ComAct*, *state)*. We define here START*(subj*, *ComAct*, *state)*, INTERRUPT, and ABRUPT and refer for the other components to the appendix.

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* T, *state) := false*

INITIALIZECOMPLETIONPREDICATE$_{ComAct}$ *(subj, state) =*
*Completed (subj, ComAct, state) := false*

ENTER[NON]BLOCKINGTRYROUND (subj, *state) =*
*tryMode (subj, state) := [non]blocking*

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)*

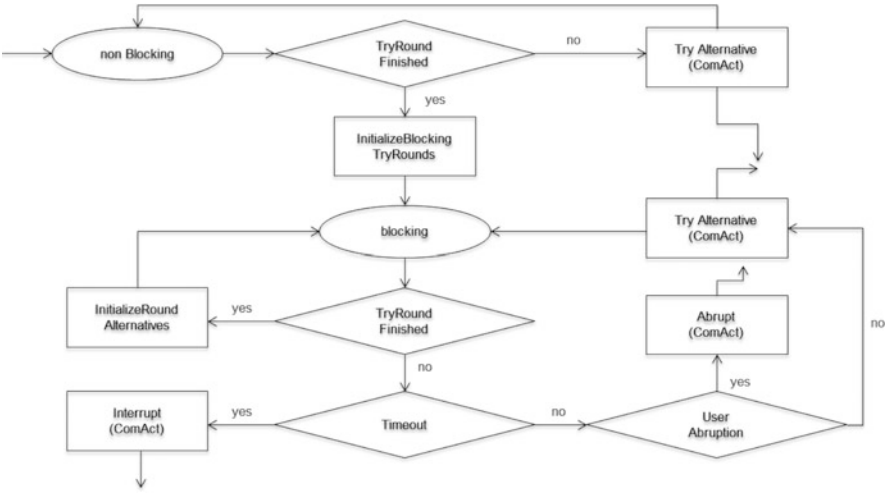


**Fig. 12.2** PERFORM *(subj, ComAct, state)*

## 12.5 MultiProcess-Communication

In this section, we refine TRYALTERNATIVE$_{\text{ComAct}}$ (and thereby by one more level of detail also PERFORM *(subj, ComAct, state)*) by a definition of the elements which enable this component for multiprocess communication in S-BPM (see Sect. 5.6.4).

As said in Sect. 12.4.2, the first TRYALTERNATIVE$_{ComAct}$ step consists in calling the CHOOSE&PREPAREALTERNATIVE$_{ComAct}$ component, followed by a call of the component

TRY$_{ComAct}$ to execute the *ComAct* for the chosen alternative and the corresponding prepared message(s) (if this *ComAct* is possible for the message(s)). This is synthesized in symbolic form by the following definition:[7]

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

The two components define the multiprocess character of S-BPM communication. Multiprocess communication means to communicate a bundle of *mult(alt)* >1 messages belonging to the chosen multi*alt*ernative. Bundling means that to successfully execute a multi*ComAct* a subject must successfully execute the *ComAct* for exactly the bundled messages that is *mult(alt)* many, without executing in between any other communication. Thus, executing a multi*ComAct* is a multiround of single *ComAct*s and appears as detailing one iteration step TRYALTERNATIVE$_{ComAct}$ of the TryRound described in Fig. 12.2.

A further characteristics of a multi*ComAct* in S-BPM consists in the requirement that (a) all relevant messages (those in the set *MsgToBeHandled)* must be prepared together before for each of them the execution of the *ComAct*-step is attempted and that (b) when the multi*ComAct* fails—that is if the *ComAct* fails for at least one of the bundled messages—the information on which *ComAct*-executions were successful resp. unsuccessful is available so that in case of failure the procedure HANDLEMULTIROUNDFAIL$_{ComAct}$ for error handling and possibly some compensation can be called.

We define CHOOSE&PREPAREALTERNATIVE$_{ComAct}$ in Sect. 12.5.1 und TRY$_{Send}$ and TRY$_{Receive}$ in Sect. 12.5.2.

### 12.5.1 Selection and Preparation of Messages

A *subject* can choose a communication *alt*ernative among those possible in a *state* in a nondeterministic manner or following a priority scheme. We express this by abstract functions *select*$_{Alt}$ and *priority* which can be refined as soon as a concrete state and the selection scheme intended there become known.

For each chosen communication alternative, the corresponding message to be sent resp. the kind of the to be received message (in case of a multicommunication the elements of the set *MsgToBeHandled*) must be prepared. This is done by the component PREPAREMSG$_{ComAct}$ described below.

Additionally a MANAGEALTERNATIVEROUND-component must guarantee that (a) each possible communication *alt*ernative in *Alternative (subj, state)* is selected in each TryRound exactly once and that (b) in case of a multicommunication *alt*ernative the multiround is initialized. For (a) in each round, the static set *Alternative (subj, state)* is copied into a dynamic set *RoundAlternative*.

[7] We use the **seq** operator [see Börger and Stärk (2003)] to describe sequential execution order for ASMs.

This description is synthesized in symbolic form by the following definition whose component PREPAREMSG is defined below:

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))*

Before sending a message, a subject will *composeMsg* from the relevant data, that is from the values of the underlying data structures, which are accessed via an abstract function *msgData*. Similarly in a given state, a receiver chooses one message kind out of those which are possible in this state for to be expected messages, using a selection function *select*$_{MsgKind}$. The abstract functions used here represent the interface to the underlying data states and can be refined as soon as the data structures become known. We assume only that there are functions *sender (msg)*, *type (msg)*, and *receiver (msg)* to extract the indicated information from a message; thus, *composeMsg* has to insert this information. Similarly for *expectedMsgKind* and *select*$_{MsgKind}$.

The preceding description defines the component PREPAREMSG$_{Send}$ and is symbolically synthesized as follows:

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)* := $\{m_1, .., m_{mult\ (alt)}\}$
  **if** *ComAct* = *Receive* **then**
    **let** $m_i$ = *select*$_{MsgKind\ (subj,\ state,\ alt,i)}$(*ExpectedMsgKind (subj, state, alt)*)
      *MsgToBeHandled (subj, state)* := $\{m_1, .., m_{mult\ (alt)}\}$

## 12.5.2 Sending and Receiving Messages

TRY$_{Send}$ is defined by the flowchart in Fig. 12.3, TRY$_{Receive}$ by the analogous only slightly different flowchart in Fig. 12.4.

Both diagrams describe for multicommunication nodes the multiround of a TryRound-*ComAct*-step: once a communication *alt*ernative has been selected and the corresponding set *MsgToBeHandled* has been prepared, during the multiround successively for each $m \in$ *MsgToBeHandled* an attempt is made to send resp. receive *m* performing the steps described below. After concluding the *ComAct* for an *m* (with success or failure), the subject continues the multiround for the next available $m \in$ *MsgToBeHandled*; at the end of the multiround in case of failure of

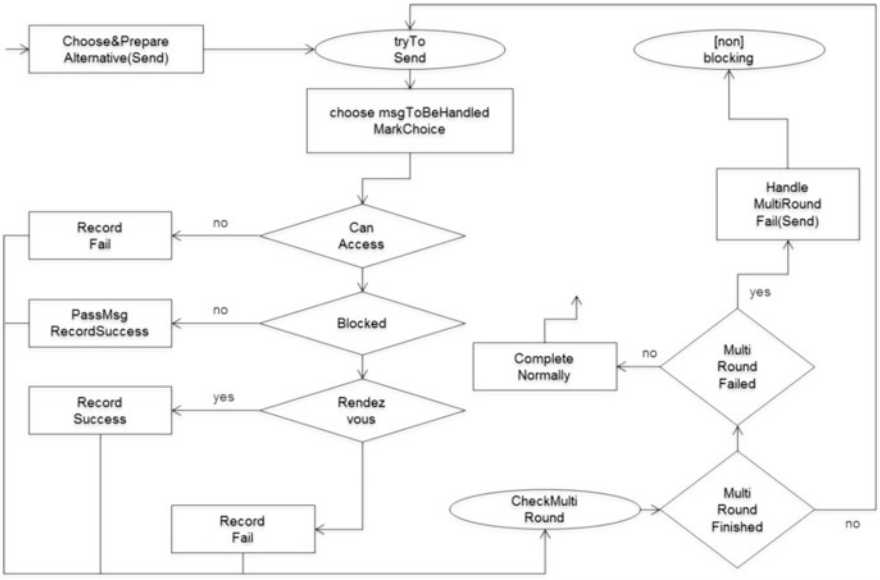


**Fig. 12.3** TRYALTERNATIVE$_{Send}$

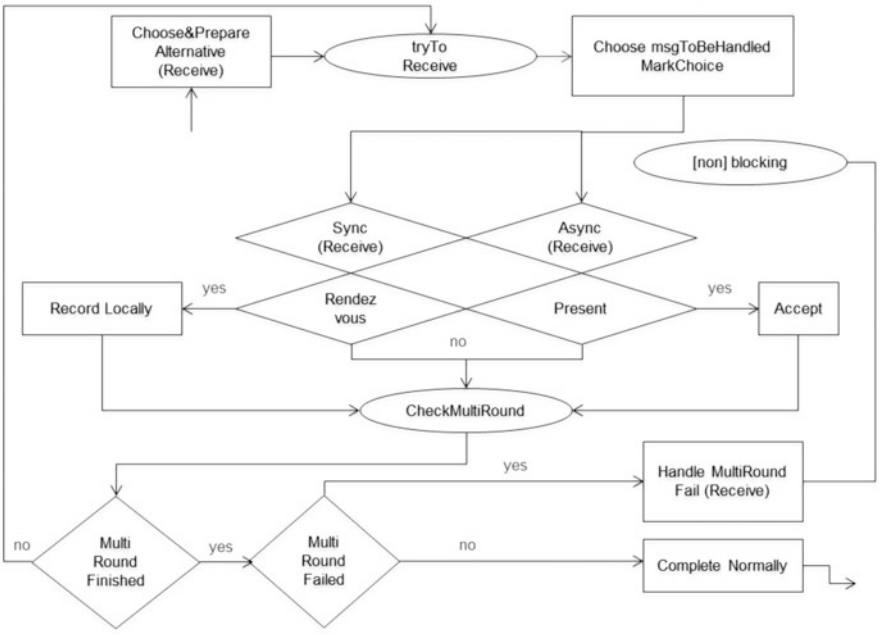


**Fig. 12.4** TRYALTERNATIVE$_{Receive}$

the *ComAct*, the subject proceeds to the next *alt*ernative, resp., in case of success, it sets *Completed* for this *ComAct* in this state to true.

Here are the steps in the order of their execution:

1. A sender checks whether it can access for *m* the input pool of the receiver. If the check outcome is negative, this attempt to send *m* fails. Otherwise, the sender proceeds to the next step.
2. Sender and receiver try to communicate *m* asynchronously. If sending *m* is not *Blocked* resp. if a message matching *m* is *Present* in the input pool of the receiver, *ComAct* succeeds for this *m*. Otherwise, the sender proceeds to the next step resp. the attempt to receive *m* fails.
3. Sender and receiver try to communicate *m* synchronously. If it succeeds, *ComAct* is successful for this *m*; otherwise, it fails for this *m*.

The meaning of the here not furthermore specified predicates and component machines (like passing a message to the input pool resp. to the local data space or transferring a message from the input pool to the local data space of the receiver) should be intuitively clear so that we refer for their detailed definition to the appendix, not to disrupt the synoptic character of this chapter.

## 12.6 Refinement for Internal Functions

Communication yields no deadlock even in the presence of communication alternatives (TryRound) and/or multicommunication (MultiRound) if one introduces a *Timeout* systematically for each communication node. This can be done also for internal functions by introducing *Timeout* and/or *UserAbruption* there too (see Sect. 5.7.6). It comes up to refine the SID-transition scheme in the **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, service(state),state)*

## Reference

Börger, E., Stärk R. Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, 2003.