Standard for the Subject-oriented Specification of Systems

Albert Fleischmann *Editor*

Standard for the Subject-oriented Specification of Systems

Working Document

Egon Börger
xyz

Stefan Borgert
xyz

Matthes Elstermann
xyz

Albert Fleischmann
xyz

Reinhard Gniza
xyz

Herbert Kindermann
xyz

Florian Krenn
xyz

Werner Schmidt
xyz

Robert Singer
FH JOANNEUM–University of Applied Sciences

Christian Stary
xyz

Florian Strecker
xyz

André Wolski
xyz

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version. Violations are liable to prosecution under the German Copyright Law.

© 2020 Institute of Innovative Process Management, Ingolstadt

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typeset by the authors

Production and publishing: XYZ

ISBN: 978-3-123-45678-9 (dummy)

Short contents

Preface

Contents

CHAPTER 1

Foundation

To facilitate the understanding of the following sections we will introduce the philosophy of subject-orienting modeling which is based on the Parallel Activity Specification Scheme (PASS). Additional, we will give a short introduction to ontologies—especially the Web Ontology Language (OWL)—, and to Abstract State Machines (ASM) as underlying concepts of this standard document.

1.1 Subject Orientation and PASS

In this section, we lay the ground for PASS as a language for describing processes in a subject-oriented way. This section is not a complete description of all PASS features, but it gives the first impression about subject-orientation and the specification language PASS. The detailed concepts are defined in the upcoming chapters.

The term subject has manifold meanings depending on the discipline. In philosophy, a subject is an observer and an object is a thing observed. In the grammar of many languages, the term subject has a slightly different meaning. "According to the traditional view, the subject is the doer of the action (actor) or the element that expresses what the sentence is about (topic)." [Kee76]. In PASS the term subject corresponds to the doer of an action whereas in ontology description languages, like RDF (see section 1.2), the term subject means the topic what the "sentence" is about.

1.1.1 Subject-driven Business Processes

Subjects represent the behavior of an active entity. A specification of a subject does not say anything about the technology used to execute the described behavior. This is different to other encapsulation approaches, such as multi-agent systems.

Subjects communicate with each other by exchanging messages. Messages have a name and a payload. The name should express the meaning of a message informally and the payloads are the data (business objects) transported. Internally, subjects execute local activities such as calculating a price, storing an address, etc.

A subject sends messages to other subjects, expects messages from other subjects, and executes internal actions. All these activities are done in sequences which are defined in a subject's behavior specification. Subject-oriented process specifications are always embedded in a context. A context is defined by the business organization and the technology by which a business process is executed.

Subject-oriented system development integrates established theories and concepts. It has been inspired by various process algebras (see e.g. [2], [3], [4]), by the basic structure of nearly all natural languages (Subject, Predicate, Object) and the systemic sociology developed by Niklas Luhmann (an introduction can be found in [5]). According to the organizational theory developed by Luhmann, the smallest organization consists of communication executed between at least two information processing entities [5]. The integrated concepts have been enhanced and adapted to organizational stakeholder requirements, such as providing a simple graphical notation, as detailed in the following sections.

1.1.2 Subject Interaction and Behavior

We introduce the basic concepts of process modeling in S-BPM using a simple order process. A customer sends an order to the order handling department of a supplier. He is going to receive an order confirmation and the ordered product by the shipment company. Figure 1.3 shows the communication structure of that process. The involved subjects and the messages they exchange can easily be grasped.

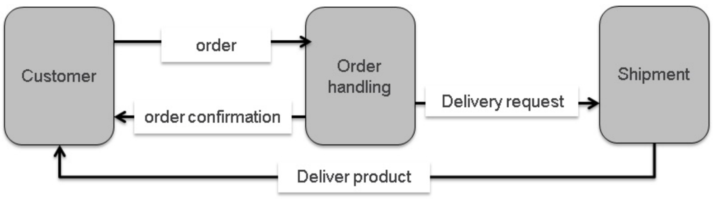


Figure 1.1: The Communication Structure in the Order Process

Each subject has a so-called input pool which is its mailbox for receiving messages. This input pool can be structured according to the business requirements at hand. The modeler can define how many messages of which type and/or from which sender can be deposited and what the reaction is if these restrictions are violated. This means the synchronization through message exchange can be specified for each subject individually.

Messages have an intuitive meaning expressed by their name. A formal semantics is given by their use and the data which are transported with a message. Figure 1.2 depicts the behavior of the subjects "customer" and "order handling".

In the first state of its behavior, the subject "customer" executes the internal function "Prepare order". When this function is finished the transition "order prepared" follows. In the succeeding state "send order" the message "order" is sent to the subject "order handling". After this message is sent (deposited in the input pool of subject "order handling"), the subject "Customer" goes into the state "wait for confirmation". If this message is not in the input pool the subject stops its execution until the corresponding message arrives in the input pool. On arrival, the subject removes the message from the input pool and follows the transition into state "Wait for product" and so on.

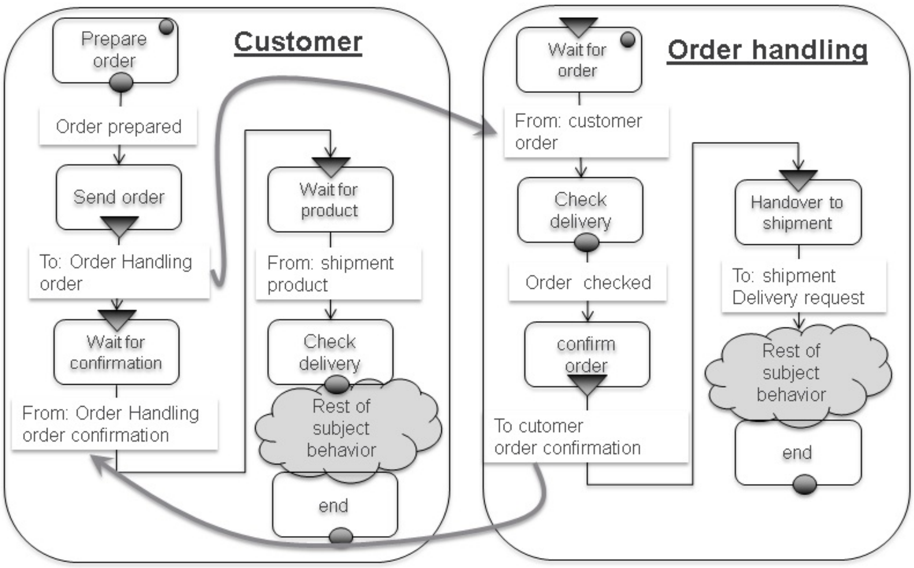


Figure 1.2: The Behavior of Subjects

The subject "Order Handling" waits for the message "order" from the subject "customer". If this message is in the input pool it is removed and the succeeding function "check order" is executed and so on.

The behavior of each subject describes in which order it sends messages, expects (receives) and performs internal functions. Messages transport data from the sending to the receiving subject and internal functions operate on internal data of a subject. These data aspects of a subject are described in section 1.1.3. In a dynamic and fast-changing world, processes need to be able to capture known but unpredictable events. In our example let us assume that a customer can change an order. This means the subject "customer" may send the message "Change order" at any time. Figure 1.3 shows the corresponding communication structure, which now contains the message "change order".

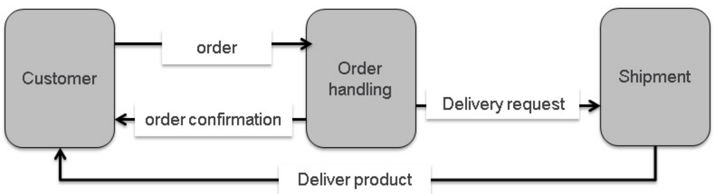


Figure 1.3: The Communication Structure with Change Message

Due to this unpredictable event, the behavior of the involved subjects needs also to be adapted. Figure 1.4 illustrates the respective behavior of the customer.

The subject "customer" may have the idea to change its order in the state "wait for confirmation" or in the state "wait for product". The flags in these states indicate that there is a so-called behavior extension described by a so-called non-deterministic event guard [12, 22]. The non-deterministic event created in the subject is the idea "change order". If this idea comes up, the current states, either "wait for confirmation" or "wait for product", are left, and the subject "customer" jumps into state "change order" in the guard behavior. In this state, the message

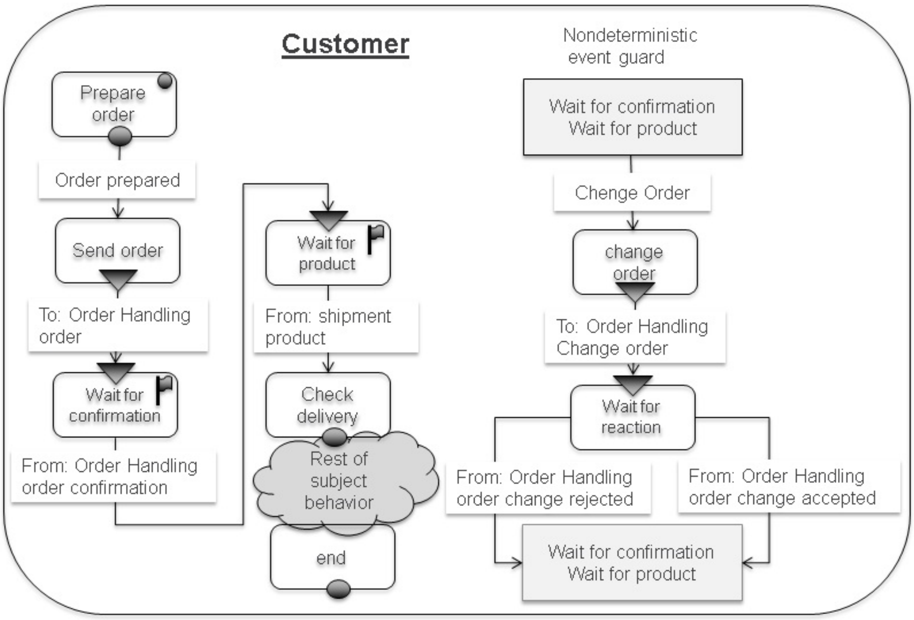


Figure 1.4: Customer is allowed to Change Orders

"change order" is sent and the subject waits in the state "wait for reaction". In this state, the answer can either be "order change accepted" or "order change rejected". Independently of the received message the subject "customer" moves to the state "wait for product". The message "order change accepted" is considered as confirmation, if a confirmation has not arrived yet (state "wait for confirmation"). If the change is rejected the customer has to wait for the product(s) he/she has ordered originally. Similar to the behavior of the subject "customer" the behavior of the subject "order handling" has to be adapted.

1.1.3 Subjects and Objects

Up to now, we did not mention data or the objects with their predicates, to get complete sentences comprising subject, predicate, and object. Figure 1.1.3 displays how subjects and objects are connected. The internal function "prepare order" uses internal data to prepare the data for the order message. This order data is sent as the payload of the message "order".

The internal functions in a subject can be realized as methods of an object or functions implemented in a service if a service-oriented architecture is available. These objects have an additional method for each message. If a message is sent, the method allows receiving data values sent with the message, and if a message is received the corresponding method is used to store the received data in the object [22]. This means either subject are the entities which use synchronous services as an implementation of functions or asynchronous services are implemented through subjects or even through complex processes consisting of several subjects. Consequently, the concept Service Oriented Architecture (SOA) is complementary to S-BPM: Subjects are the entities which use the services offered by SOAs (cf. [25]).

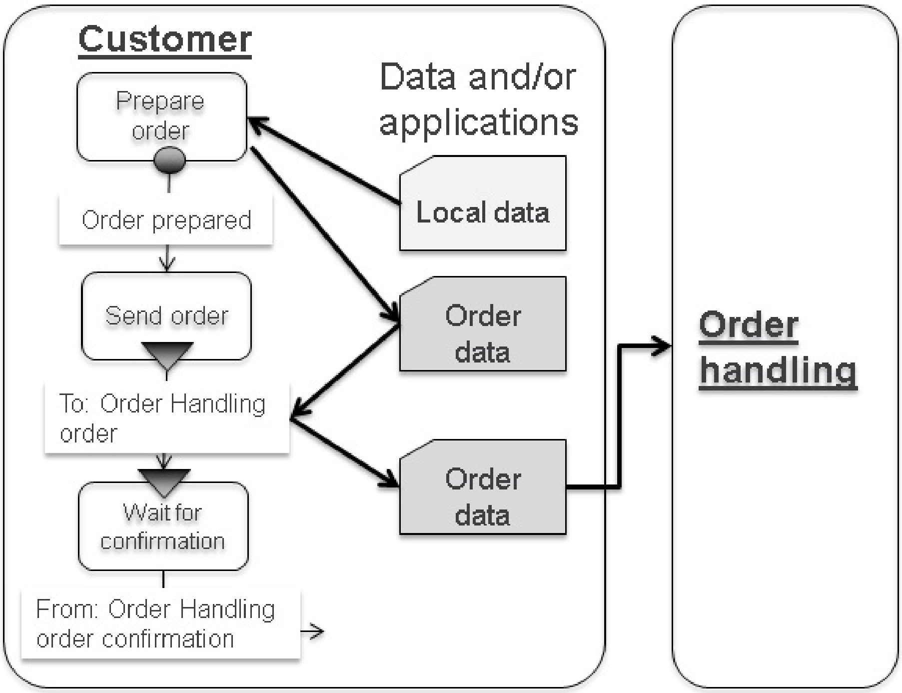


Figure 1.5: Subjects and Objects

1.2 Introduction to Ontologies and OWL

This short introduction to ontology, the Resource Description Framework and Web Ontology Language (OWL), should help to get an understanding of the PASS ontology outlined in section 2 and 3.

Ontologies are a formal way to describe taxonomies and classification networks, essentially defining the structure of knowledge for various domains: the nouns representing classes of objects and the verbs representing relations between the objects of classes.

In computer science and information science, an ontology encompasses a representation, formal naming, and definition of the classes, properties, and relations between the data, and entities that substantiate considered domains.

The Resource Description Framework (RDF) provides a graph-based data model or framework for structuring data as statements about resources. A "resource" may be any "thing" that exists in the world: a person, place, event, book, museum object, but also an abstract concept like data objects. Figure 1.6 shows an RDF graph.

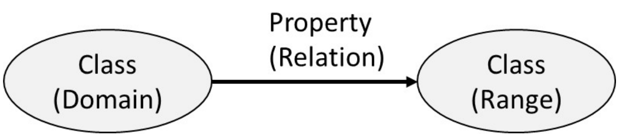


Figure 1.6: RDF graphic

RDF is based on the idea of making statements about resources (in particular web resources) in expressions of the form subject–predicate–object, known as triples. The subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. In the context of ontology, the term subject expresses what the sentence is about (topic) (see 1.1).

For describing ontologies several languages have been developed. One widely used language is OWL (worldwide web ontology language) which is based on the Resource Description Framework (RDF).

OWL has classes, properties, and instances. Classes represent terms also called concepts. Classes have properties and instances are individuals of one or more classes.

A class is a type of thing. A type of "resource" in the RDF sense can be person, place, object, concept, event, etc.. Classes and subclasses form a hierarchical taxonomy and members of a subclass inherit the characteristics of their parent class (superclass). Everything true for the parent class is also true for the subclass.

A member of a subclass "is a", or "is a kind of" its parent class. Ontologies define a set of properties used in a specific knowledge domain. In an ontology context, properties relate members of one class to members of another class or a literal.

Domains and ranges define restrictions on properties. A domain restricts what kinds of resources or members of a class can be the subject of a given property in an RDF triple. A range restricts what kinds of resources/members of a class or data types (literals) can be the object of a given property in an RDF triple.

Entities belonging to a certain class are instances of this class or individuals. A simple ontology with various classes, properties and individual is shown below:

Ontology statement examples:

- **Class definition statements:**
  - Parent `isA` Class
  - Mother `isA` Class
  - Mother `subClassOf` Parent
  - Child `isA` Class
- **Property definition statement:**
  - `isMotherOf` is a relation between the classes Mother and Child
- **Individual/instance statements:**
  - MariaSchmidt `isA` Mother
  - MaxSchmidt `isA` Child
  - MariaSchmidt `isMotherOf` MaxSchmidt

1.3 Introduction to Abstract State Machines

An abstract state machine (ASM) is a state machine operating on states that are arbitrary data structures (structure in the sense of mathematical logic, that is a nonempty set together with several functions (operations) and relations over the set).

The language of the so-called Abstract State Machine 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 the form *f (t1, ..., tn) := 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.

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 executable rules, 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; in special cases, multiple agents can also execute their steps simultaneously (in a synchronous manner).

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

```
if Cond then M1 else M2
```

instead of the equivalent ASM with two rules:

```
if Cond then M1
if not Cond then M2
```

Another notation used below is

```
let x=t in 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, E., Stärk R. Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, 2003.

CHAPTER 2

Structure of a PASS Description

In this chapter, we describe the structure of a PASS specification. The structure of a PASS description consists of the subjects and the messages they exchange.

2.1 Informal Description

2.1.1 Subject

Subjects represent the behavior of an active entity. A specification of a subject does not say anything about the technology used to execute the described behavior. Subjects communicate with each other by exchanging messages. Messages have a name and a payload. The name should express the meaning of a message informally and the payloads are the data (business objects) transported. Internal subjects execute local activities such as calculating a price, storing an address, etc. External subjects represent interfaces for other business processes.

A subject sends messages to other subjects, receives messages from other subjects, and executes internal actions. All these activities are done in logical order which is defined in a subject's behavior specification.

In the following, we use an example of the informal definition of subjects. In the simple scenario of the business trip application, we can identify three subjects, namely the employee as the applicant, the manager as the approver, and the travel office as the travel arranger.

In general, there are the following types of subjects:

- Fully specified subjects
- Multi-subjects
- Single subjects
- Interface subjects

Fully specified Subjects

This is the standard subject type. A subject communicates with other subjects by exchanging messages. Fully specified subjects consist of the following components:

- Business Objects—Each subject has some business objects. A basic structure of business objects consists of an identifier, data structures, and data elements. The identifier of a business object is derived from the business environment in which it is used. Examples are business trip requests, purchase orders, packing lists, invoices, etc. Business objects are composed of data structures. Their components can be simple data elements of a certain type (e.g., string or number) or even data structures themselves.

- Sent messages—Messages which a subject sends to other subjects. Each message has a name and may transport some data objects as a payload. The values of these payload data objects are copied from internal business objects of a subject.

- Received messages—Messages received by a subject. The values of the payload objects are copied to business objects of the receiving subject.

- Input Pool—Messages sent to subjects are deposited in the input pool of the receiving subject. The input pool is a very important organizational and technical concept in this case.

- Behavior—The behavior of each subject describes in which logical order it sends messages, expects (receives) messages, and performs internal functions. Messages transport data from the sending to the receiving subject and internal functions operate on internal data of a subject.

Multsubjects and Multiprocesses

Multi-subjects are similar to fully specified subjects. If in a process model several identical subjects are required, e.g. to increase the throughput, this requirement can be modeled by a multi-subject. If several communicating subjects in a process model are multi-subjects they can be combined to a multi-process.

In a business process, there may be several identical sub-processes that perform certain similar tasks in parallel and independently. This is often the case in a procurement process when bids from multiple suppliers are solicited. A process or sub-process is therefore executed simultaneously or sequentially multiple times during overall process execution. A set of type-identical, independently running processes or sub-processes are termed multi-process. The actual number of these independent sub-processes is determined at runtime.

Multi-processes simplify process execution since a specific sequence of actions can be used by different processes. They are recommended for recurring structures and similar process flows.

An example of a multi-process can be illustrated as a variation of the current booking process. The travel agent should simultaneously solicit up to five bids before making a reservation. Once three offers have been received, one is selected and a room is booked. The process of obtaining offers from the hotels is identical for each hotel and is therefore modeled as a multi-process.

Single subjects

Single subjects can be instantiated only once. They are used if for the execution of a subject a resource is required which is only available once.

Interface Subjects

Interface subjects are used as interfaces to other process systems. If a subject of a process system sends or receives messages from a subject which belongs to another workflow system. These so-called interface subjects represent fully described subjects which belong to that other process system. Interface subjects specifications contain the sent messages, received messages and the reference to the fully described subject which they represent.

2.1.2 Subject-to-Subject Communication

After the identification of subjects involved in the process (as process-specific roles), their interaction relationships need to be represented. These are the messages exchanged between the subjects. Such messages might contain structured information—so-called business objects.

The result is a model of the communication relationships between two or more subjects, which is referred to as a **Subject Interaction Diagram** (SID) or, synonymously, as a Communication Structure Diagram (CSD) (see figure 2.1).

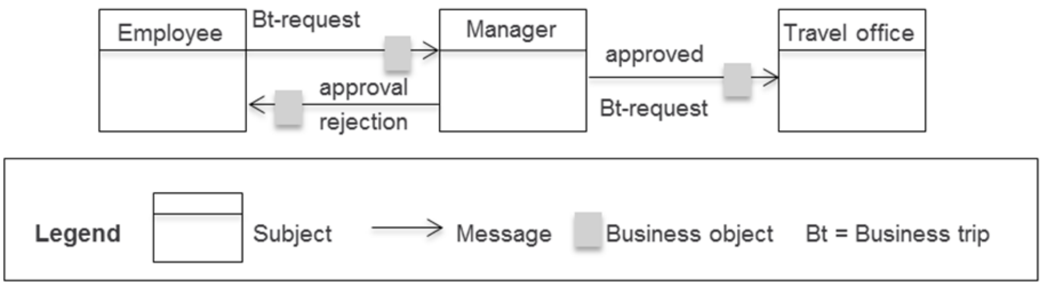


Figure 2.1: Subject interaction diagram for the process 'business trip application'

Messages represent the interactions of the subjects during the execution of the process. We recommend naming these messages in such a way that they can be immediately understood and also reflect the meaning of each particular message for the process. In the sample 'business trip application', therefore, the messages are referred to as 'business trip request', 'rejection', and 'approval'.

Messages serve as a container for the information transmitted from a sending to a receiving subject. There are two options for the message content:

- Simple data types—Simple data types are string, integer, character, etc. In the business trip application example, the message 'business trip request' can contain several data elements of type string (e.g., destination, the reason for traveling, etc.), and of type number (e.g., duration of the trip in days).

- Business Objects—Business Objects in their general form are physical and logical 'things' that are required to process business transactions. We consider data structures composed of elementary data types, or even other data structures, as logical business objects in business processes. For instance, the business object 'business trip request' could consist of the data structures 'data on applicants', 'travel data', and 'approval data' with each of these in turn containing multiple data elements.

2.1.3 Message Exchange

In the previous subsection, we have stated that messages are transferred between subjects and have described the nature of these messages. What is still missing is a detailed description of how messages can be exchanged, how the information they carry can be transmitted, and how subjects can be synchronized. These issues are addressed in the following sub-sections.

Synchronous and Asynchronous Exchange of Messages

In the case of an asynchronous exchange of messages, sender and receiver wait for each other until a message can be passed on. If a subject wants to send a message and the receiver (subject) is not yet in a corresponding receive state, the sender waits until the receiver can accept this message. Conversely, a recipient has to wait for the desired message until it is made available by the sender.

The disadvantage of the synchronous method is a close temporal coupling between sender and receiver. This raises problems in the implementation of business processes in the form of workflows, especially across organizational borders. As a rule, these also represent system boundaries across which a tight coupling between sender and receiver is usually very costly. For long-running processes, sender and receiver may wait for days, or even weeks, for each other.

Using asynchronous messaging, a sender can send anytime. The subject puts a message into a message buffer from which it is picked up by the receiver. However, the recipient sees, for example, only the oldest message in the buffer (in case the buffer is implemented as FIFO or LIFO storage) and can only accept this particular one. If it is not the desired message, the receiver is blocked, even though the message may already be in the buffer, but in a buffer space that is not visible to the receiver. To avoid this, the recipient has the alternative to take all of the messages from the buffer and manage them by himself. In this way, the receiver can identify the appropriate message and process it as soon as he or she needs it. In asynchronous messaging, sender and receiver are only loosely coupled. Practical problems can arise due to the in reality limited physical size of the receive buffer, which does not allow an unlimited number of messages to be recorded. Once the physical boundary of the buffer has been reached due to high occupancy, this may lead to unpredictable behavior of workflows derived from a business process specification. To avoid this, the input-pool concept has been introduced in PASS. Nevertheless, the number of messages must always be limited, as a business process must have the capacity to handle all messages to maintain some sort of service level.

Exchange of Messages via the Input Pool

To solve the problems outlined in the asynchronous message exchange, the input pool concept has been developed. Communication via the input pool is considerably more complex than previously shown; however, it allows transmitting an unlimited number of messages simultaneously. Due to its high practical importance, it is considered as a basic construct of PASS.

Consider the input pool as a mailbox of work performers, the operation of which is specified in detail. Each subject has its input pool. It serves as a message buffer to temporarily store messages received by the subject, independent of the sending communication partner. The input pools are therefore inboxes for flexible configuration of the message exchange between the subjects. In contrast to the buffer in which only the front message can be seen and accepted, the pool solution enables picking up (i.e. removing from the buffer) any message. For a subject, all messages in its input pool are visible.

The input pool has the following configuration parameters (see figure 2.2):

- Input-pool size—The input-pool size specifies how many messages can be stored in an input pool, regardless of the number and complexity of the message parameters transmitted with a message. If the input pool size is set to zero, messages can only be exchanged synchronously.
- Maximum number of messages from specific subjects—For an input pool, it can be determined how many messages received from a particular subject may be stored simultaneously in the input pool. Again, a value of zero means that messages can only be accepted synchronously.
- Maximum number of messages with specific identifiers—For an input pool, it can be determined how many messages of a specifically identified message type (e.g., invoice) may be stored simultaneously in the input pool, regardless of what subject they originate from. A specified size of zero allows only for synchronous message reception.
- Maximum number of messages with specific identifiers of certain subjects—For an input pool, it can be determined how many messages of a specific identifier of a particular subject may be stored simultaneously in the input pool. The meaning of the zero value is analogous to the other cases.

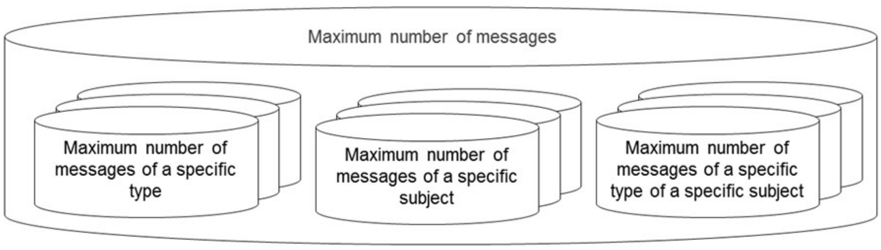


Figure 2.2: Configuration of Input Pool Parameters

By limiting the size of the input pool, its ability to store messages may be blocked at a certain point in time during process runtime. Hence, messaging synchronization mechanisms need to control the assignment of messages to the input pool. Essentially, there are three strategies to handle access to input pools:

- Blocking the sender until the input pool's ability to store messages has been reinstated—Once all slots are occupied in an input pool, the sender is blocked until the receiving subject picks up a message (i.e. a message is removed from the input pool). This creates space for a new message. In case several subjects want to put a message into a fully occupied input pool, the subject that has been waiting longest for an empty slot is allowed to send. The procedure is analogous if corresponding input pool parameters do not allow storing the message in the input pool, i.e., if the corresponding number of messages of the same name or from the same subject has been put into the input pool.

- Delete and release of the oldest message—In case all the slots are already occupied in the input pool of the subject addressed, the oldest message is overwritten with the new message.

- Delete and release of the latest message—The latest message is deleted from the input pool to allow depositing of the new incoming message. If all the positions in the input pool of the addressed subject are taken, the latest message in the input pool is overwritten with the new message. This strategy applies analogously when the maximum number of messages in the input pool has been reached, either concerning sender or message type.

2.2 OWL Description

The various building blocks of a PASS description and their relations are defined in an ontology. The following figure 2.3 gives an overview of the structure of the PASS specifications.

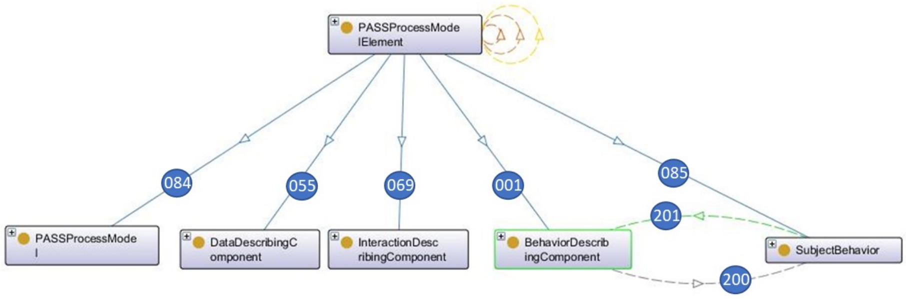


Figure 2.3: Elements of PASS Process Models

The class `PASSProcessModelElement` has five subclasses (subclass relations 084, 055, 069, 001 and 085 in figure 2.3). Only the classes `PASSProcess-Model`, `DataDescriptionCOmponent`, `InteractionDescribingComponent` are used for defining the structural aspects of a process specification in PASS. The classes `BehaviorDescribingComponent` and `SubjectBehavior` define the dynamic aspects. In which sequences messages are sent and received or internal actions are executed. These dynamic aspects are considered in detail in the next chapter.

2.2.1 PASS Process Model

The central entities of a PASS process model are subjects which represent the active elements of a process and the messages they exchange. Messages transport data from one subject to others (payload). Figure 2.4 shows the corresponding ontology for the PASS process models.

`PASSProcessModelElements` and `PASSProcessModells` have a name. This is described with the property `hasAdditionalAttribute` (property 208 in 2.3). The class subject and the class `MessageExchange` have the relation `hasRe-lation toModelComponent` to the class `PASSProcessModel` (property 226 in 2.3). The properties `hasReceiver` and `hasSender` express that a message has a sending and receiving subject (properties 225 and 227 in 2.3) whereas the properties `hasOutgoingMessageExchange` and `hasIncomingMessageExchange`

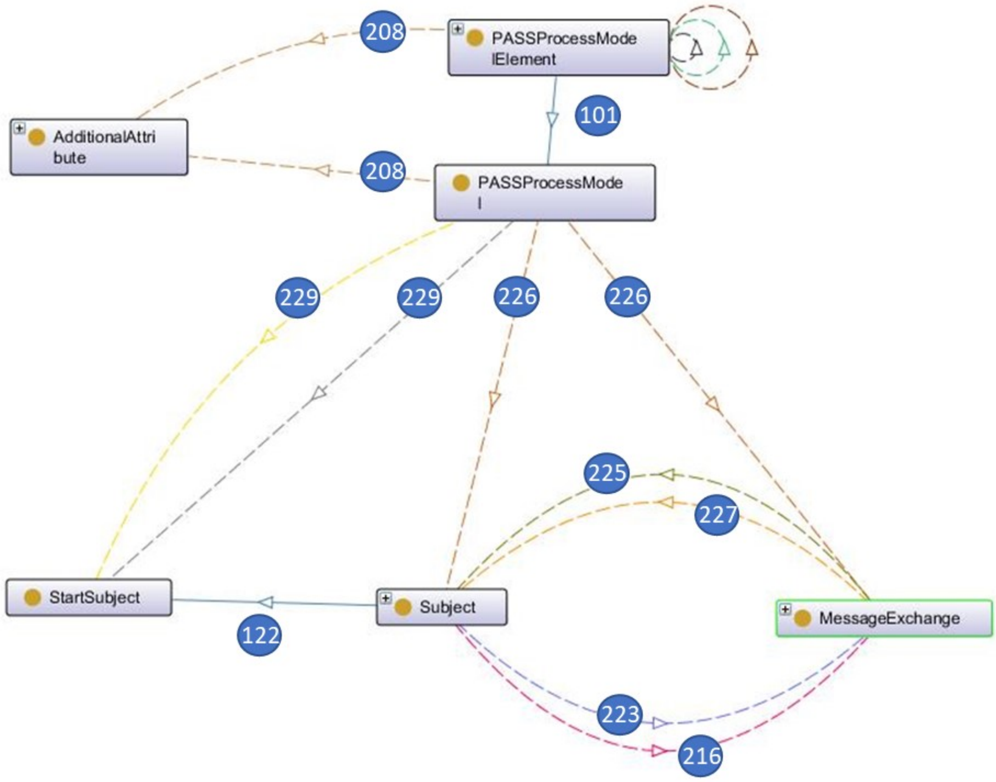


Figure 2.4: PASS Process Modell

define which messages are sent or received by a subject. Property `hasStart-Subject` (property 229 in 2.3) defines a start subject for a `PASSProcessModell`. A start subject is a subclass of the class subject (subclass relation 122 in 2.3).

2.2.2 Data Describing Component

Each subject encapsulates data (business objects). The values of these data elements can be transferred to other subjects. The following figure 2.5 shows the ontology of this part of the PASS-ontology.

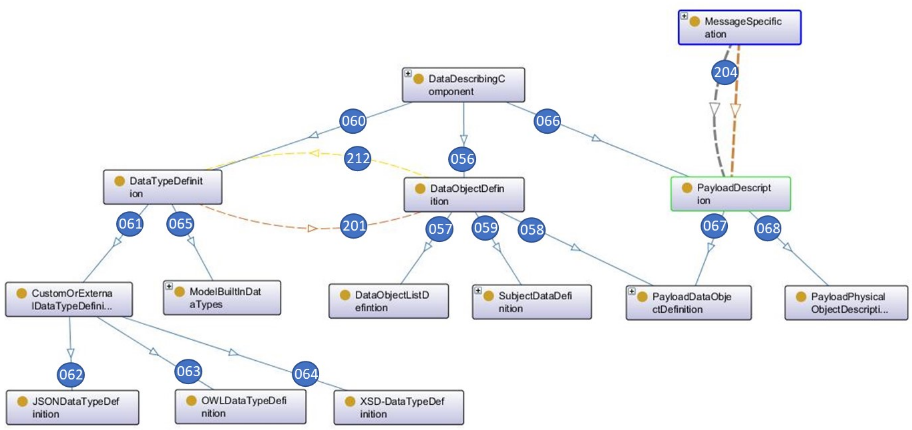


Figure 2.5: Data Description Component

Three subclasses are derived from the class `DtadescribingCombonent` (in figure 2.5 are these the relations 060, 056 and 066). The subclass `PayLoadDescription` defines the data tranported by messages. The relation of `PayloadDescriptions` to messages is defined by the property `ContainsPayloadDescription` (in figure 2.5 number 204).

There are two types of payloads. The class `PayloadPhysicalObjectDescription` is used if a message will be later implemented by a physical transport like a parcel. The class `PayLoadDataObjectDefinition` is used to transport normal data (Subclass relations 068 and 67 in figure 2.5). These payload objects are also a subclass of the class `DataObjectDefinition` (Subclass relation 058 in figure 2.5).

Data objects have a certain type. Therefore class `DataObjectDefinition` has the relation `hasDatatype` to class `DataTypeDefinition` (property 212 in figure2.5). Class `DataTypeDefinition` has two subclasses (subclass relations 061 and 065 in figure 2.5). The subclass `ModelBuiltInDataTypes` are user defined data types whereas the class `CustomOfExternalDataTypeDefinition` is the superclass of JSON, OWL or XML based data type definitions(subclass relations 062, 63 and 064 in figure 2.5).

2.2.3 Interaction Describing Component

The following figure 2.6 shows the subset of the classes and properties required for describing the interaction of subjects.

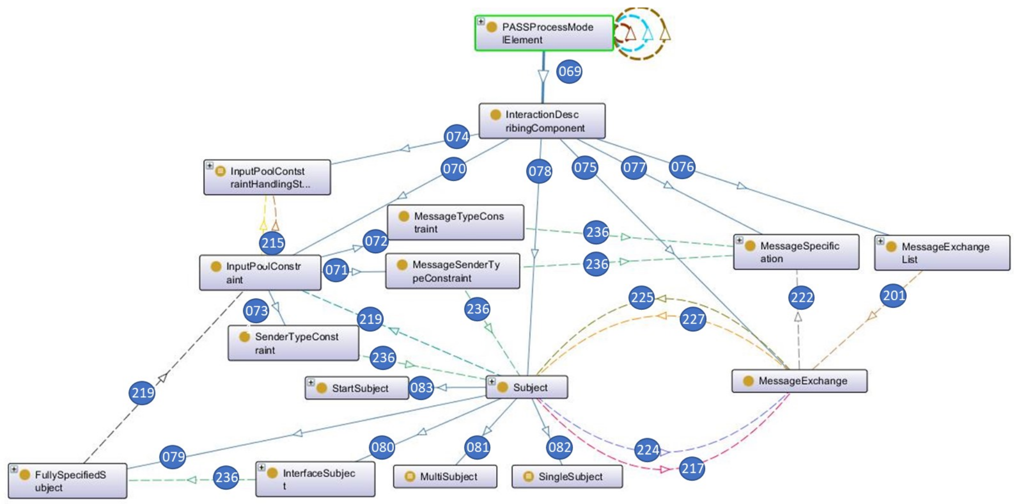


Figure 2.6: Subject Interaction Diagram

The central classes are `Subject` and `MessageExchange`. Between these classes are defined the properties `hasIncomingTransition` (in figure 2.6 number 217) and `hasOutgoingTransition` (in figure 2.6 number 224). This properties defines that subjects have incoming and outgoing messages. Each message has a sender and a receiver (in figure 2.6 number 227 and number 225). Messages have a type. This is expressed by the property `hasMessageType` (in figure 2.6 number 222). Instead of the property 222 a message exchange may have the property 201 if a list of messages is used instead of a single message.

Each subject has an input pool. Input pools have three types of constraints (see section 2.1.3). This is expressed by the property references (in figure 2.6

number 236) and `InputPoolConstraints` (in figure 2.6 number 219). Constraints which are related to certain messages have references to the class `MessageSpecification`.

There are four subclasses of the class `subject` (in figure 2.6 number 079, 080, 081 and 082). The specialties of these subclasses are described in section 2.1.1. A class `StartSubject` (in figure 2.6 number 83) which is a subclass of class subject denotes the subject in which a process instance is started.

All other relations are subclass relations. The class `PASSProcessModelElement` is the central PASS class. From this class, all the other classes are derived (see next sections). From class `InteractionDescribingCOmponent` all the classes required for describing the structure of a process system are derived.

2.3 ASM Description

In this chapter, only the structure of a PASS model is considered. Execution has not been considered. Because ASM only considers execution aspects in this chapter an ASM specification of the structural aspects does not make sense. The execution semantics is part of chapter 4.

CHAPTER 3

Execution of a PASS Model

3.1 Informal Description of Subject Behavior and its Execution

The execution of the subject means sending and receiving messages and executing internal activities in the defined order. In the following sections, it is described what sending and receiving messages and executing internal functions means.

3.1.1 Sending Messages

Before sending a message, the values of the parameters to be transmitted need to be determined. In case the message parameters are simple data types, the required values are taken from local variables or business objects of the sending subject, respectively. In the case of business objects, a current instance of a business object is transferred as a message parameter.

The sending subject attempts to send the message to the target subject and store it in its input pool. Depending on the described configuration and status of the input pool, the message is either immediately stored or the sending subject is blocked until delivery of the message is possible.

In the sample business trip application, employees send completed requests using the message 'send business trip request' to the manager's input pool. From a send state, several messages can be sent as an alternative. The following example shows a send state in which the message M1 is sent to the subject S1, or the message M2 is sent to S2, therefore referred to as alternative sending (see Figure 3.1). It does not matter which message is attempted to be sent first. If the send mechanism is successful, the corresponding state transition is executed. In case the message cannot be stored in the input pool of the target subject, sending is interrupted automatically, and another designated message is attempted to be sent. A sending subject will thus only be blocked if it cannot send any of the provided messages.

By specifying priorities, the order of sending can be influenced. For example, it can be determined that the message M1 to S1 has a higher priority than the message M2 to S2. Using this specification, the sending subject starts with sending message M1 to S1 and then tries only in case of failure to send message M2 to S2. In case of message M2 can also not be sent to the subject S2, the attempts to send start from the beginning.

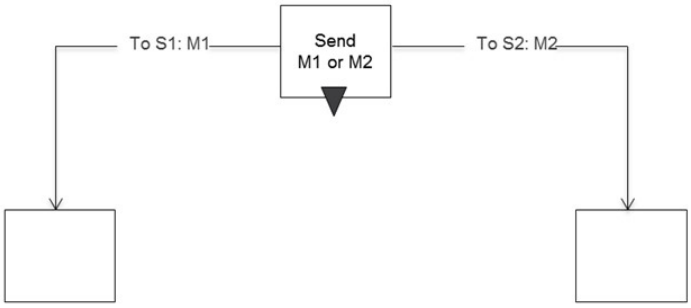


Figure 3.1: Example of alternative sending

The blocking of subjects when attempting to send can be monitored over time with the so-called timeout. The example in Figure 3.2 shows with 'Timeout: 24 h' an additional state transition which occurs when within 24 hours one of the two messages cannot be sent. If a value of zero is specified for the timeout, the process immediately follows the timeout path when the alternative message delivery fails.

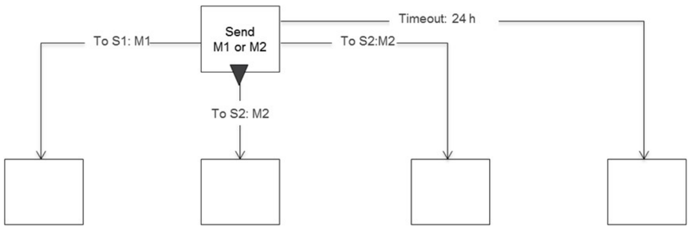


Figure 3.2: Send using time monitoring

3.1.2 Receiving Messages

Analogously to sending, the receiving procedure is divided into two phases, which run inversely to send.

The first step is to verify whether the expected message is ready for being picked up. In the case of synchronous messaging, it is checked whether the sending subject offers the message. In the asynchronous version, it is checked whether the message has already been stored in the input pool. If the expected message is accessible in either form, it is accepted, and in a second step, the corresponding state transition is performed. This leads to a takeover of the message parameters of the accepted message to local variables or business objects of the receiving subject. In case the expected message is not ready, the receiving subject is blocked until the message arrives and can be accepted.

In a certain state, a subject can expect alternatively multiple messages. In this case, it is checked whether any of these messages are available and can be accepted. The test sequence is arbitrary unless message priorities are defined. In this case, an available message with the highest priority is accepted. However, all other messages remain available (e.g., in the input pool) and can be accepted in other receive states.

Figure 3.3 shows a receive state of the subject 'employee' which is waiting for the answer regarding a business trip request. The answer may be an approval or a rejection.

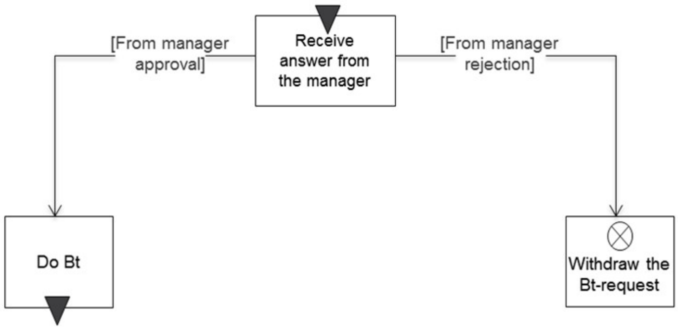


Figure 3.3: Example of alternative receiving

Just as with sending messages, also receiving messages can be monitored over time. If none of the expected messages are available and the receiving subject is therefore blocked, a time limit can be specified for blocking. After the specified time has elapsed, the subject will execute the transition as it is defined for the timeout period. The duration of the time limit may also be dynamic, in the sense that at the end of a process instance the process stakeholders assigned to the subject decide that the appropriate transition should be performed. We then speak of a manual timeout.

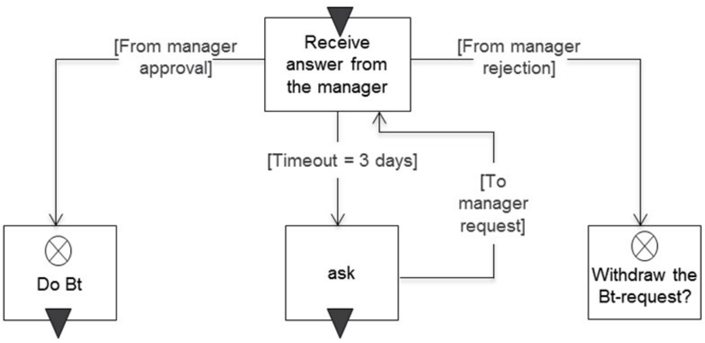


Figure 3.4: Time monitoring for message reception

Figure 3.4 shows that, after waiting three days for the manager's answer, the employee sends a corresponding request.

Instead of waiting for a message for a certain predetermined period of time, the waiting can be interrupted by a subject at all times. In this case, a reason for abortion can be appended to the keyword 'breakup'. In the example shown in Figure 3.5, the receiving state is left due to the impatience of the subject.

3.1.3 Standard Subject Behavior

The possible sequences of a subject's actions in a process are termed subject behavior. States and state transitions describe what actions a subject performs and how they are interdependent. In addition to the communication for sending and receiving, a subject also performs so-called internal actions or functions.

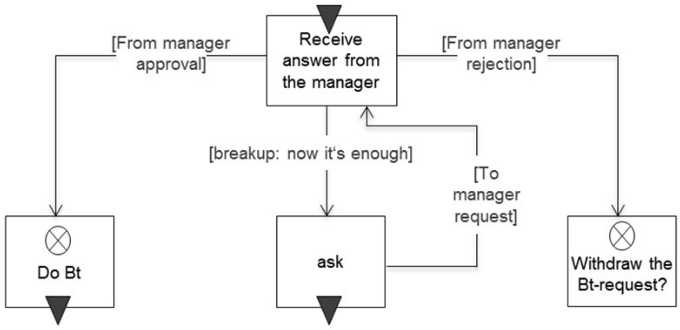


Figure 3.5: Message reception with manual interrupt

States of a subject are therefore distinct: There are actions on the one hand, and communication states to interact with other subjects (receive and send) on the other. This results in three different types of states of a subject. Figure 3.6 shows the different types of states with the corresponding symbols.

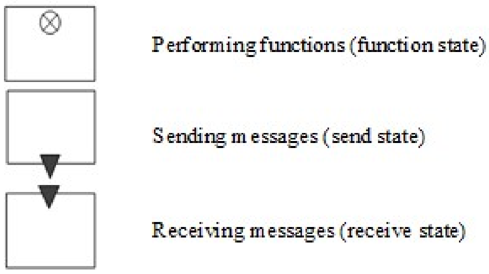


Figure 3.6: State types and coresponding symbols

In S-BPM, work performers are equipped with elementary tasks to model their work procedures: sending and receiving messages and immediate accomplishment of a task (function state).

In case an action associated with a state (send, receive, do) is possible, it will be executed, and a state transition to the next state occurs. The transition is characterized through the result of the action of the state under consideration: For a send state, it is determined by the state transition to which subject what information is sent. For a receive state, it becomes evident in this way from what subject it receives which information. For a function state, the state transition describes the result of the action, e.g., that the change of a business object was successful or could not be executed.

The behavior of subjects is represented by modelers using Subject Behavior Diagrams (SBD). Figure 3.7 shows the subject behavior diagram depicting the behavior of the subjects 'employee', 'manager', and 'travel office', including the associated states and state transitions.

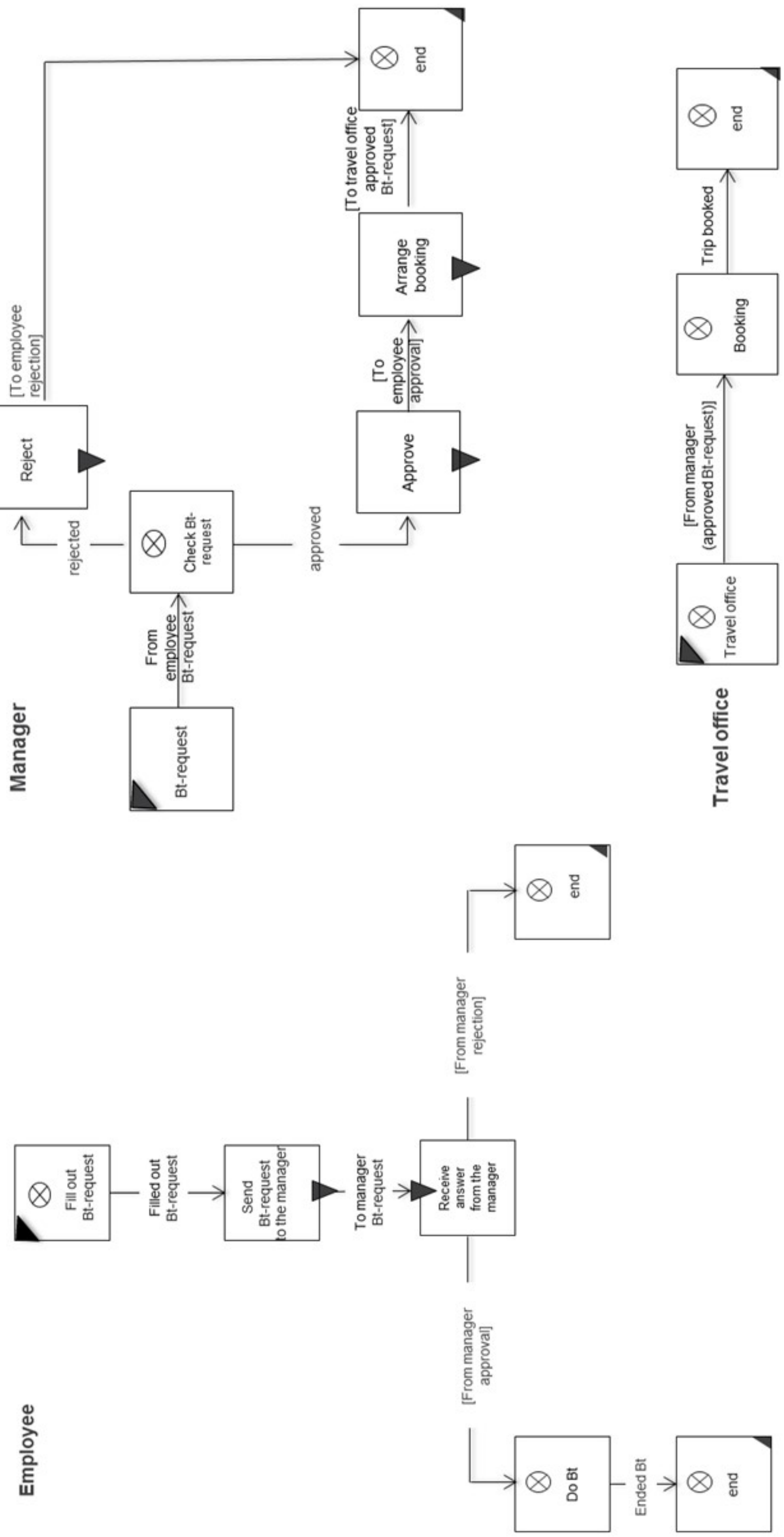


Figure 3.7: Subject behavior diagram for the subjects 'employee', 'manager', and 'travel office'

3.1.4 Extended Behavior

To reduce description efforts some additional specification constructs have been added to PASS. These constructs are informally explained in the following sections.

Macros

Quite often, a certain behavior pattern occurs repeatedly within a subject. This happens in particular when in various parts of the process identical actions need to be performed. If only the basic constructs are available to this respect, the same subject behavior needs to be described many times.

Instead, this behavior can be defined as a so-called behavior macro. Such a macro can be embedded at different positions of a subject behavior specification as often as required. Thus, variations in behavior can be consolidated, and the overall behavior can be significantly simplified.

The brief example of the business trip application is not an appropriate scenario to illustrate here the benefit of the use of macros. Instead, we use an example of order processing. Figure 3.8 contains a macro for the behavior to process customer orders. After placing the 'order', the customer receives an order confirmation; once the 'delivery' occurs, the delivery status is updated.

As with the subject, the start and end states of a macro also need to be identified. For the start states, this is done similarly to the subjects by putting black triangles in the top left corner of the respective state box. In our example, 'order' and 'delivery' are the two correspondingly labeled states. In general, this means that a behavior can initiate a jump to different starting points within a macro.

The end of a macro is depicted by gray bars, which represent the successor states of the parent behavior. These are not known during the macro definition.

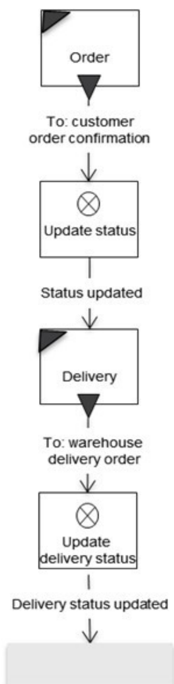


Figure 3.8: Behavior macro class 'request for approval'

Figure 3.9 shows a subject behavior in which the modeler uses the macro ’order processing’ to model both a regular order (with purchase order), as well as a call order.

The icon for a macro is a small table, which can contain multiple columns in the first line for different start states of the macro. The valid start state for a specific case is indicated by the incoming edge of the state transition from the calling behavior. The middle row contains the macro name, while the third row again may contain several columns with possible output transitions, which end in states of the surrounding behavior.

The left branch of the behavioral description refers to regular customer orders. The embedded macro is labeled correspondingly and started with the status ’order’, namely through linking the edge of the transition ’order accepted’ with this start state. Accordingly, the macro is closed via the transition ’delivery status updated’.

The right embedding deals with call orders according to organizational frameworks and frame contracts. The macro starts therefore in the state ’delivery’. In this case, it also ends with the transition ’delivery status updated’.

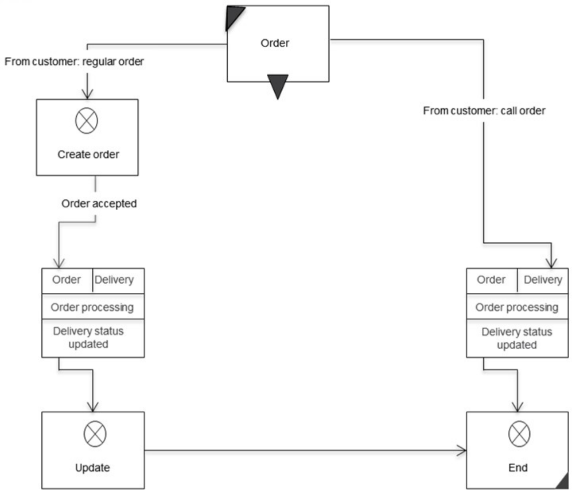


Figure 3.9: Subject behavior for order processing with macro integration

Similar subject behavior can be combined into macros. When being specified, the environment is initially hidden, since it is not known at the time of modeling.

Guards: Exception Handling and Extensions

Exception Handling— Handling of an exception (also termed message guard, message control, message monitoring, message observer) is a behavioral description of a subject that becomes relevant when a specific, exceptional situation occurs while executing a subject behavior specification. It is activated when a corresponding message is received, and the subject is in a state in which it can respond to the exception handling. In such a case, the transition to exception handling has the highest priority and will be enforced.

Exception handling is characterized by the fact that it can occur in a process in many behavior states of subjects. The receipt of certain messages, e.g., to abort the process, always results in the same processing pattern. This pattern would have to be modeled for each state in which it is relevant. Exception handling causes high modeling effort and leads to complex process models since from each affected state a corresponding transition has to be specified. To prevent this situation, we introduce a concept similar to exception handling in programming languages or interrupt handling in operating systems.

To illustrate the compact description of exception handling, we use again the service management process with the subject 'service desk' introduced in section 5.6.5. This subject identifies a need for a business trip in the context of processing a customer order—an employee needs to visit the customer to provide a service locally. The subject 'service desk' passes on a service order to an employee. Hence, the employee issues a business trip request. In principle, the service order may be canceled at any stage during processing up to its completion. Consequently, this also applies to the business trip application and its subsequent activities.

Below, it is first shown how the behavior modeling looks without the concept of exception handling. The cancellation message must be passed on to all affected subjects to bring the process to a defined end. Figure 3.10 shows the communication structure diagram with the added cancellation messages to the involved subjects.

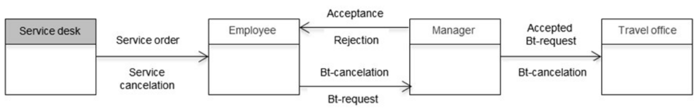


Figure 3.10: Communication structure diagram (CSD) of the business trip application

A cancellation message can be received by the employee either while filling out the application or while waiting for the approval or rejection message from the manager. Concerning the behavior of the subject 'employee', the state 'response received from manager' must also be enriched with the possible input message containing the cancellation and the associated consequences (see Figure 3.11). The verification of whether filing the request is followed by a cancellation is modeled through a receive state with a timeout. In case the timeout is zero, there is no cancellation message in the input pool and the business trip request is sent to the manager. Otherwise, the manager is informed of the cancellation and the process terminates for the subject 'employee'.

A corresponding adjustment of the behavior must be made for each subject which can receive a cancellation message, including the manager, the travel office, and the interface subject 'travel agent'.

This relatively simple example already shows that taking such exception messages into account can quickly make behavior descriptions confusing to understand. The concept of exception handling, therefore, should enable supplementing exceptions to the default behavior of subjects in a structured and compact form.

Instead of, as shown in Figure 3.11, modeling receive states with a timeout zero and corresponding state transitions, the behavioral description is enriched with the exception handling 'service cancellation'. Its initial state is labeled with the states from which it is branched to, once the message 'service cancellation' is received. In the example, these are the states 'fill out Bt-request' and 'receive

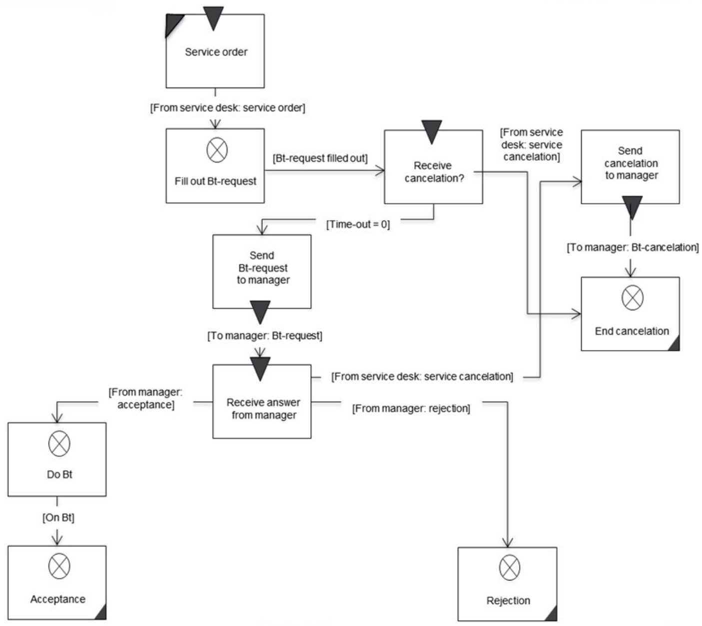


Figure 3.11: Handling the cancelation message using existing constructs

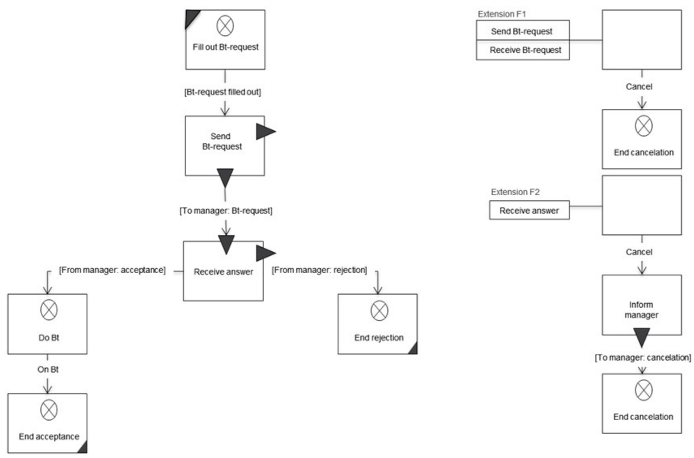


Figure 3.12: Behavior of subject ’employee’ with exception handling

answer from manager'. Each of them is marked by a triangle on the right edge of the state symbol. The exception behavior leads to an exit of the subject after the message 'service cancellation' has been sent to the subject 'manager'.

A subject behavior does not necessarily have to be brought to an end by an exception handling; it can also return from there to the specified default behavior. Exception handling behavior in a subject may vary, depending on from which state or what type of message (cancellation, temporary stopping of the process, etc.) it is called. The initial state of exception handling can be a receive state or a function state.

Messages, like 'service cancellation', that lead to exception handling always have higher priority than other messages. This is how modelers express that specific messages are read in a preferred way. For instance, when the approval message from the manager is received in the input pool of the employee, and shortly thereafter the cancellation message, the latter is read first. This leads to the corresponding abort consequences.

Since now additional messages can be exchanged between subjects, it may be necessary to adjust the corresponding conditions for the input-pool structure. In particular, the input-pool conditions should allow storing an interrupt message in the input pool. To meet organizational dynamics, exception handling and extensions are required. They allow taking not only discrepancies but also new patterns of behavior, into account.

Behavior Extensions— When exceptions occur, currently running operations are interrupted. This can lead to inconsistencies in the processing of business objects. For example, the completion of the business trip form is interrupted once a cancelation message is received, and the business trip application is only partially completed. Such consequences are considered acceptable, due to the urgency of cancelation messages. In less urgent cases, the modeler would like to extend the behavior of subjects in a similar way, however, without causing inconsistencies. This can be achieved by using a notation analogous to exception handling. Instead of denoting the corresponding diagram with 'exception', it is labeled with 'extension'.

Behavior extensions enrich a subject's behavior with behavior sequences that can be reached from several states equivocally.

For example, the employee may be able to decide on his own that the business trip is no longer required and withdraw his trip request. Figure 3.13 shows that the employee can cancel a business trip request in the states 'send business trip request to manager' and 'receive answer from manager'. If the transition 'withdraw business trip request' is executed in the state 'send business trip request to manager', then the extension 'F1' is activated. It leads merely to canceling of the application. Since the manager has not yet received a request, he does not need to be informed.

In case the employee decides to withdraw the business trip request in the state 'receive answer from manager', then extension 'F2' is activated. Here, first the supervisor is informed, and then the business trip is canceled.

Alternative Actions (Freedom of Choice)

So far, the behavior of subjects has been regarded as a distinct sequence of internal functions, send and receive activities. In many cases, however, the sequence of internal execution is not important.

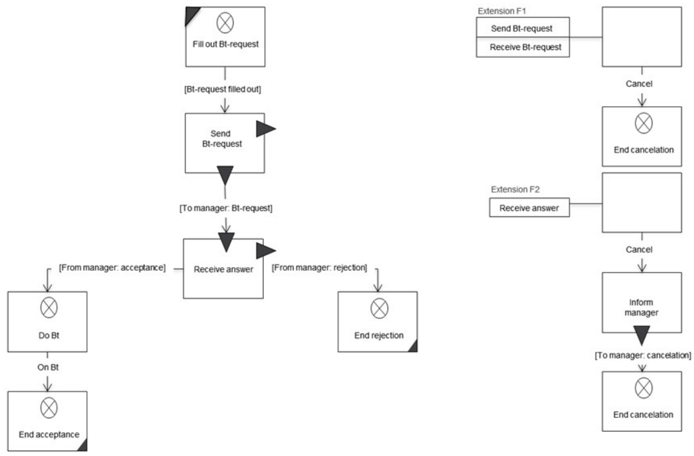


Figure 3.13: Subject behavior of employee with behavior extensions

Certain sequences of actions can be executed overlapping. We are talking about freedom of choice when accomplishing tasks. In this case, the modeler does not specify a strict sequence of activities. Rather, a subject (or concrete entity assigned to a subject) will organize to a particular extent its own behavior at runtime.

The freedom of choice with respect to behavior is described as a set of alternative clauses which outline several parallel paths. At the beginning and end of each alternative, switches are used: A switch set at the beginning means that this alternative path is mandatory to get started, a switch set at the end means that this alternative path must be completely traversed. This leads to the following constellations:

- Beginning is set/end is set: Alternative needs to be processed to the end.
- Beginning is set/end is open: Alternative must be started but does not need to be finished.
- Beginning is open/end is set: Alternative may be processed, but if so must be completed.
- Beginning is open/end is open: Alternative may be processed but does not have to be completed.

The execution of an alternative clause is considered complete when all alternative sequences, which were begun and had to be completed, have been entirely processed and have reached the end operator of the alternative clause.

Transitions between the alternative paths of an alternative clause are not allowed. An alternate sequence starts in its start point and ends entirely within its endpoint.

Figure 3.14 shows an example for modeling alternative clauses. After receiving an order from the customer, three alternative behavioral sequences can be started, whereby the leftmost sequence, with the internal function 'update order' and sending the message 'deliver order' to the subject 'warehouse', must be started in any case. This is determined by the 'X' in the symbol for the start of the alternative sequences (the gray bar is the starting point for alternatives). This sequence must be processed through to the end of the alternative because it is also marked in the end symbol of this alternative with an 'X' (gray bar as the endpoint of the alternative).

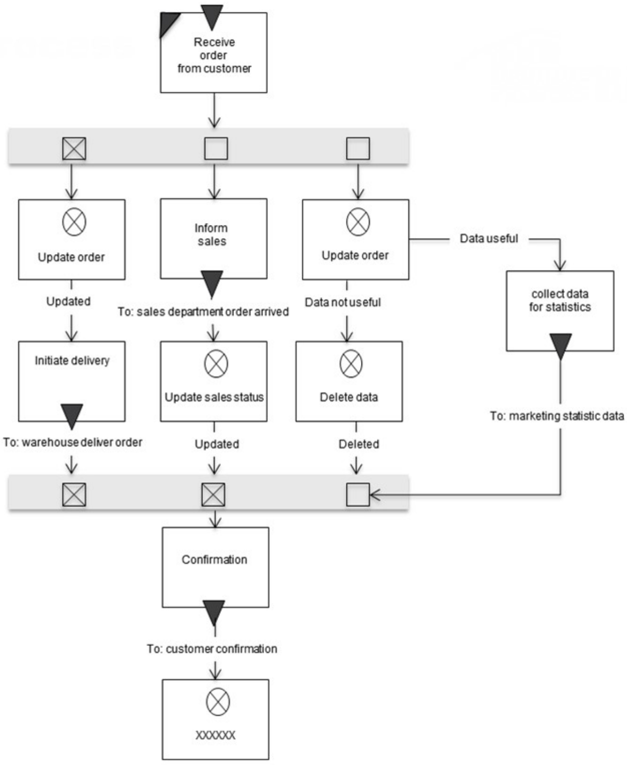


Figure 3.14: Example of Process Alternatives

The other two sequences may, but do not have to be, started. However, in case the middle sequence is started, i.e., the message 'order arrived' is sent to the sales department, it must be processed to the end. This is defined by an appropriate marking in the end symbol of the alternatives ('X' in the lower gray bar as the endpoint of the alternatives). The rightmost path can be started but does not need to be completed.

The individual actions in the alternative paths of an alternative clause may be arbitrarily executed in parallel and overlapping, or in other words: A step can be executed in an alternative sequence, and then be followed by an action in any other sequence. This gives the performer of a subject the appropriate freedom of choice while executing his actions.

In the example, the order can thus first be updated, and then the message 'order arrived' sent to sales. Now, either the message 'deliver order' can be sent to the warehouse or one of the internal functions, 'update sales status' or 'collect data for statistics', can be executed.

The left alternative must be executed completely, and the middle alternative must also have been completed, if the first action ('inform sales' in the example) is executed. Only the left alternative can be processed because the middle one was never started. Alternatively, the sequence in the middle may have already reached its endpoint, while the left is not yet complete. In this case, the process waits until the left one has reached its endpoint. Only then will the state 'confirmation' be reached in the alternative clause. The right branch neither needs to be started, nor to be completed. It is therefore irrelevant for the completion of the alternative construct.

The leeway for freedom of choice with regards to actions and decisions associated with work activities can be represented through modeling the various alternatives—situations can thus be modeled according to actual regularities and preferences.

3.2 Ontology of Subject Behavior Description

Each subject has a base behavior (see property 202 in 3.15) and may have additional subject behaviors (see class `SubjectBehavior` in 3.15) for macros and guards. All these behaviors are subclasses of the class `SubjectBehavior`. The details of these behaviors are defined as state transition diagrams (PASS behavior diagrams). These behavior diagrams are represented in the ontology with the class `BehaviorDescribingComponent` (see figure 3.15). The behavior diagrams have the relation `belongsTo` to the class `SubjectBehavior`. The other classes are needed for `embeddingsubjects` into the subject interaction diagram (SID) of a PASS specification (see chapter 2.2).

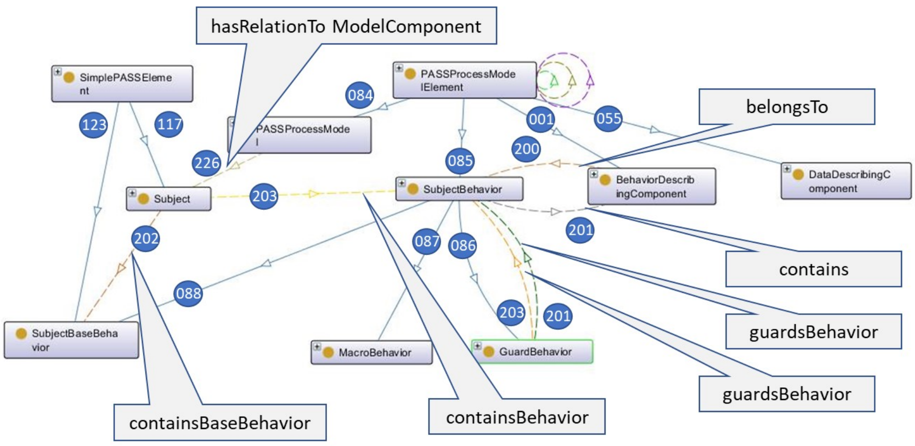


Figure 3.15: Structure of Subject Behavior Specification

3.2.1 Behavior Describing Component

The following figure shows the details of the class `BehaviorDescribingComponent`. This class has the subclasses State, Transition and `TranssitionCondition` (see figure 3.16). The subclasses of the state represent the various types of states (class relations 025, 014 und 024 in 3.16). The standard states `DoSTates`, `SendState` and `ReceiveState` are subclasses of the class `StandardPASSState` (class relations 114, 115 und 116 in 3.16). The subclass relations 104 and 020 allow that there exist a start state (class `InitialStatOfBehavior` in 3.16) and none or several end states (see subclass relation 020 in figure3.16) The fact that there must be at least one start state and none or several end states is defined by so called axioms which are not shown in figure 3.16.

States can be starting and/or endpoints of transitions (see properties 228 and 230 in figure 3.16). This means a state may have outgoing and/or incoming tran-

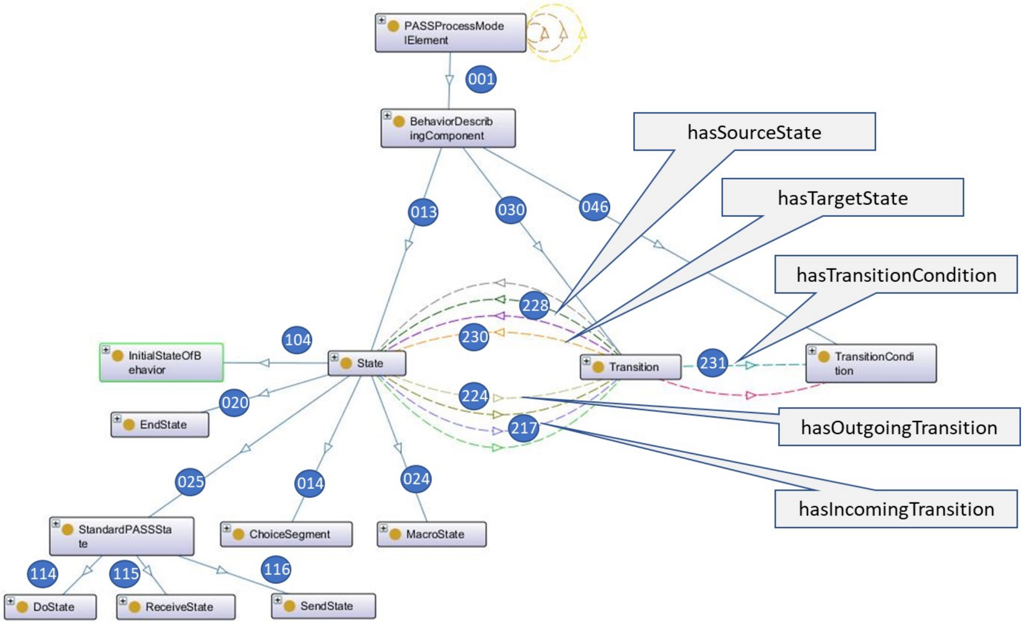


Figure 3.16: Subject Behavior describingComponent

sitions (see properties 224 and 217 in figure 3.16). Each transition is controlled by a transition condition which must be true before a behavior follows a transition from the source state to the target state.

States

As shown in figure 3.17 the class state has a subclass `StandardPASSState` (subclass relation 025) which have the subclasses `ReceiveState`, `SendState` and `DoState`(subclass relations 027, 026, 025). A state can be a start state (subclass `InitialStateOfBehavior` subclass relation 022). Besides these standard states there are macro states (subclass 024). Macro states contain a reference (subclass 029) to the corresponding macro (Property 201).

More complex states are choice segments (subclass relation 014). A choice segment contains choice segment paths (subclass 015 and property 200). Each choice segment path can be of one of four types. If a segment path is started than it must be finished or not or a segment path must be started and must be finished or not (subclass relations 16, 17, 18 and 19).

Transitions

Transitions connect the source state with the target state (see figure 3.16). A transition can be executed if the transition condition is valid. This means the state of a behavior changes from the current state which is the source state to the target state. In PASS there are two basic types of transitions, `DoTransitions` and `CommunicationTranstions` (subclasses 34 and 31 in figure 3.18). The class `CommunicationTransition` is divided into the subclasses `ReceiveTransition` and `SendTransition` (subclasses 32 and 33 in figure 3.18). Each transition has depending from its type a corresponding transition condition (property 231 in figure 3.18) which defines a data condition which must be valid in in order to execute a transition.

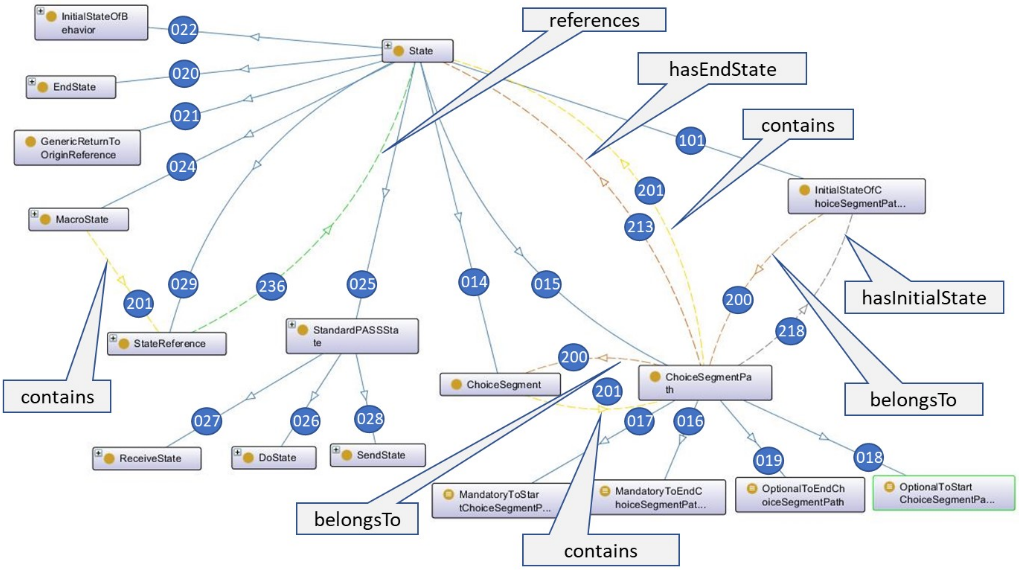


Figure 3.17: Details of States

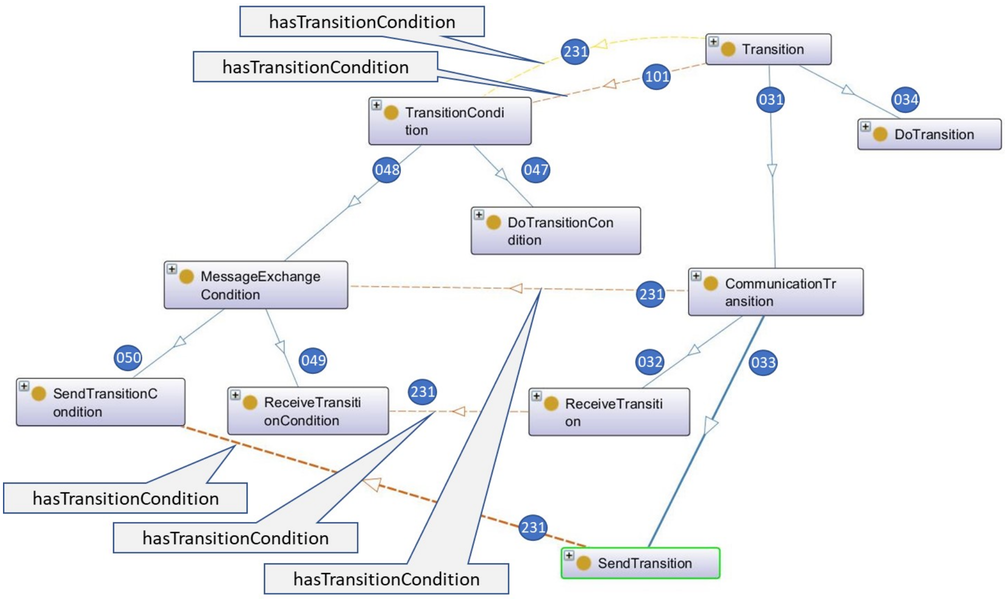


Figure 3.18: Details of transitions

3.3 ASM Definition of Subject Execution

`Behavior(subj;state)` is the ASM-Rule to interpret a specific node of a behavior diagram for a specific subject.

It expresses that when a subject in a given state has completed a given action (function, send or receive operation)—read: Performing the action has been completed while the subject was in the given SID-state. Assuming that the action has been started by the subject upon entering this state-then the subject proceeds to start its next action in its successor state. The successor state is determined by an `ExitCondition` whose value is defined by the just-completed action. Figure B.1 shows the ASM code for the behavior rule.

```
Behavior_subj(D) = {Behavior(subj ; node) | node  Node(D)}

Behavior(subj ; state) =
if SID state(subj ) = state then
  if Completed(subj ; service(state); state) then
    let edge =
        selectEdge ({e ϵ 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)
```

Figure 3.19: ASM Code for Behavior

Because of historical reasons in the ASM the names of the objects in the ontology are not used. Table 3.1 shows the mapping of the variable names used in the ASM (in ASM also called places) to the ontology object names. The names printed in *italic* in the column "ASM-Interpreter" corespond to the **bold** printed names in the column "OWL Model". The OWL model element **State** coresponds to the place *SID_state* or *D* is mapped to **SubjectBehaviour**.

OWL Model element	ASM interpreter	Description
X - Execution concept – the state the subject is currently in as defined by a **State** in the model	*SID_state*	Execution concept – no model representation, Not to be confused by a model "state" in an SBD Diagram. State in the SBD diagram define possible SID_States.
SubjectBehavior – under the assumption that it is complete and sound.	*D*	A Diagram that is a completely connected SBD
State	*node*	A specific element of diagram D - Every node 1:1 to state
State	*state*	The current active state of a diagram determined by the nodes of Diagram D
InitialStateOfBehavior, EndState	*initial state, end state*	The interpreter expects and SBD Graph D to contain exactly one initial (start) state and at least one end state.
Transition	*edge / outEdge*	"Passive Element" of an edge in an SBD-graph
TransitionConditionn	*ExitCondition*	Static Concept that represents a Data condition
Execution Concept – ID of a Subject Carrier responsible possible multiple Instances of according to specific **SubjectBehavior**	*subj*	Identifier for a specific Subject Carrier that may be responsible for multiple Subjects
Represented in the model with **InterfaceSubject**	*ExternalSubject*	A representation of a service execution entity outside of the boundaries of the interpreter (The PASS-OWL Standardization community decided on the new Term of Interface Subject to replace the often-misleading older term of External Subject)

OWL Model element	ASM interpreter	Description
SubjectBehavior or rather **SubjectBaseBehavior** as MacroBehaviors and GuardBehaviors are not covered by Börger	*subject-SBD* / $SBDsubject_{subject}$	Names for completely connected graphs / diagrams representing SBDs
Object Property: ***hasFunctionSpecification*** (linking **State**, and **FunctionSpecification** –>(**State** ***hasFunctionSpecification*** **FunctionSpecification**)	*service(state)* / *service(node)*	Rule/Function that reads/returns the service of function of a given state/node
DoState **SendState** **ReceiveState**	*function state*, *send state*, *receive state*	The ASM spec does not itself contain these terms. The description text, however, uses them to describe states with an according service (e.g. a state in which a (ComAct = Send) service is executed is referred to as a send state) Seen from the other side: a SendState is a state with service(state) = Send) Both send and receive services are a ComAct service. The ComAct service is used to define common rules of these communication services.
CommunicationActs with sub-classes (**ReceiveFunction SendFunction**) ***DefaultFunctionReceive1_-EnvironmentChoice*** ***DefaultFunctionReceive2_-AutoReceiveEarliest*** ***DefaultFunctionSend***	*ComAct*	Specialized version of Perform-ASM Rule for communication, either send or receive. These rules distinguish internally between send and receive.

OWL Model element	ASM interpreter	Description

Table 3.1: Mapping of ASM places to objects of the ontology

The complete ASM based definition of PASS was developed by Egon Börger and can be found in Annex B

3.3.1 Internal Functions/Action

A detailed internal Behavior of a subject in a state with internal function A can be defined in terms of the ASM 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 section 3.3.3 but only once it is known how to start, to perform and to complete A. For example, Start(subj ;A; state) could mean to call function A which is implemented as a methode of a class. The completion predicate coincides with the termination condition of the method.

The complete ASM based definition of PASS was developed by Egon Börger and can be found in Annex B.

3.3.2 Internal Functions/Action

A detailed internal `Behavior` of a subject in a state with internal function A can be defined in terms of the ASM sub-machines `Start` and `Perform` together with the predicate `Completed` for the parameters `(subj ;A; state)` in the same manner as has been done for communication actions in section 3.3.3 but only once it is known how to start, to perform and to complete A. For example, `Start(subj ;A; state)` could mean to call function A which is implemented as a method of a class. The completion predicate coincides with the termination condition of the method. »»»> 9542de488102928f002a500d8df07552608d9983

3.3.3 Communication Action

```
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
        INTERRUPT_ComAct(subj ; state)
    elseif UserAbruption(subj ; state)
      then Abrupt_ComAct(subj ; state)
      else TRYALTERNATIVE_ComAct(subj ; state)
```

Figure 3.20:

APPENDIX A

Classes and Properties of the PASS Ontology

A.1 All Classes (95)

- SRN = Subclass Reference Number; Is used for marking the coresponding relations in the following figures. The number identifies the subclass relation to the next level of super class.
- PASSProcessModelElement
  - BehaviorDescribingComponent; SRN: 001
    Group of PASS-Model components that describe aspects of the behavior of subjects
    - Action; SRN: 002
      An Action is a grouping concept that groups a state with all its outgoing valid transitions
    - DataMappingFunction ; SRN: 003
      Standard Format for DataMappingFunctions must be define: XML? OWL? JSON? Definitions of the ability/need to write or read data to and from a subject's personal data storage. DataMappingFunctions are behavior describing components since they define what the subject is supposed to do (mapping and translating data) Mapping may be done during reception of message, where data is taken from the message/Business Object (BO) and mapped/put into the local data field. It may be done during sending of a message where data is taken from the local vault and put into a BO. Or it may occur during executing a do function, where it is used to define read(get) and write (set) functions for the local data.
      - DataMappingIncomingToLocal ; SRN: 004
        A DataMapping that specifies how data is mapped from an an external source (message, function call etc.) to a subject's private defined data space.
      - DataMappingLocalToOutgoing ; SRN: 005
        A DataMapping that specifies how data is mapped from a subject's private data space to an an external destination (message, function call etc.)
    - FunctionSpecification ; SRN: 006
      A function specification for state denotes
      Concept: Definitions of calls of (mostly technical) functions (e.g. Web-service, Scripts, Database access,) that are not part of the process model.
      Function Specifications are more than "Data Properties"? –> - If special function types (e.g. Defaults) are supposed to be reused, having them as explicit entities is a the better OWL-modeling choice.
      - CommunicationAct ; SRN: 007
        A super class for specialized FunctionSpecification of communication acts (send and receive)

        - ReceiveFunction ; SRN: 008
          Specifications/descriptions for Receive-Functions describe in detail what the subject carrier is supposed to do in a state.
          DefaultFunctionReceive1_EnvoironmentChoice : present the surrounding execution environment with the given exit choices/conditions currently available depending on the current state of the subjects in-box. Waiting and not executing the receive action is an option.
          DefaultFunctionReceive2_AutoReceiveEarliest: automatically execute the according activity with the highest priority as soon as possible. In contrast to DefaultFunctionReceive1, it is not an option to prolong the reception and wait e.g. for another message.
        - SendFunction ; SRN: 009
          Comments have to be added
    - DoFunction ; SRN: 010
      Specifications or descriptions for Do-Functions describe in detail what the subject carrier is supposed to do in an according state. The default DoFunction
      1: present the surrounding execution environment with the given exit choices/-conditions and receive choice of one exit option –> define its Condition to be fulfilled in order to go to the next according state. The default DoFunction
      2: execute automatic rule evaluation (see DoTransitionCondition - ToDo) More specialized Do-Function Specifications may contain Data mappings denoting what of a subjects internal local Data can and should be:
      a) read: in order to simply see it or in order to send it of to an external function (e.g. a web service)
      b) write: in order to write incoming Data from e.g. a web Service or user input, to the local data fault
- ReceiveType ; SRN: 011
  Comments have to be added
- SendType ; SRN: 012
  Comments have to be added
- State ; SRN: 013
  A state in the behavior descriptions of a model
    - ChoiceSegment ; SRN: 014
      ChoiceSegments are groups of defined ChoiceSegementPaths. The paths may contain any amount of states. However, those states may not reach out of the bounds of the ChoiceSegmentPath.
    - ChoiceSegmentPath ; SRN: 015
      ChoiceSegments are groups of defined ChoiceSegementPaths. The paths may contain any amount of states. However, those states may not reach out of the bounds of the ChoiceSegmentPath.The path may contain any amount of states but may those states may not reach out of the bounds of the choice segment path. Similar to an initial state of a behavior a choice segment path must have one determined initial state. A transition within a choice segment path must not have a target state that is not inside the same choice segment path.
        - MandatoryToEndChoiceSegmentPath ; SRN: 016
          Comments have to be added
        - MandatoryToStartChoiceSegmentPath ; SRN: 017
          Comments have to be added
        - OptionalToEndChoiceSegmentPath ; SRN: 018
          Comments have to be added
        - OptionalToStartChoiceSegmentPath ; SRN: 019
          ChoiceSegmentPath and (isOptionalToEndChoiceSegmentPath value false)
    - EndState ; SRN: 020
      An end state a behavior. A subject behavior may have one or more end states. Only Do and Receive states may be end states. Send States cannot be end states.There are no individual end states that are not Do, Send, or Receive States at the same time.
    - GenericReturnToOriginReference ; SRN: 021
      Comments have to be added

- InitialStateOfBehavior ; SRN: 022
  The initial state of a behavior
- InitialStateOfChoiceSegmentPath ; SRN: 023
  Similar to an initial state of a behavior a choice segment path must have one determined initial state
- MacroState ; SRN: 024
  A state that references a macro behavior that is executed upon entering this state. Only after executing the macro behavior this state is finished also.
- StandardPASSState ; SRN: 025
  A super class to the standard PASS states: Do, Receive and Send
  - DoState ; SRN: 026
    The standard state in a PASS subject behavior diagram denoting an action or activity of the subject in itself.
  - ReceiveState ; SRN: 027
    The standard state in a PASS subject behavior diagram denoting an receive action or rather the waiting for a receive possibility.
  - SendState ; SRN: 028
    The standard state in a PASS subject behavior diagram denoting a send action
- StateReference ; SRN: 029
  A state reference is a model component that is a reference to a state in another behavior. For most modeling aspects it is a normal state.

- Transition ; SRN: 030
  An edge defines the transition between two states. A transition can be traversed if the outcome of the action of the state it originates from satisfies a certain exit condition specified by it's "Alternative
  - CommunicationTransition ; SRN: 031
    A super class for the CommunicationTransitions.
    - ReceiveTransition ; SRN: 032
      Comments have to be added
    - SendTransition ; SRN: 033
      Comments have to be added
  - DoTransition ; SRN: 034
    Comments have to be added
  - SendingFailedTransition ; SRN: 035
    Comments have to be added
  - TimeTransition ; SRN: 036
    Generic super calls for all TimeTransitions, transitions with conditions based on time events. E.g.passing of a certain time duration or the (reoccurring) calendar event.
    - ReminderTransition ; SRN: 037
      Reminder transitions are transitions that can be traverses if a certain time based event or frequency has been reached. E.g. a number of months since the last traversal of this transition or the event of a certain preset calendar date etc.
      - CalendarBasedReminderTransition ; SRN: 038
        A reminder transition, for defining exit conditions measured in calendar years or months
        Conditions are e.g.: reaching of (in model) preset calendar date (e.g. 1st of July) or the reoccurrence of a a long running frequency ("every Month", "2 times a year")"
      - TimeBasedReminderTransition ; SRN: 039
        Comments have to be added
    - TimerTransition ; SRN: 040
      Generic super calls for all TimeTransitions, transitions with conditions based on time events. E.g.passing of a certain time duration or the (reoccurring) calendar event.
      - BusinessDayTimerTransition ; SRN: 041
        imer transitions, denote time outs for the state they originate from. The

condition for a timer transition is that a certain amount of time has passed since the state it originates from has been entered.
The time unit for this timer transition is measured in business days. The definition of a business day depends on a subject's relevant or legal location

    - DayTimeTimerTransition ; SRN: 042
      Timer Transitions, denoting time outs for the state they originate from. The condition for a timer transition is that a certain amount of time has passed since the state it originates from has been entered.
      Day or Time Timers are measured in normal 24 hour days. Following the XML standard for time and day duration. They are to be differed from the timers that are timeout in units of years or months.
    - YearMonthTimerTransition ; SRN: 044
      Timer transitions, denote time outs for the state they originate from. The condition for a timer transition is that a certain amount of time has passed since the state it originates from has been entered.
      Year or Month timers measure time in calendar years or months. The exact definitions for years and months depends on relevant or legal geographical location of the subject.
  - UserCancelTransition ; SRN: 045
    A user cancel transition denotes the possibility to exit a receive state without the reception of a specific message.
    The user cancel allows for an arbitrary decision by a subject carrier/processor to abort a waiting process.
- TransitionCondition ; SRN: 046
  An exit condition belongs to alternatives which in turn is given for a state. An alternative (to leave the state) is only a real alternative if the exit condition is fulfilled (technically: if that according function returns "true")
  Note: Technically and during execution exit conditions belong to states. They define when it is allowed to leave that state. However, in PASS models exit conditions for states are defined and connected to the according transition edges. Therefore transition conditions are individual entities and not DataProperties.
  The according matching must be done by the model execution environment.
  By its existence, an edge/transition defines one possible follow up "state" for its state of origin. It is coupled with an "Exit Condition" that must be fulfilled in the originating state in order to leave the state.
  - DoTransitionCondition ; SRN: 047
    A TransitionCondition for the according DoTransitions and DoStates.
  - MessageExchangeCondition ; SRN: 048
    MessageExchangeConditon is the super class for Send End Receive Transition Conditions the both require either the sending or receiving (exchange) of a message to be fulfilled.
    - ReceiveTransitionCondition ; SRN: 049
      ReceiveTransitionConditions are conditions that state that a certain message must have been taken out of a subjects in-box to be fulfilled.
      These are the typical conditions defined by Receive Transitions.
    - SendTransitionCondition ; SRN: 050
      SendTransitionConditions are conditions that state that a certain message must have been successfully passed to another subjects in-box to be fulfilled.
      These are the typical conditions defined by Send transitions.
  - SendingFailedCondition ; SRN: 051
    Comments have to be added
  - TimeTransitionCondition ; SRN: 052
    A condition that is deemed 'true' and thus the according edge is gone, if: a surrounding execution system has deemed the time since entering the state and starting with the execution of the according action as too long (predefined by the outgoing edge)
    A condition that is true if a certain time defined has passed since the state this condition belongs to has been entered. (This is the standard TimeOut Exit condition)

        - ReminderEventTransitionCondition ; SRN: 053
          Comments have to be added
        - TimerTransitionCondition ; SRN: 054
          Comments have to be added

- DataDescribingComponent ; SRN: 055
  Subject-Oriented PASS Process Models are in general about describing the activities and interaction of active entities. Yet these interactions are rarely done without data that is being generated by activities and transported via messages. While not considered by Börger's PASS interpreter, the community agreed on adding the ability to integrate the means to describe data objects or data structures to the model and enabling their connection to the process model. It may be defined that messages or subject have their individual $DataObjectDefinition$ *in form of a* $SubjectDataDefinition$ *in the case of* $FullySpecifiedSubjects$ *and* $PayloadDataObjectDesfinition$ *in the case of* $MessageSpecifications$ *In general, it expected that these* $DataObjectDefinition$ *list on or more data fields for the message or subject with an internal data type that is described via a* $DataTypeDefinition$*. There is a rudimentary concept for a simple build-in data type definition closely oriented at the concept of ActNConnect. Otherwise, the principle idea of the OWL standard is to allow and employ existing or custom technologies for the serialized definition of data structures (*$CustomOrExternalDataTypeDefinition$*) such as XML-Schemata (XSD), according elements with JSON or directly the powerful expressiveness of OWL itself.*

    - DataObjectDefinition ; SRN: 056
      Data Object Definitions are model elements used to describe that certain other model elements may posses or carrier Data Objects.
      E.G. a message may carrier/include a Business Objects. Or the private Data Space of a Subject may contain several Data Objects.
      A Data Objects should refer to a DataTypeDefinition denoting its DataType and structure.
      DataObject: states that a data item does exist (similar to a variable in programming)DataType: the definition of an Data Object's structure.
        - DataObjectListDefintion ; SRN: 057
          Data definition concept for PASS model build in capabilities of data modeling. Defines a simple list structure.
        - PayloadDataObjectDefinition ; SRN: 058
          Messages may have a description regarding their payload (what is transported with them).
          This can either be a description of a physical (real) object or a description of a (digital) data object
        - SubjectDataDefinition ; SRN: 059
          Comments have to be added
    - DataTypeDefinition ; SRN: 060
      Data Type Definitions are complex descriptions of the supposed structure of Data Objects.
      DataObject: states that a data item does exist (similar to a variable in programming).
      DataType: the definition of an Data Object's structure.
        - CustomOrExternalDataTypeDefinition ; SRN: 061
          Using this class, tool vendors can include their own custom data definitions in the model.
            - JSONDataTypeDefinition ; SRN: 062
              Comments have to be added
            - OWLDataTypeDefinition ; SRN: 63
              Comments have to be added
            - XSD-DataTypeDefinition ; SRN: 064
              XML Schemata Description (XSD) is an established technology for describing structure of Data Objects (XML documents) with many tools available that can verify a document against the standard definition
        - ModelBuiltInDataTypes ; SRN: 065
          Comments have to be added

- PayloadDescription ; SRN: 066
  Comments have to be added
  - PayloadDataObjectDefinition ; SRN: 067
    Messages may have a description regarding their payload (what is transported with them).
    This can either be a description of a physical (real) object or a description of a (digital) data object
  - PayloadPhysicalObjectDescription ; SRN: 068
    Messages may have a description regarding their payload (what is transported with them).
    This can either be a description of a physical (real) object or a description of a (digital) data object

- InteractionDescribingComponent ; SRN: 069
  This class is the super class of all model elements used to define or specify the interaction means within a process model
  - InputPoolConstraint ; SRN: 070
    Subjects do implicitly posses input pools.
    During automatic execution of a PASS model in a work-flow engine this message box is filled with messages.
    Without any constraints models this message in-box is assumed to be able to store an infinite amount of messages.
    For some modeling concepts though it may be of importance to restrict the size of the input pool for certain messages or senders.
    This is done using several different Type of InputPoolConstraints that are attached to a fully specified subject.
    Should a constraint be applicable, an "InputPoolConstraintHandlingStrategy" will be executed by a work-flow engine to determine what to do with the message that does not fit in the pool.
    Limiting the input pool for certain reasons to size 0 together with the InputPoolConstraintStrategy-Blocking is effectively modeling that a communication must happen synchronously instead of the standard asynchronous mode. The sender can send his message only if the receiver is in an according receive state, so the message can be handled directly without being stored in the in-box.
    - MessageSenderTypeConstraint ; SRN: 071
      An InputPool constraint that limits the number of message of a certain type and from a certain sender in the input pool.
      E.g. "Only one order from the same customer" (during happy hour at the bar)
    - MessageTypeConstraint ; SRN: 072
      An InputPool constraint that limits the number of message of a certain type in the input pool.
      E.g. You can accept only "three request at once
    - SenderTypeConstraint ; SRN: 073
      An InputPool constraint that limits the number of message from a certain Sender subject in the input pool.
      E.g. as long as a customer has non non-fulfilled request of any type he may not place messages
  - InputPoolContstraintHandlingStrategy ; SRN: 074
    Should an InputPoolConstraint be applicable, an "InputPoolConstraintHandlingStrategy" will be executed by a work-flow engine to determine what to do with the message that does not fit in the pool.
    There are types of HandlingStrategies.
    InputPoolConstraintStrategy-Blocking - No new message will be adding will need to be repeated until successful
    InputPoolConstraintStrategy-DeleteLatest - The new message will be added, but the last message to arrive before that applicable to the same constraint will be overwritten with the new one. (LIFO deleting concept)
    InputPoolConstraintStrategy-DeleteOldest - The message will be added, but the earliest message in the input pool applicable to the same constraint will be deleted (FIFO deleting concept)

InputPoolConstraintStrategy-Drop - Sending of the message succeeds. However the new message will not be added to the in-box. Rather it will be deleted directly.

- MessageExchange ; SRN: 075
  A message exchange is an element in the interaction description section that specifies exactly one possibility of exchanging messages in the given process context of the model. A message exchange is a triple of, a sender, a receiver, and the specification of the message that may be exchanged.
  While message exchanges are singular occurrences, they may be grouped in MessageExchangeLists
- MessageExchangeList ; SRN: 076
  While MessageExchanges are singular occurrences, they may be grouped in MessageExchangeLists.
  In graphical PASS modeling that is usually the case when one arrow between two subjects contains more than one message and thereby specifies more than one possible message exchange channel between the two subjects.
- MessageSpecification ; SRN: 077
  MessageSpecification are model elements that specify the existence of a message. At minimum its name and id.
  It may contain additional specification for its payload (contained Data, exact form etc.)
- Subject ; SRN: 078
  The subject is the core model element of a subject-oriented PASS process model.
  - FullySpecifiedSubject ; SRN: 079
    Fully specified Subjects in a PASS graph are entities that, in contrast to interface subjects, linked to one ore more Behaviors (they posses a behavior).
  - InterfaceSubject ; SRN: 080
    Interface Subjects are Subjects that are not linked to a behavior. In contrast, they may refer to FullySpecifiedSubjects that are described in other process models.
  - MultiSubject ; SRN: 081
    The Multi-Subject is term for a subject that "has a maximum subject instantiation restriction" within a process context larger than 1.
  - SingleSubject ; SRN: 082
    Single Subject are subject with a maximumInstanceRestriction of 1
  - StartSubject ; SRN: 083
    Subjects that start their behavior with a Do or Send state are active in a process context from the beginning instead of requiring a message from another subject. Usually there should be only one Start subject in a process context.

- PASSProcessModel ; SRN: 084
  The main class that contains all relevant process elements
- SubjectBehavior ; SRN: 085
  Additional to the subject interaction a PASS Model consist of multiple descriptions of subject's behaviors. These are graphs described with the means of $BehaviorDescribingComponents$ *A subject in a model may be linked to more than one behavior.*
  - GuardBehavior ; SRN: 086
    A guard behavior is a special usually additional behavior that guards the Base Behavior of a subject.ßß It starts with a (guard) receive state denoting a special interrupting message. Upon reception of that message the subject will execute the according receive transition and the follow up states until it is either redirected to a state on the base behavior or terminates in an end-state within the guard behavior
  - MacroBehavior ; SRN: 087
    A macro behavior is a specialized behavior that may be entered and exited from a function state in another behavior.
  - SubjectBaseBehavior ; SRN: 088
    The standard behavior model type

- SimplePASSElement ; SRN: 089
  Comments have to be added
  - CommunicationTransition ; SRN: 090
    A super class for the CommunicationTransitions.

  - ReceiveTransition ; SRN: 091
    Comments have to be added
  - SendTransition ; SRN: 092
    Comments have to be added

- DataMappingFunction ; SRN: 093
  Definitions of the ability/need to write or read data to and from a subject's personal data storage.
  DataMappingFunctions are behavior describing components since they define what the subject is supposed to do (mapping and translating data)
  Mapping may be done during reception of message, where data is taken from the message/Business Object (BO) and mapped/put into the local data field.
  It may be done during sending of a message where data is taken from the local vault and put into a BO.
  Or it may occur during executing a do function, where it is used to define read(get) and write (set) functions for the local data.
  - DataMappingIncomingToLocal ; SRN: 094
    A DataMapping that specifies how data is mapped from an an external source (message, function call etc.) to a subject's private defined data space.
  - DataMappingLocalToOutgoing ; SRN: 095
    A DataMapping that specifies how data is mapped from a subject's private data space to an an external destination (message, function call etc.)"
- DoTransition ; SRN: 096
  Comments have to be added
- DoTransitionCondition ; SRN: 097
  A TransitionCondition for the according DoTransitions and DoStates.
- EndState ; SRN: 098
  An end state a behavior. A subject behavior may have one or more end states. Only Do and Receive states may be end states. Send States cannot be end states.
  There are no individual end states that are not Do, Send, or Receive States at the same time.
- FunctionSpecification ; SRN: 099
  A function specification for state denotes
  Concept: Definitions of calls of (mostly technical) functions (e.g. Web-service, Scripts, Database access,) that are not part of the process model.
  Function Specifications are more than "Data Properties"? –> - If special function types (e.g. Defaults) are supposed to be reused, having them as explicit entities is a the better OWL-modeling choice.
  - CommunicationAct ; SRN: 100
    A super class for specialized FunctionSpecification of communication acts (send and receive)
    - ReceiveFunction ; SRN: 101
      Specifications/descriptions for Receive-Functions describe in detail what the subject carrier is supposed to do in a state.
      DefaultFunctionReceive1_EnvoironmentChoice : present the surrounding execution environment with the given exit choices/conditions currently available depending on the current state of the subjects in-box. Waiting and not executing the receive action is an option.
      DefaultFunctionReceive2_AutoReceiveEarliest: automatically execute the according activity with the highest priority as soon as possible. In contrast to DefaultFunctionReceive1, it is not an option to prolong the reception and wait e.g. for another message.
    - SendFunction ; SRN: 102
      Comments have to be added
  - DoFunction ; SRN: 103
    Specifications or descriptions for Do-Functions describe in detail what the subject carrier is supposed to do in an according state.
    The default DoFunction 1: present the surrounding execution environment with the given exit choices/conditions and receive choice of one exit option –> define its Condition to be fulfilled in order to go to the next according state.

The default DoFunction 2: execute automatic rule evaluation (see DoTransitionCondition).
More specialized Do-Function Specifications may contain Data mappings denoting what of a subjects internal local Data can and should be:
a) read: in order to simply see it or in order to send it of to an external function (e.g. a web service)
b) write: in order to write incoming Data from e.g. a web Service or user input, to the local data fault

- InitialStateOfBehavior ; SRN: 104
  The initial state of a behavior
- MessageExchange ; SRN: 105
  A message exchange is an element in the interaction description section that specifies exactly one possibility of exchanging messages in the given process context of the model.
  A message exchange is a triple of, a sender, a receiver, and the specification of the message that may be exchanged.
  While message exchanges are singular occurrences, they may be grouped in MessageExchangeLists
- MessageExchangeCondition ; SRN: 106
  MessageExchangeConditon is the super class for Send End Receive Transition Conditions the both require either the sending or receiving (exchange) of a message to be fulfilled.
  - ReceiveTransitionCondition ; SRN: 107
    ReceiveTransitionConditions are conditions that state that a certain message must have been taken out of a subjects in-box to be fulfilled.
    These are the typical conditions defined by Receive Transitions.
  - SendTransitionCondition ; SRN: 108
    SendTransitionConditions are conditions that state that a certain message must have been successfully passed to another subjects in-box to be fulfilled.
    These are the typical conditions defined by Send transitions.
- MessageExchangeList ; SRN: 109
  While MessageExchanges are singular occurrences, they may be grouped in MessageExchangeLists.
  In graphical PASS modeling that is usually the case when one arrow between two subjects contains more than one message and thereby specifies more than one possible message exchange channel between the two subjects.
- MessageSpecification ; SRN: 110
  MessageSpecification are model elements that specify the existence of a message. At minimum its name and id.
  It may contain additional specification for its payload (contained Data, exact form etc.)
- ModelBuiltInDataTypes ; SRN: 111
  Comments have to be added
- PayloadDataObjectDefinition ; SRN: 112
  Messages may have a description regarding their payload (what is transported with them). This can either be a description of a physical (real) object or a description of a (digital) data object
- StandardPASSState ; SRN: 113
  A super class to the standard PASS states: Do, Receive and Send
  - DoState ; SRN: 114
    The standard state in a PASS subject behavior diagram denoting an action or activity of the subject in itself.
  - ReceiveState ; SRN: 115
    The standard state in a PASS subject behavior diagram denoting an receive action or rather the waiting for a receive possibility.
  - SendState ; SRN: 116
    The standard state in a PASS subject behavior diagram denoting a send action
- Subject ; SRN: 117
  The subject is the core model element of a subject-oriented PASS process model.

- FullySpecifiedSubject ; SRN: 118
  Fully specified Subjects in a PASS graph are entities that, in contrast to interface subjects, linked to one ore more Behaviors (they posses a behavior).
- InterfaceSubject ; SRN: 119
  Interface Subjects are Subjects that are not linked to a behavior. In contrast, they may refer to FullySpecifiedSubjects that are described in other process models.
- MultiSubject ; SRN: 120
  The Multi-Subject is term for a subject that "has a maximum subject instantiation restriction" within a process context larger than 1.
- SingleSubject ; SRN: 121
  Single Subject are subject with a maximumInstanceRestriction of 1
- StartSubject ; SRN: 122
  Subjects that start their behavior with a Do or Send state are active in a process context from the beginning instead of requiring a message from another subject.
  Usually there should be only one Start subject in a process context.

- SubjectBaseBehavior ; SRN: 123
  The standard behavior model type

A.2 Object Properties (42)

Property name		Domain-Range	Comments	Reference
belongsTo	Domain: Range:	PASSProcessModelElement PASSProcessModelElement	Generic ObjectProperty that links two process elements, where one is contained in the other (inverse of contains).	200
contains	Domain: Range:	PASSProcessModelElement PASSProcessModelElement	Generic ObjectProperty that links two model elements where one contains another (possible multiple)	201
containsBaseBehavior	Domain: Range:	Subject SubjectBehavior		202
containsBehavior	Domain: Range:	Subject SubjectBehavior		203
containsPayload-Description	Domain: Range:	MessageSpecification PayloadDescription		204
guardedBy	Domain: Range:	State, Action GuardBehavior		205
guardsBehavior	Domain: Range:	GuardBehavior SubjectBehavior	Links a GuardBehavior to another SubjectBehavior. Automatically all individual states in the guarded behavior are guarded by the guard behavior. There is an SWRL Rule in the ontology for that purpose.	206
guardsState	Domain: Range:	State, Action guardedBy		207
hasAdditionalAttribute	Domain: Range:	PASSProcessModelElement AdditionalAttribute		208
hasCorrespondent	Domain: Range:	 Subject	Generic super class for the ObjectProperties that link a Subject with a MessageExchange either in the role of Sender or Receiver.	209

Property name		Domain-Range	Comments	Reference
hasDataDefinition	Domain: Range:	 DataObjectDefinition		210
hasDataMapping-Function	Domain: Range:	state, SendTransition, ReceiveTransition DataMappingFunction		211
hasDataType	Domain: Range:	PayloadDescription or DataObject-Definition DataTypeDefinition		212
hasEndState	Domain: Range:	SubjectBehavior or ChoiceSegmentPath State, not SendState		213
hasFunction-Specification	Domain: Range:	State FunctionSpecification		214
hasHandlingStrategy	Domain: Range:	InputPoolConstraint InputPoolContstraint-HandlingStrategy		215
hasIncomingMessage-Exchange	Domain: Range:	Subject MessageExchange		216
hasIncomingTransition	Domain: Range:	State Transition		217
hasInitialState	Domain: Range:	SubjectBehavior or ChoiceSegmentPath State		218
hasInputPoolConstraint	Domain: Range:	Subject InputPoolConstraint		219
hasKeyValuePair	Domain: Range:			220
hasMessageExchange	Domain: Range:	Subject	Generic super class for the Object-Properties linking a subject with either incoming or outgoing MessageExchanges.	221

Property name		Domain-Range	Comments	Reference
hasMessageType	Domain: Range:	MessageTypeConstraint or MessageSenderTypeConstraint or MessageExchange MessageSpecification		222
hasOutgoingMessage-Exchange	Domain: Range:	Subject MessageExchange		223
hasOutgoingTransition	Domain: Range:	State Transition		224
hasReceiver	Domain: Range:	MessageExchange Subject		225
hasRelationToModel-Component	Domain: Range:	PASSProcessModelElement PASSProcessModelElement	Generic super class of all object properties in the standard-pass-ont that are used to link model elements with one-another.	226
hasSender	Domain: Range:	MessageExchange Subject		227
hasSourceState	Domain: Range:	Transition State		228
hasStartSubject	Domain: Range:	PASSProcessModel StartSubject		229
hasTargetState	Domain Range	Transition State		230
hasTransitionCondition	Domain Range	Transition TransitionCondition		231
isBaseBehaviorOf	Domain: Range:	SubjectBaseBehavior	A specialized version of the "belongsTo" ObjectProperty to denote that a -SubjectBehavior belongs to a Subject as its BaseBehavior	232

Property name		Domain-Range	Comments	Reference
isEndStateOf	Domain: Range:	State and not SendState SubjectBehavior or ChoiceSegmentPath		233
isInitialStateOf	Domain: Range:	State SubjectBehavior or ChoiceSegmentPath		234
isReferencedBy	Domain: Range:			235
references	Domain: Range:			236
referencesMacroBehavior	Domain: Range:	MacroState MacroBehavior		237
refersTo	Domain: Range:	CommunicationTransition MessageExchange	Communication transitions (send and receive) should refer to a message exchange that is defined on the interaction layer of a model.	238
requiresActiveReception-OfMessage	Domain: Range:	ReceiveTransitionCondition MessageSpecification		239
requiresPerformed-MessageExchange	Domain: Range:	MessageExchangeCondition MessageExchange		240
SimplePASSObject-Propertie	Domain: Range:		Every element/sub-class of SimplePASSObjectProperties is also a Child of PASSModelObjectPropertiy. This is simply a surrogate class to group all simple elements together	241

A.3 Data Properties (27)

Property name		Domain-Range	Comments	Reference
hasBusinessDayDurationTimeOutTime	Domain: Range:			
hasCalendarBasedFrequencyOrDate	Domain: Range:			
hasDataMappingString	Domain: Range:			
hasDayTimeDurationTimeOutTime	Domain: Range:			
hasDurationTimeOutTime	Domain: Range:			
hasFeelExpressionAsDataMapping	Domain: Range:		See `https://www.omg.org/spec/DMN` for specification of Feel-Statement-Strings The idea of these expression is to map data fields from and to the internal Data storage of a subject	
hasGraphicalRepresentation	Domain: Range:		The process models are in principle abstract graph structures. Yet the visualization of process models is very important since many process models are initially created in a graphical form using a graph editor that was created to foster human comprehensibility. If available any process element may have a graphical representation attached to it	

Property name		Domain-Range	Comments	Reference
hasKey	Domain: Range:			
hasLimit	Domain: Range:			
hasMaximumSubjectInstanceRestriction	Domain: Range:			
hasMetaData	Domain: Range:			
hasModelComponentComment	Domain: Range:		equivalent to rdfs:comment	
hasModelComponentID	Domain: Range:		The unique ID of a PASSProcessModelComponent	
hasModelComponentLabel	Domain: Range:		The human legible label or description of a model element.	

Property name		Domain-Range	Comments	Reference
hasPriorityNumber	Domain: Range:		Transitions or Behaviors have numbers that denote their execution priority in situations where two or more options could be executed. This is important for automated execution. E.g. when two messages are in the in-box and could be followed, the message denoted on the transition with the higher priority (lower priority number) is taken out and processed. Similarly, SubjectBehaviors with higher priority (lower priority number) are to be executed before Behaviors with lower priority.	

Property name		Domain-Range	Comments	Reference
hasReoccuranceFrequenyOrDate	Domain: Range:		A data field meant for the two classes Reoccurance-TimeOutTransition and ReoccuranceTimeOutExit-Condition. ToDo: Define the according data format for describing the iteration frequencies or reoccurring dates. Opinion: rather complex: expressive capabilities should cover expressions like: "every 2nd Monday of Month at 7:30 in Morning." Every 29th of July" or "Every Hour", "ever 25 Minuets" , "once each day", "twice each week" etc	
hasSVGRepresentation	Domain: Range:		The Scalable Vector Graphic (SVG) XML format is a text based standard to describe vector graphics. Adding according image information as XML literals is therefor a suitable, yet not necessarily easily changeable option to include the graphical representation of model elements in the an OWL file.	
hasTimeBasedReoccuranceFrequencyOrDate	Domain: Range:			

Property name		Domain-Range	Comments	Reference
hasTimeValue	Domain: Range:		Generic super class for all data properties of time based transitions.	
hasToolSpecificDefinition	Domain: Range:		This is a placeholder DataProperty meant as a tie in point for tool vendors to include tool specific data values/properties into models. By denoting their own data properties as sub-classes to this one the according data fields can easily be recognized as such. However, this is only an option and a place holder to remind that something like this is possible.	
hasValue	Domain: Range:			
hasYearMonthDurationTimeOutTime	Domain: Range:			
isOptionalToEndChoiceSegmentPath	Domain: Range:			
isOptionalToStartChoiceSegmentPath	Domain: Range:			
owl:topDataProperty	Domain: Range:			
PASSModelDataProperty	Domain: Range:		Generic super class of all DataProperties that PASS process model elements may have.	

Property name		Domain-Range	Comments	Reference
SimplePASSDataProperties	Domain: Range:		Every element/sub-class of SimplePASSDataProperties is also a Child of PASS-ModelDataPropertiy. This is simply a surrogate class to group all simple elements together	

APPENDIX **B**

An ASM Interpreter Model for PASS

B.1 Subject Behavior Diagram Interpretation

```
Behavior_subj(D) = {Behavior(subj ; node) | node  Node(D)}

Behavior(subj ; state) =
if SID state(subj ) = state then
  if Completed(subj ; service(state); state) then
    let edge =
        selectEdge ({e ϵ 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)
```

Figure B.1: Top Level of Interpreter Model

B.2 ALTERNATIVE SEND/RECEIVE ROUND INTERPRETATION

```
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
          INTERRUPT_ComAct(subj ; state)
    elseif UserAbruption(subj ; state)
       then Abrupt_ComAct(subj ; state)
       else TRYALTERNATIVE_ComAct(subj ; state)
```

Figure B.2: Alternative Send/Receive Round Interpretation

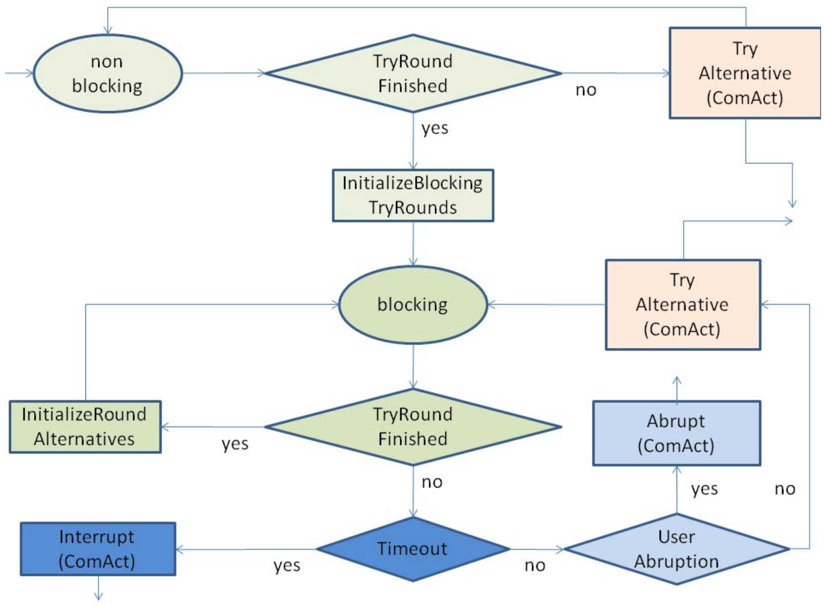


Figure B.3: Diagram of Alternative Send/Receive Round Interpretation

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*

Figure B.4: Start function

[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 >= blockingStartTime(subj ; state) + time

Figure B.5: Try non/blocking Round

B.3 MsgElaboration Interpretation for Multi Send/Receive

B.4 Multi Send/Receive Round Interpretation

B.5 Actual Send Interpretation

B.6 ACTUAL RECEIVE INTERPREATION

B.7 Alternative Action Interpretation

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)*

Figure B.6: Interupt Handling

Figure B.7: Interpretation for Multi Send/Receive

Figure B.8: State Diagram Multi Send

Figure B.9: State Diagram Multi Receive

Figure B.10: ASM Interpreter for Multi Send/Receive

Figure B.11: Interpretation for asynchronous send

Figure B.12: Interpretation for synchronous send

Figure B.13: Interpretation for asynchronous receive

Figure B.14: Interpretation for synchronous receive

Figure B.15: Alternative Actions

Figure B.16: Start and Perform Alternative Actions

Figure B.17: Auxiliary Wait/Exit Rule Interpretation

B.8 Interrupt Behavior

Figure B.18: Interrupt Behaviour (Guard)

APPENDIX C

Mapping Ontology to Abstract State Machine

The following tables show the relationships between the PASS ontology and the PASS execution semantics described as ASMs. Because of historical reasons in the ASMs names for entities and relations are different from the names used in the ontology. The tables below show the mapping of the entitiy and relation names in the ontology to the names used in the ASMs.

C.1 Mapping of ASM Places to OWL Entities

Places are formally also functions or rules, but are used in principle as passive/static storage places.

OWL Model element	ASM interpreter	Description
X - Execution concept – the state the subject is currently in as defined by a **State** in the model	*SID_state*	Execution concept – no model representation, Not to be confused by a model "state" in an SBD Diagram. State in the SBD diagram define possible SID_States.
SubjectBehavior – under the assumption that it is complete and sound.	*D*	A Diagram that is a completely connected SBD
State	*node*	A specific element of diagram D - Every node 1:1 to state
State	*state*	The current active state of a diagram determined by the nodes of Diagram D
InitialStateOfBehavior, EndState	*initial state, end state*	The interpreter expects and SBD Graph D to contain exactly one initial (start) state and at least one end state.
Transition	*edge / outEdge*	"Passive Element" of an edge in an SBD-graph
TransitionConditionn	*ExitCondition*	Static Concept that represents a Data condition
Execution Concept – ID of a Subject Carrier responsible possible multiple Instances of according to specific **SubjectBehavior**	*subj*	Identifier for a specific Subject Carrier that may be responsible for multiple Subjects
Represented in the model with **InterfaceSubject**	*ExternalSubject*	A representation of a service execution entity outside of the boundaries of the interpreter (The PASS-OWL Standardization community decided on the new Term of Interface Subject to replace the often-misleading older term of External Subject)
SubjectBehavior or rather **SubjectBaseBehavior** as MacroBehaviors and GuardBehaviors are not covered by Börger	*subject-SBD* / $SBDsubject_{subject}$	Names for completely connected graphs / diagrams representing SBDs
Object Property: ***hasFunctionSpecification*** (linking **State**, and **FunctionSpecification** –>(**State** ***hasFunctionSpecification*** **FunctionSpecification**)	*service(state)* / *service(node)*	Rule/Function that reads/returns the service of function of a given state/node

OWL Model element	ASM interpreter	Description
DoState **SendState** **ReceiveState**	*function state,* *send state,* *receive state*	The ASM spec does not itself contain these terms. The description text, however, uses them to describe states with an according service (e.g. a state in which a (ComAct = Send) service is executed is referred to as a send state) Seen from the other side: a SendState is a state with service(state) = Send) Both send and receive services are a ComAct service. The ComAct service is used to define common rules of these communication services.
CommunicationActs with sub-classes (**ReceiveFunction SendFunction**) ***DefaultFunctionReceive1_EnvironmentChoice*** ***DefaultFunctionReceive2_AutoReceiveEarliest*** ***DefaultFunctionSend***	*ComAct*	Specialized version of Perform-ASM Rule for communication, either send or receive. These rules distinguish internally between send and receive.

C.2 Main Execution/Interpreting Rules

The interpreter ASM Spec has main-function or rules that are being executed while interpreted.

- BEHAVIOR(subj,state)
- PROCEED(subj,service(state),state)
- PERFORM(subj,service(state),state)
- START (subj,X, node)

These make up the main interpreter algorithm for PASS SBDs and therefore have no corresponding model elements but rather are or contain the instructions of how to interpret a model.

OWL Model element	ASM interpreter	Description
Execution concept	*BEHAVIOR(subj;state)*	Main interpreter ASM-rule/Method
Execution concept	*BEHAVIOR(subj;node)*	ASM-Rule to interpret a specific node of Diagram D for a specific subject
Execution concept	**Behaviorsubj (D)**	Set of all ASM rules to interprete all nodes/states in a SBD(iagram) D for a given subj (set of all *BEHAVIOR(subj;node)*)
State hasFunctionSpecification FunctionSpecification Specialized in: **DoFunction** and **CommunicationActs** with **ReceiveFunction** **SendFunction** There exist a few default activities: **DefaultFunctionDo1_EnvoironmentChoice** **DefaultFunctionDo2_AutomaticEvaluation**	*PERFORM(subj ; service(state); state)*	Main interpreter ASM-rule/Method
CommunicationActs with **ReceiveFunction** **SendFunction** **DefaultFunctionReceive1_EnvironmentChoice** **DefaultFunctionReceive2_AutoReceiveEarliest** **DefaultFunctionSend**	*PERFORM(subj ;ComAct; state)*	ASM-Rule specifying the execution of a Comunication act in an according state)

Table C.2: Main Execution/Interpreting Rules

C.3 FUNCTIONS

Functions return some element. They are activities that can be performed to determine something. Dynamic functions can be considered as “variables” known from programming languages, they can be read and written. Static functions are initialized before the execution, they can only be read. Derived functions "evaluate” other functions, they can only be read. “They may be thought of as a global method with read-only variables”

OWL Model element	ASM interpreter	Description
Function that the return state should correspond to/be derived from one of the multiple **State** in an SBD model	*SID_state(subj)*	Dynamic ASM-Function that stores the current state of a subj
State **hasOutgoingTransition** **Transition** (input / worked on link / output (Set of Transition) (linking State with)	*OutEdge(state)* *OutEdge(state;i)*	Function that returns the set of outgoing edges of a state or a single specific edge i
Object Property: **hasTargetState** (linking **Transition** and **State** —> **Transition** **hasTargetState** **State**	*target(edge)/* *target(outEdge) /*	Function that returns the follow up state of an outgoing transition (*outEdge* is a special denomination for an *edge*returned by the *outEdge*-Function)
Object Property: **hasSourceState** (linking **Transition** and **State**–> **Transition** **hasSourceState** **State** (input / worked on link / output)	*source(edge)*	Function that returns the source state of an edge
Determine Follow up state Mechanic		
Exit conditions in PASS are defined on their corresponding **Transitions** and therefore are called **TransitionCondition** **Transitions** have (**hasTransitionCondition**) (**State** → **hasOutgoingTransition** → **Transition** → **hasTransitionCondition** → **TransitionCondition**)	*ExitCond(e)* *ExitCond(outEdge)* *ExitCond_i(e)* *ExitCond(e)(subj,state)*	Derived Function that evaluates the ExitCondition of a given edge/outgoing edge
Execution Concept	$\text{select}_{\text{Edge}}$	ASM Function that determines an edged (transition) to follow.
Execution Concept (connected to: **State**, and **FunctionSpecification**)	*completed(subj ; service(state); state)*	Function that returns true if the Service of a certain state is complete IF the subject is in that state
Execution Concept		Rule/Function that gives that returns the service of function of a given state

Table C.3: Derived Functions

C.4 Extended Concepts – Refinements for the Semantics of Core Actions

OWL Model element	ASM interpreter	Description
Function that the return state should correspond to/be derived from one of the multiple **State** in an SBD model	*SID_state(subj)*	Dynamic ASM-Function that stores the current state of a subj
State **hasOutgoingTransition** **Transition** (input / worked on link / output (Set of Transition) (linking State with)	*OutEdge(state)* *OutEdge(state;i)*	Function that returns the set of outgoing edges of a state or a single specific edge i

Table C.4: Refinrments places

C.5 INPUT POOL HANDLING

OWL Model element	ASM interpreter	Description
Refers to a set of **InputPoolConstraints** of **Subject** that has **hasInputPoolConstraints** – for its Input Pool	*constraintTable(inputPool)*	Function that Returns the set of all input Pool constrains
Execution Concept with evalution relevance for: **MessageSenderTypeConstraint** and **SenderTypeConstraint**	*sender/receiver*	Identifiers for possible subject instances trying to access an input pool
Refers to a set of **InputPoolConstraints** of **Subject** that has **hasInputPoolConstraints** – for its Input Pool	*msgType)*	Function that Returns the set of all input Pool constrains
Execution Concept	*select* $_{MsgKind(subj\,;state;alt;i)}$	ASM Function that determines the message kind ("message type") to be received in a given receive state.
InputPoolContstraintHandlingStrategy And their individual defaultinstances: **InputPoolConstraintStrategy-Blocking** **InputPoolConstraintStrategy-DeleteLatest** **InputPoolConstraintStrategy-DeleteOldest** **InputPoolConstraintStrategy-Drop**	*/Blocking; DropYoungest; DropOldest; DropIncoming/*	Default Input Pool handling strategies for
Execution Concept – can be restricted by **InputPoolConstraint** – for its Input Pool	*P / inputPool*	The actual Input Pool
synchronous communication	Definition for an input pool **constraint set to 0** requiring sender and receiver interpreter to be in the corresponding send and receive states at the same time in order to actually communicate (as messages cannot be passed to an input pool)	

Table C.5: Input Pool Handling

C.6 Other Functions

OWL Model element	ASM interpreter	Description
Exit conditions in PASS are defined on their corresponding **Transitions** and therefore are called **TransitionCondition**. Execution Concept: can be set on. Execution Concept: used to determine the correct exit	*NormalExitCond*	is used internally to "remember" that neither a timeout nor a user cancel have happened, so that the correct exit transition can be taken.
In the model to be interpreted the according aspects are captured by **TimerTransitions** that have (**hasTransitionCondition**) a **TimerTransitionCondition** containing the date. The timeout(state) function should read the information.	**Timer/Timeout Mechanic:** The evaluation and handling of timeouts is defined (and refined) with several rules and functions.*OutEdge(timeout(state), Timeout(subj , state, timeout(state)), SetTimeoutClock(subj ; state)* are used to evaluate the timeout condition, *OutEdge(Interrupt_service(state)(subj , state)* is used to define how the corresponding service should be canceled. *OutEdge(TimeoutExitCond)* is used internally to "remember" that a timeout happened, so that the correct exit transition can be taken.	
In PASS models the possibility to arbitrarily cancel the execution of a (receive) function and the possible course of action afterwards may be discerned via a **UserCancelTransitions**	**User Cancel/Abrupt Mechanic:** The evaluation and handling of user cancels is defined (and refined) with several rules and functions. *UserAbruption(subj, state)* is used to evaluate the user decision, *Abrupt_service(state)(subj , state)* is used to define how the corresponding service should be abrupted. *AbruptionExitCond* is used internally to "remember" that a user cancel happened, so that the correct exit transition can be taken.	
With the definition of the data properties **hasMaximumSubjectInstanceRestriction** The MultiSubject are actually the standard and **SingleSubject** the special case	*MultiRound / mult(alt) / InitializeMultiRoun / ContinueMultiRoundSuccess (among others*	Definition of Functions and ASM rules for interaction between multiple Subjects at once
Handling of **ChoiceSegment** & **ChoiceSegmentPath** **hasOutgoingTransition** **Transition** (input / worked on link / output (Set of Transition) (linking State with)	*AltAction / altEntry(D) / altExit(D) AltBehDgm(altSplit) altJoin(altSplit)*	Rules for the semantics/handling of ChoiceSegements
State **hasOutgoingTransition** **Transition** (input / worked on link / output (Set of Transition) (linking State with)	*Compulsory(altEntry(D))* and textit-Compulsory(altExit(D))	

Table C.6: Other Functions

C.7 Elements Not Covered not by Börger (directly)

OWL Model element	Description
ReminderTransition / ReminderEventTransitionCondition	This type time-logic-based transitions did not exist when the original ASM interpreter was conceived. They were added to PASS for the OWL Standard. They can be handled by assuming the existence of an implicit calendar subject that sends an interrupt message (reminder) upon a time condition (e.g. reaching of a calendarial date) has been achieved. (includes the specialized (**CalendarBasedReminderTransition, TimeBasedReminderTransition**
DataDescribingComponent / DataMappingFunction	The PASS OWL standard envisions the integration and usage of classic data element (Data Objects) as part of a process model. The Börger Interpreter does not assume the existence of such data elements as part of the model. However, the refinement concept of ASMs could easily been used to integrate according interpretation aspects. (Includes Elements such as **PayloadDescription** for Messages or **DataMappingFunction**

Table C.7: Other Functions

APPENDIX D

PASS ASM Specification with detailed Comments

In the following you can find a detailed description of the ASM semantic of PASS.

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.

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

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

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 AsmBook [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].

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

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 $\mathrm{SBD}_{subject}$ to indicate that it is an SBD with this underlying *subject*.

Copyright: 2011 Carl Hanser Verlag München

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

$outEdge \in OutEdge(node)$ 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 states* or simply *states*. 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),

Copyright: 2011 Carl Hanser Verlag München

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

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).

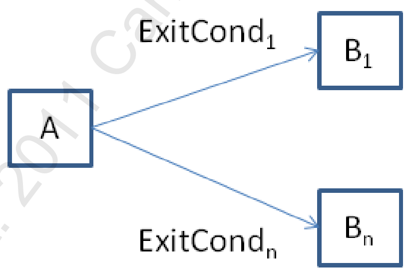


Fig. 1. SID-transition graph structure

Copyright: 2011 Carl Hanser Verlag München

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

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

$outEdge(state, i)$ 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($subj, node$) | $node \in Node(D)$}

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 runs*) 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

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

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

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.

Copyright: 2011 Carl Hanser Verlag München

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

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

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

$ConstraintViolation(msg, row)$ iff [5]
$\quad Match(msg, row) \wedge size(\{m \in P \mid Match(m, row)\}) + 1 \not\leq n$
where
$\quad Match(m, row)$ iff
$\quad\quad (sender(m) = s$ **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.

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

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

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]

$$\begin{aligned} &\mathit{CanAccess}(\mathit{sender}, P) \text{ if and only if} \\ &\quad \mathit{sender} = \mathit{select}_P(\{\mathit{subject} \mid \mathit{TryingToAccess}(\mathit{subject}, P)\}) \end{aligned}$$

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 *Alternatives*,[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,
- $\text{TRYALTERNATIVE}_{ComAct}$, 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 $\text{CHOOSE\&PREPAREALTERNATIVE}_{ComAct}$ of $\text{TRYALTERNATIVE}_{ComAct}$.

[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 *Alternatives* 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 *Alternatives* 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.

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

If the 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$(subj, \text{COMACT}, state)$. The flowchart in Fig. 2 visualizes the structure of PERFORM$(subj, \text{COMACT}, state)$.[9] The symmetry between non-blocking and

Copyright: 2011 Carl Hanser Verlag München

[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

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

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.

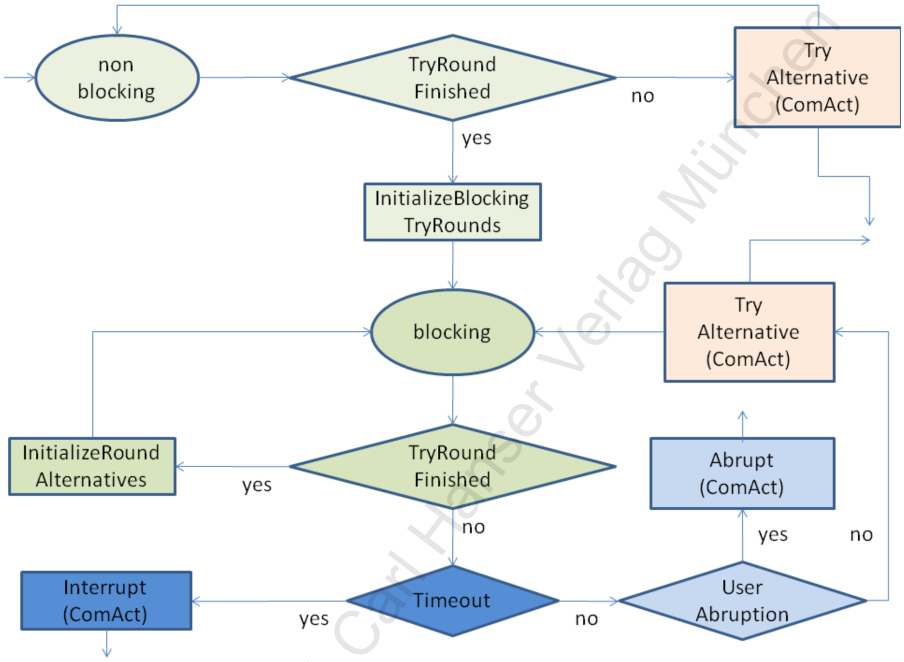


Fig. 2. PERFORM($subj$, COMACT, $state$)

Copyright: 2011 Carl Hanser Verlag München

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.

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

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

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

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

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

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

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

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

To conclude this section: an attempt to $\textsc{TryAlternative}_{ComAct}$ 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 $\textsc{TryAlternative}_{ComAct}$ in [*non*]*blocking tryMode* is to $\textsc{Choose\&PrepareAlternative}_{ComAct}$. Then it will $\textsc{Try}_{ComAct}$ for the prepared message(s).[10]

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

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

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 $\textsc{EnterTryAlternative}_{ComAct}$ update because the machine is used only as composed by **seq** (with $\textsc{Try}_{ComAct}$ in $\textsc{TryAlternative}_{ComAct}$).

Copyright 2011 Carl Hanser Verlag München

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

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

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

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 $\text{TryAlternative}_{ComAct}$. As before the *ComAct* parameter stands for *Send* or *Receive*.

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

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 $\text{PrepareMsg}_{Receive}$ may select $mult(alt)$ elements from a set of $ExpectedMsgKind(alt)$ using some choice function $select_{MsgKind}$.[13]

$\text{PrepareMsg}_{ComAct}(subj, state, alt) =$
  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*) *alternative* $mult(alt)$ 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.

Copyright: 2011 Carl Hanser Verlag München

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

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.

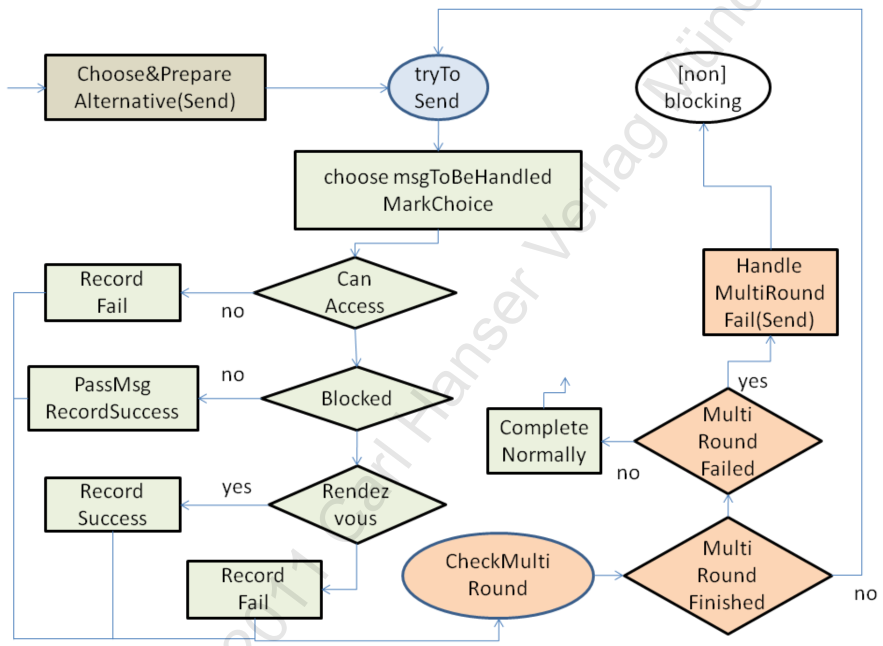


Fig. 3. TRYALTERNATIVE$_{Send}$

Copyright: 2011 Carl Hanser Verlag München

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 *AlternativeRounds*. We include into it a record of the current choice because this information is needed to describe the Rendezvous predicate for synchronous communication.

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

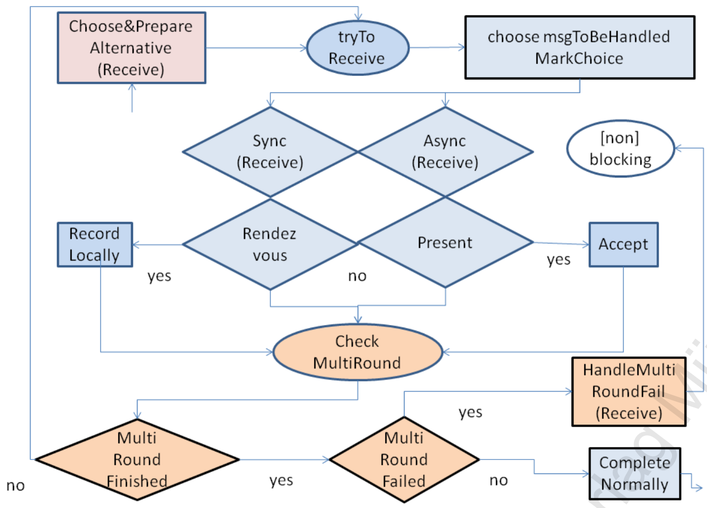


Fig. 4. TRYALTERNATIVE$_{Receive}$

- 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*.

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

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.

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

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

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

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*.

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

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).

$$
\begin{array}{l}
\textsc{Sync}(\mathit{Send})(\mathit{subj}, \mathit{state}, \mathit{msg}) = \\
\quad \textsc{ContinueMultiRound}_{\mathit{Success}}(\mathit{subj}, \mathit{state}, \mathit{msg}) \\
\mathit{PossibleSync}_{\mathit{Send}}(\mathit{subj}, \mathit{msg}) \text{ iff } \mathit{RendezvousWithReceiver}(\mathit{subj}, \mathit{msg})
\end{array}
$$

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.

Copyright 2011 Carl Hanser Verlag München

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

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

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

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

3.4 How to Actually Receive a Message

In this section we define the two ASYNCH(*Receive*) and SYNC(*Receive*) components which asynchronously or synchronously carry out the actual *Receive* action and 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
  $s \in Sender \cup \{any\}$ **and** $t \in MsgType \cup \{any\}$
  $commMode \in \{async, sync\}$ // accepted communication mode

[20] This MultiRound location is updated in MARKCHOICE.

Copyright 2011 Carl Hanser Verlag München

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

The communication mode decides upon whether the receiver will try to $\textsc{Async}(Receive)$ or to $\textsc{Sync}(Receive)$ a message of a chosen expected message kind.

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

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

When $\textsc{Sync}(Receive)(subj, state)$ is called, the receiver knows that there is a *sender* for a *RendezvousWithSender* (a subject which right now via a $\textsc{Try}_{Send}$ 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 $\textsc{RecordLocally}$ the *msgToBeSent*, bypassing the input pool,[21] and $\textsc{ContinueMultiRound}_{Success}(subj, state, currMsgKind(subj, state))$.

$\textsc{Sync}(Receive)(subj, state, msgKind) =$
  let $P = inputPool(subj)$, $sender = \iota s(CanAccess(s, P))$
    $\textsc{RecordLocally}(subj, msgToBeSent(sender, SID\_state(sender)))$
    $\textsc{ContinueMultiRound}_{Success}(subj, state, msgKind)$
$Sync(Receive)(msgKind)$ 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 $\textsc{Try}_{Send}$ in Sect. 3.3.

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

$$\begin{aligned}
&\mathit{RendezvousWithSender}(\mathit{subj}, \mathit{msgKind}) \text{ iff}\\
&\quad \mathit{Sync}(\mathit{Receive})(\mathit{msgKind}) \textbf{ and}\\
&\qquad \textbf{let } \mathit{sender} = \iota s(\mathit{CanAccess}(s, \mathit{inputPool}(\mathit{subj})))\\
&\qquad \textbf{let } \mathit{msgToBeSent} = \mathit{msgToBeSent}(\mathit{sender}, \mathit{SID_state}(\mathit{sender}))\\
&\qquad\quad \mathit{tryMode}(\mathit{sender}) = \mathit{tryToSend} \textbf{ and } \mathit{SyncSend}(\mathit{msgToBeSent})\\
&\qquad\qquad \textbf{and } \mathit{Match}(\mathit{msgToBeSent}, \mathit{msgKind})
\end{aligned}$$

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

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

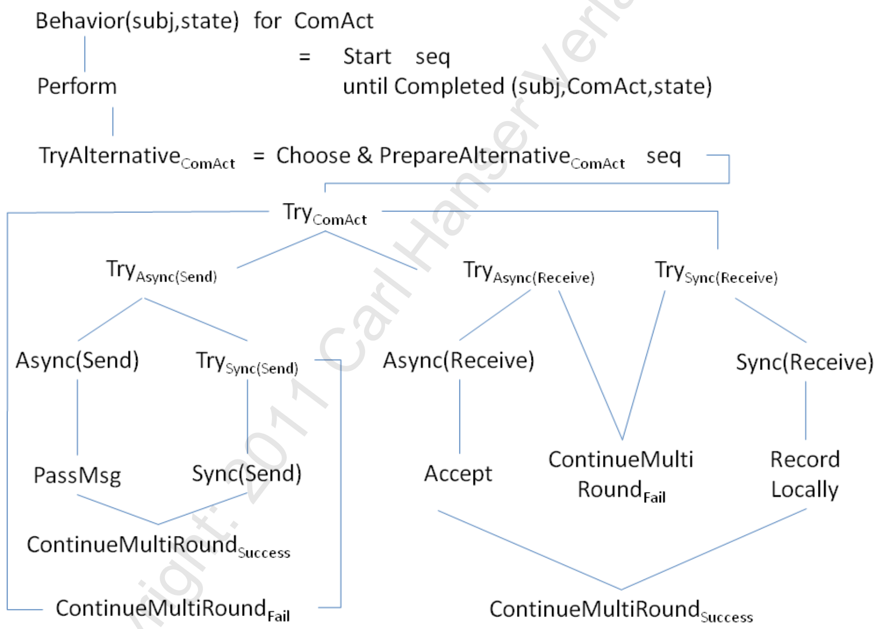


Fig. 5. Subcomponent Structure for Communication Behavior

3.5 Internal Functions

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

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

completion predicate *Completed* for the parameters $(subj, A, state)$ in the same manner as has been done for communication actions in Sect. 3.3-3.4—but only once it is known how to start, to perform and to complete A. For example, for Java coded functions A $\textsc{Start}(subj, A, state)$ could mean to call the (multi-threaded) Java interpreter *execJavaThread* defined in terms of ASMs in [4, p.101], $\textsc{Perform}(subj, A, state)$ 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 $\textsc{Behavior}(state)$ 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**
  $\textsc{Interrupt}_{service(state)}(subj, state)$
elseif $UserAbruption(subj, state)$
  then $\textsc{Abrupt}_{service(state)}(subj, state)$
  else $\textsc{Perform}(subj, state)$

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.

Copyright: 2011 Carl Hanser Verlag München

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

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

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

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

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

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

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

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

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

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

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

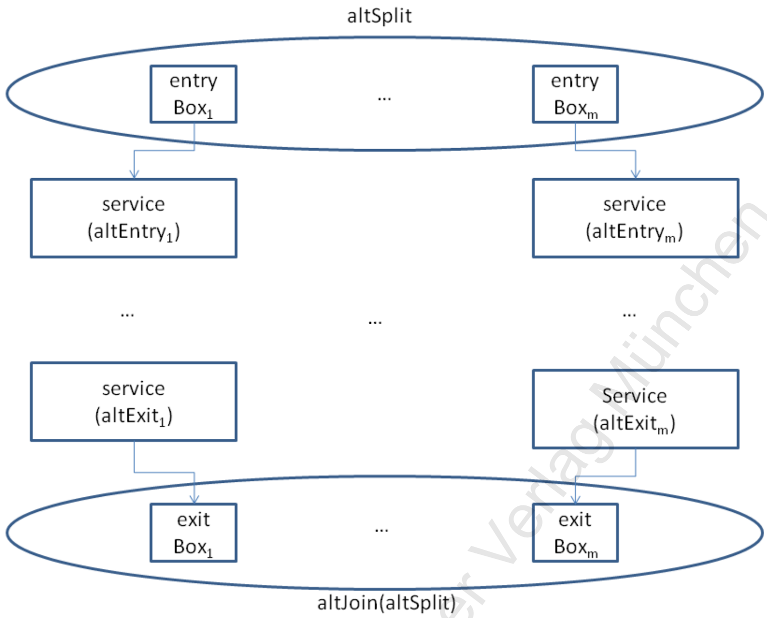


Fig. 6. Structure of Alternative Action Nodes

Copyright: 2011 Carl Hanser Verlag München

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

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

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

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

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

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

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

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

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

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.

Copyright: 2011 Carl Hanser Verlag München

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

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.

$$
\begin{array}{l}
\text{Start}(subj, \text{ExitAltAction}, altJoin(altSplit)) = \textbf{skip} \\
\text{Perform}(subj, \text{ExitAltAction}, altJoin) = \\
\quad \textbf{forall } D \in AltBehDgm(altSplit)\ SID\_state(subj, D) := \textbf{undef} \\
\quad \text{SetCompletionPredicate}_{ExitAltAction}(subj, altJoin(altSplit))
\end{array}
$$

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.

$$
\begin{array}{l}
\text{Perform}(subj, \text{AltAction}, state) = \\
\quad \text{PerformSubDgmStep}(subj, state) \\
\quad\quad \textbf{or}\ \ \text{StartNewSubDgm}(subj, state)\} \\
\textbf{where} \\
\quad \text{PerformSubDgmStep}(s, n) = \\
\quad\quad \textbf{choose } D \in ActiveSubDgm(s, n) \textbf{ in }\ \text{Behavior}(s, SID\_state(s, D)) \\
\quad \text{StartNewSubDgm}(s, n) = \\
\quad\quad \textbf{choose } D \in AltBehDgm(n) \setminus ActiveSubDgm(s, n) \\
\quad\quad\quad SID\_state(s, D) := altEntry(D) \\
\quad\quad\quad \text{Start}(s, service(altEntry(D)), altEntry(D)) \\
\quad ActiveSubDgm(s, n) = \ \{D \in AltBehDgm(n) \mid Active(s, D)\} \\
\quad R \textbf{ or } S = \textbf{choose } X \in \{R, S\} \textbf{ in }\ X
\end{array}
$$

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

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

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

Copyright 2011 Carl Hanser Verlag München

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

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

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

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

5 Notational Structuring Concepts

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

5.1 Macros

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

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

Copyright 2011 Carl Hanser Verlag München

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.

entryState_1	...	entryState_n
	M	
exitEdge_1	...	exitEdge_m

Fig. 7. Macro Table associated to a Macro State

The macro insertion constraints are therefore about how the *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.

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

Copyright: 2011 Carl Hanser Verlag München

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

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

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

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

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

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

Copyright 2011 Carl Hanser Verlag München

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

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

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

5.2 Interaction View Normalization of Subject Behavior Diagrams

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

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

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

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

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

Copyright 2011 Carl Hanser Verlag München

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

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

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

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

Case 2. *start* is an internal function node. Then its function may have to be compressed with functions of successive function states. Let $Path_1$ be the set of all paths $state_1, \ldots, state_{n+1}$ 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.

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

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

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

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

5.3 Process Networks

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

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

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

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

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

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

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

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

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

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

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

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

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

Copyright: 2011 Carl Hanser Verlag München

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

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:

$$\begin{aligned}&ObserverView(observer, D_{subj}) =\\ &\quad InteractionView(D_{subj} \downarrow observer)\end{aligned}$$

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

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

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

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

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

Copyright 2011 Carl Hanser Verlag München

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

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

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

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

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

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

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

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

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

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

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

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

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

6 Two model extension disciplines

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

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

Copyright 2011 Carl Hanser Verlag München

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

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

6.1 Interrupt Extension

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

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

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

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

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

Copyright: 2011 Carl Hanser Verlag München

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

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 *InterruptStates* s_i of D, whereas $\textsc{Behavior}(subj, state)$ remains unchanged for the other D *states* and for *states* of D'—which are assumed to be disjoint from those of D:[41]

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

$\textsc{InterruptBehavior}(subj, s_i) =$ // 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.

Copyright: 2011 Carl Hanser Verlag München

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

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 $\text{BEHAVIOR}(subj, s_i)$ 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'.

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

Copyright 2011 Carl Hanser Verlag München

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

7 S-BPM Interpreter in a Nutshell

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

7.1 Subject Behavior Diagram Interpretation

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

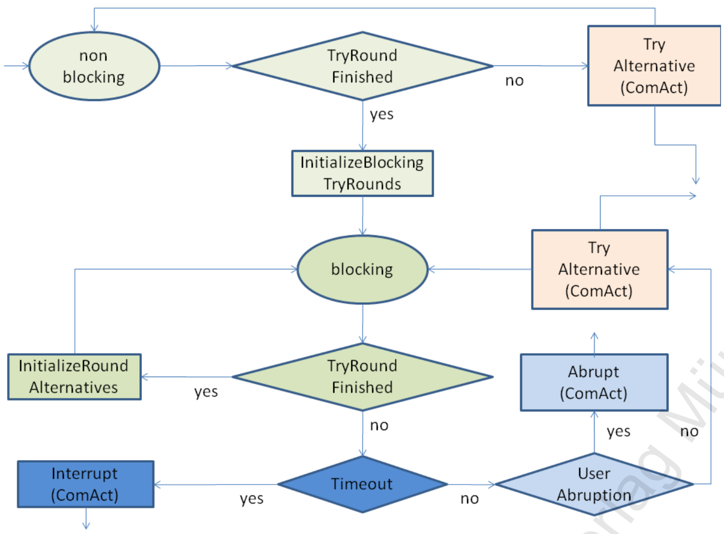
$\textsc{Behavior}(subj, state) =$
  if $SID\_state(subj) = state$ **then**
    if $Completed(subj, service(state), state)$ **then**
      let $edge =$
        $select_{Edge}(\{e \in OutEdge(state) \mid ExitCond(e)(subj, state)\})$
          $\textsc{Proceed}(subj, service(target(edge)), target(edge))$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

7.2 Alternative Send/Receive Round Interpretation

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

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

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



Interpretation of Auxiliary Macros

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

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

Copyright 2011 Carl Hanser Verlag München

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

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

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

Copyright: 2011 Carl Hanser Verlag München

7.3 MsgElaboration Interpretation for Multi Send/Receive

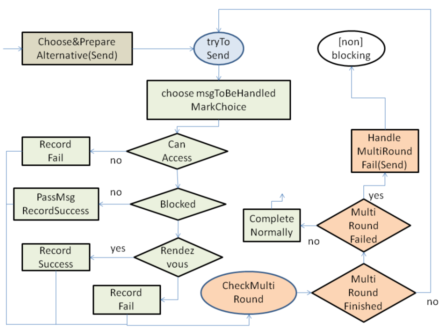
TryAlternative$_{ComAct}(subj, state) =$
$\quad$ Choose&PrepareAlternative$_{ComAct}(subj, state)$
$\quad\quad$ **seq** Try$_{ComAct}(subj, state)$

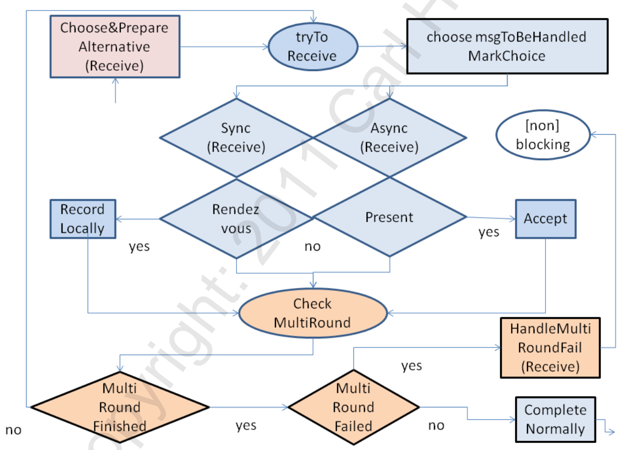
Choose&PrepareAlternative$_{ComAct}(subj, state) =$
$\quad$ **let** $alt = select_{Alt}(RoundAlternative(subj, state), priority(state))$
$\quad\quad$ PrepareMsg$_{ComAct}(subj, state, alt)$
$\quad\quad$ ManageAlternativeRound$(alt, subj, state)$
$\quad\quad$ **where**
$\quad\quad\quad$ ManageAlternativeRound$(alt,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

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

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

7.4 Multi Send/Receive Round Interpretation





Copyright: 2011 Carl Hanser Verlag München

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

$\textsc{Try}_{ComAct}(subj, state) =$
  choose $m \in MsgToBeHandled(subj, state)$
    $\textsc{MarkChoice}(m, subj, state)$
    if $ComAct = Send$ **then**
      let $receiver = receiver(m),\ pool = inputPool(receiver)$
        if not $CanAccess(subj, pool)$ **then**
          $\textsc{ContinueMultiRound}_{Fail}(subj, state, m)$
        else $\textsc{Try}_{Async(Send)}(subj, state, m)$
    if $ComAct = Receive$ **then**
      if $Async(Receive)(m)$ **then** $\textsc{Try}_{Async(Receive)}(subj, state, m)$
      if $Sync(Receive)(m)$ **then** $\textsc{Try}_{Sync(Receive)}(subj, state, m)$
where
  $\textsc{MarkChoice}(m, subj, state) =$
    $\textsc{Delete}(m, MsgToBeHandled(subj, state))$
    $currMsgKind(subj, state) := m$

$\textsc{Try}_{Async(ComAct)}(subj, state, m) =$
  if $PossibleAsync_{ComAct}(subj, m)$ // async communication possible
    then $\textsc{Async}(ComAct)(subj, state, m)$
    else
      if $ComAct = Receive$ **then**
        $\textsc{ContinueMultiRound}_{Fail}(subj, state, m)$
      if $ComAct = Send$ **then** $\textsc{Try}_{Sync(ComAct)}(subj, state, m)$
$\textsc{Try}_{Sync(ComAct)}(subj, state, m) =$
  if $PossibleSync_{ComAct}(subj, m)$ // sync communication possible
    then $\textsc{Sync}(ComAct)(subj, state, m)$
    else $\textsc{ContinueMultiRound}_{Fail}(subj, state, m)$

Copyright: 2011 Carl Hanser Verlag München

7.5 Actual Send Interpretation

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

$$
\begin{array}{l}
\textsc{Sync}(\mathit{Send})(\mathit{subj}, \mathit{state}, \mathit{msg}) = \\
\quad \textsc{ContinueMultiRound}_{\mathit{Success}}(\mathit{subj}, \mathit{state}, \mathit{msg}) \\
\mathit{PossibleSync}_{\mathit{Send}}(\mathit{subj}, \mathit{msg}) \text{ iff } \mathit{RendezvousWithReceiver}(\mathit{subj}, \mathit{msg})
\end{array}
$$

$$
\begin{array}{l}
\mathit{RendezvousWithReceiver}(\mathit{subj}, \mathit{msg}) \text{ iff} \\
\quad \mathit{tryMode}(\mathit{rec}) = \mathit{tryToReceive} \textbf{ and } \mathit{Sync}(\mathit{Receive})(\mathit{currMsgKind}) \\
\quad\quad \textbf{and } \mathit{SyncSend}(\mathit{msg}) \textbf{ and } \mathit{Match}(\mathit{msg}, \mathit{currMsgKind}) \\
\textbf{where} \\
\quad \mathit{rec} = \mathit{receiver}(\mathit{msg}),\ \mathit{recstate} = \mathit{SID\_state}(\mathit{rec}) \\
\quad \mathit{currMsgKind} = \mathit{currMsgKind}(\mathit{rec}, \mathit{recstate}) \\
\quad \mathit{blockingRow} = \\
\quad\quad \mathit{first}(\{r \in \mathit{constraintTable}(\mathit{rec}) \mid \mathit{ConstraintViolation}(\mathit{msg}, r)\}) \\
\quad \mathit{SyncSend}(\mathit{msg}) \text{ iff } \mathit{size}(\mathit{blockingRow}) = 0
\end{array}
$$

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

7.6 Actual Receive Interpreation

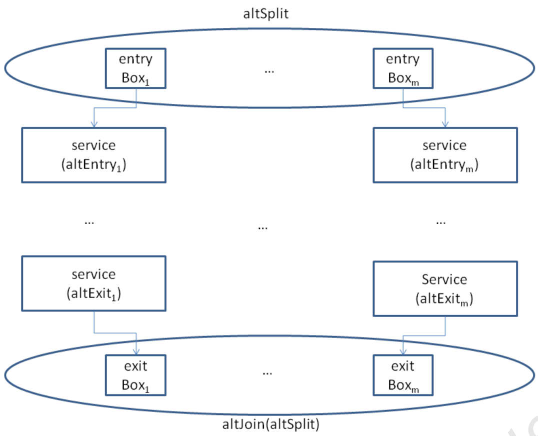
$\textsc{Async}(\mathit{Receive})(\mathit{subj}, \mathit{state}, \mathit{msg}) =$
$\quad \textsc{Accept}(\mathit{subj}, \mathit{msg})$
$\quad \textsc{ContinueMultiRound}_{\mathit{Success}}(\mathit{subj}, \mathit{state}, \mathit{msg})$
where
$\quad \textsc{Accept}(\mathit{subj}, m) =$
$\quad\quad \textbf{let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

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

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

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

7.7 Alternative Action Interpretation



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

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

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

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

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

Auxiliary Wait/Exit Rule Interpretation

$$\begin{aligned}
&\textsc{Start}(subj, \textsc{AltActionWait}, exitBox) = \\
&\quad \textsc{InitializeCompletionPredicate}_{AltActionWait}(subj, exitBox) \\
&\textsc{Perform}(subj, \textsc{AltActionWait}, exitBox) = \textbf{skip}
\end{aligned}$$

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

7.8 Interrupt Behavior

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

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

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.

Copyright 2011 Carl Hanser Verlag München

[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.

Bibliography

[Kee76] E. L. Keenan. 1976.