# Standard for the Subject-oriented Specification of Systems

Albert Fleischmann *Editor*

# Standard for the Subject-oriented Specification of Systems

Working Document

Egon Börger
xyz

Stephan Borgert
xyz

Matthes Elstermann
xyz

Albert Fleischmann
xyz

Werner Schmidt
xyz

Robert Singer
FH JOANNEUM–University of Applied Sciences

Christian Stary
xyz

André Wolski
TU Darmstadt



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.

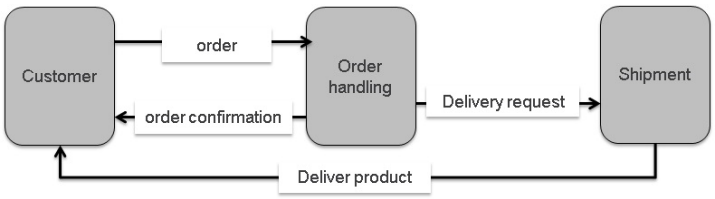


Figure 1.1: The Communication Structure in the Order Process

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.

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

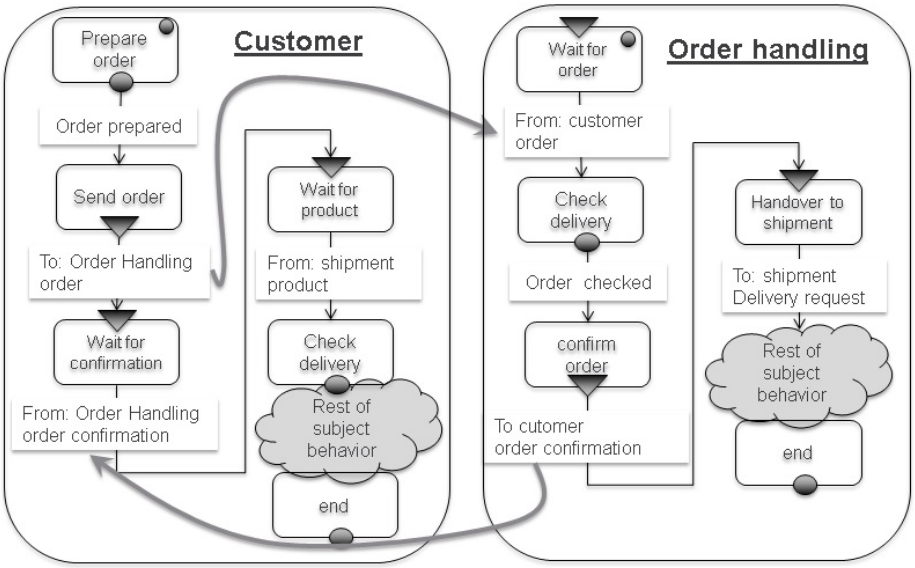


Figure 1.2: The Behavior of Subjects

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".

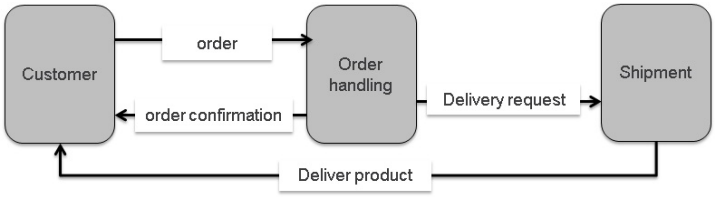


Figure 1.3: The Communication Structure with Change Message

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 "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

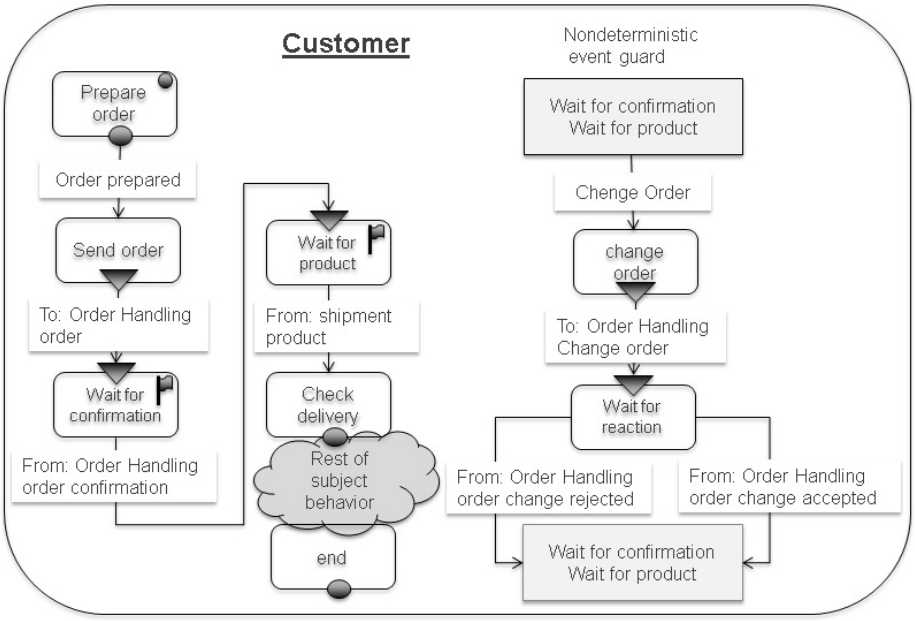


Figure 1.4: Customer is allowed to Change Orders

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 **??** 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]).

## 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.

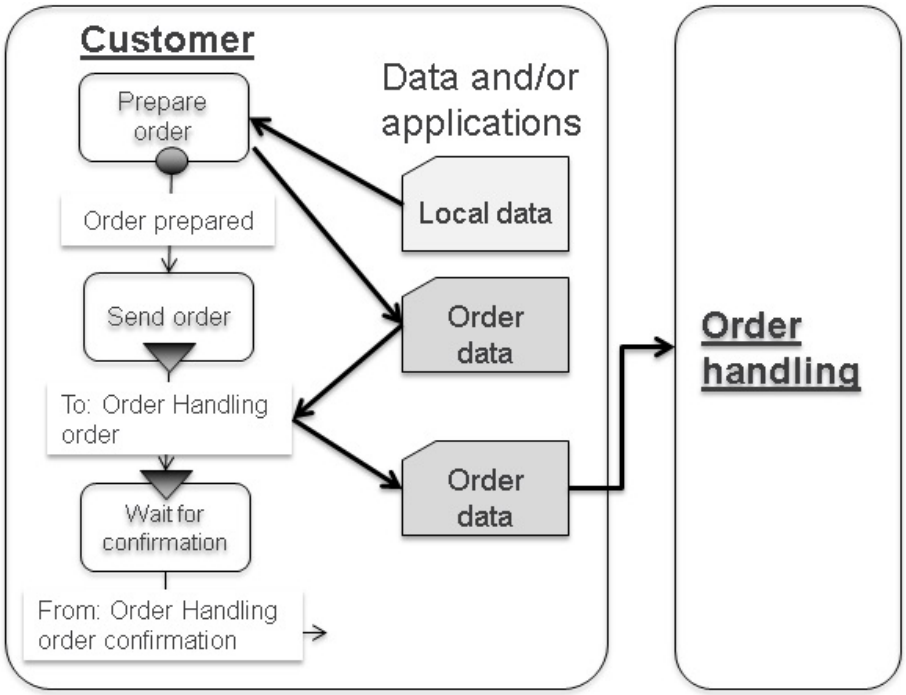


Figure 1.5: Subjects and Objects

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.

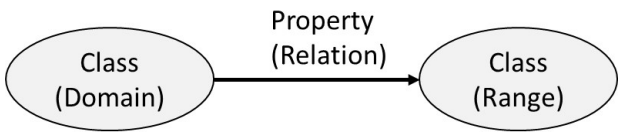


Figure 1.6: RDF graphic

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).

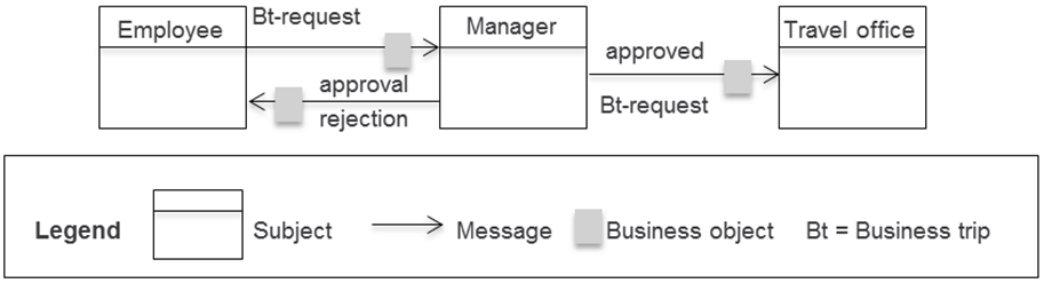


Figure 2.1: Subject interaction diagram for the process 'business trip application'

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.

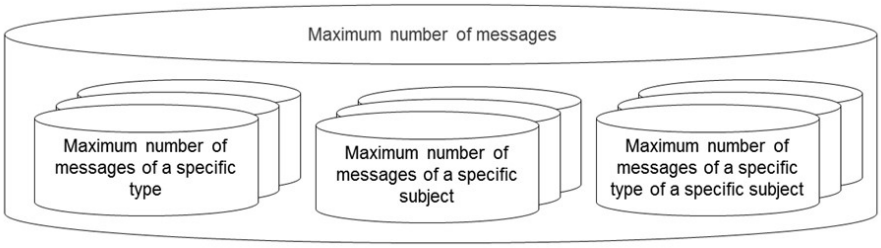


Figure 2.2: Configuration of Input Pool Parameters

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.

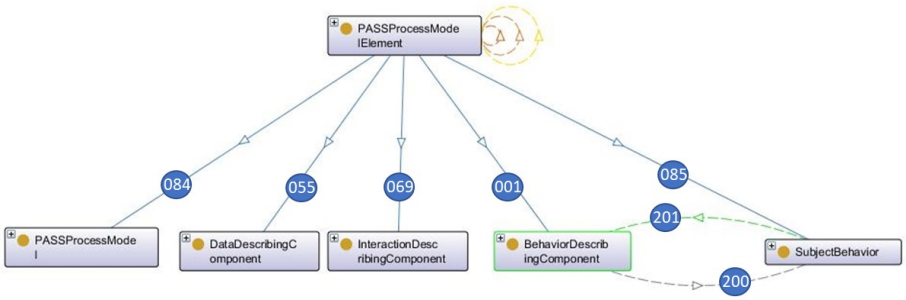


Figure 2.3: Elements of PASS Process Models

The class `PASSProcessModelElement` has five subclasses (subclass relations 084, 055, 069, 001 and 085 in figure 2.3). Only the classes `PASSProcessModel`, `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 `hasRelation 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`

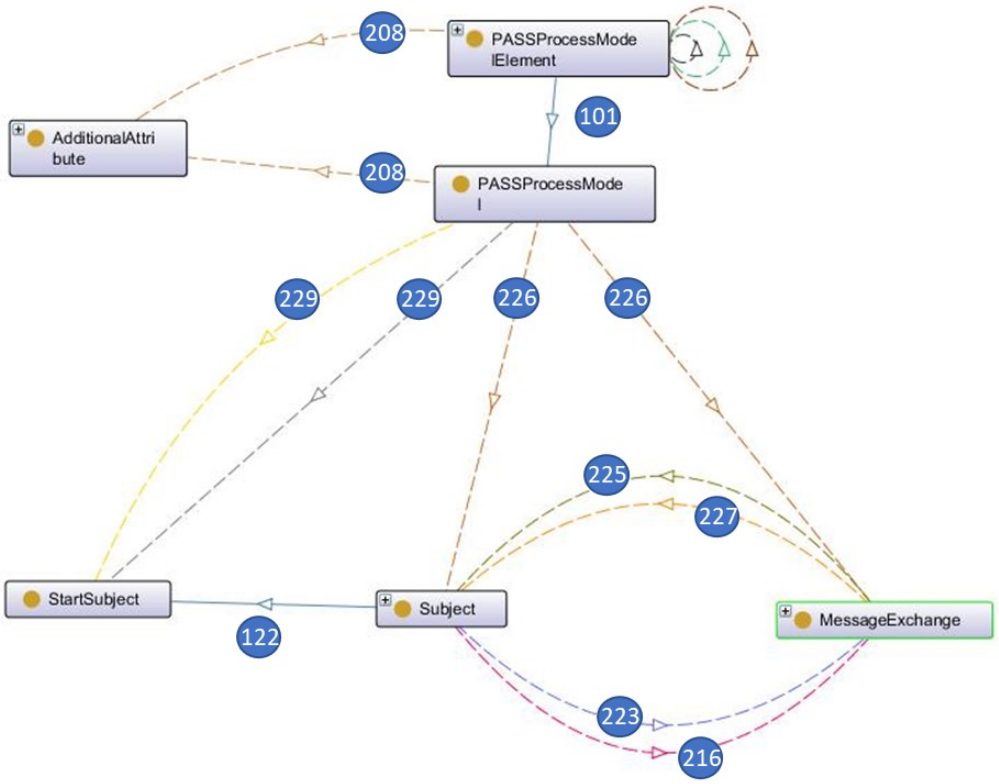


Figure 2.4: PASS Process Modell

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.

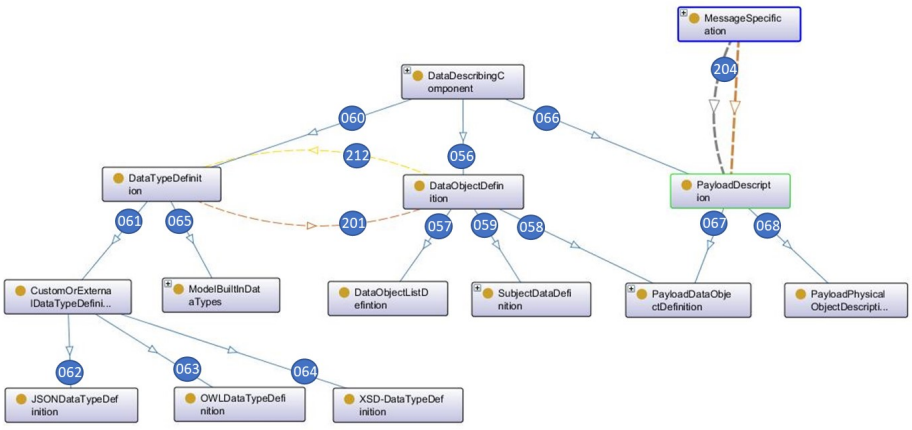


Figure 2.5: Data Description Component

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.

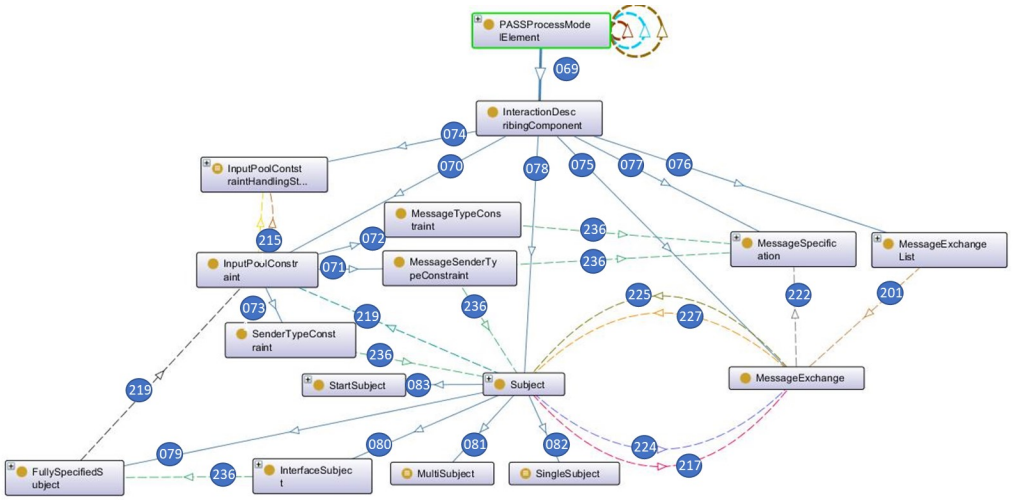


Figure 2.6: Subject Interaction Diagram

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). Con-

straints 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.

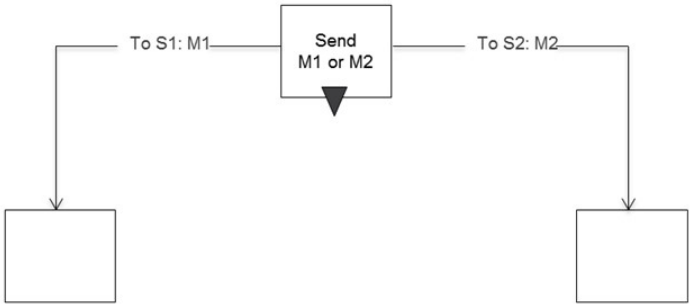


Figure 3.1: Example of alternative sending

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.

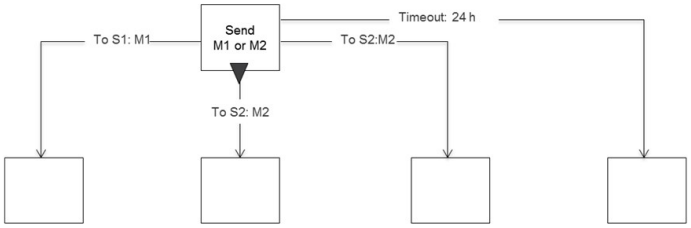


Figure 3.2: Send using time monitoring

### 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.

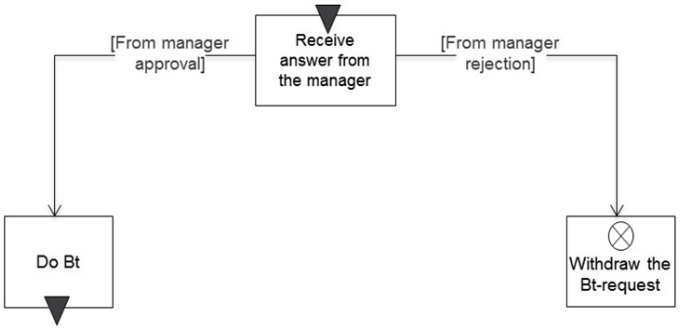


Figure 3.3: Example of alternative receiving

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.

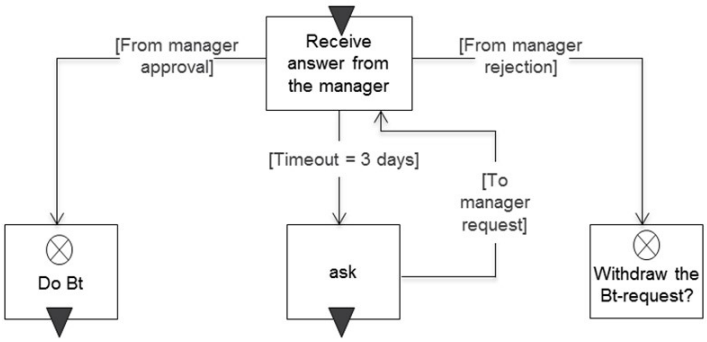


Figure 3.4: Time monitoring for message reception

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.

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.

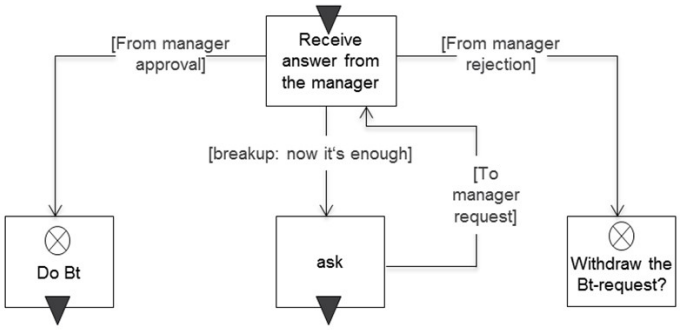


Figure 3.5: Message reception with manual interrupt

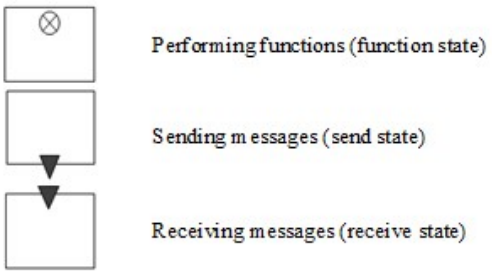


Figure 3.6: State types and coresponding symbols

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.

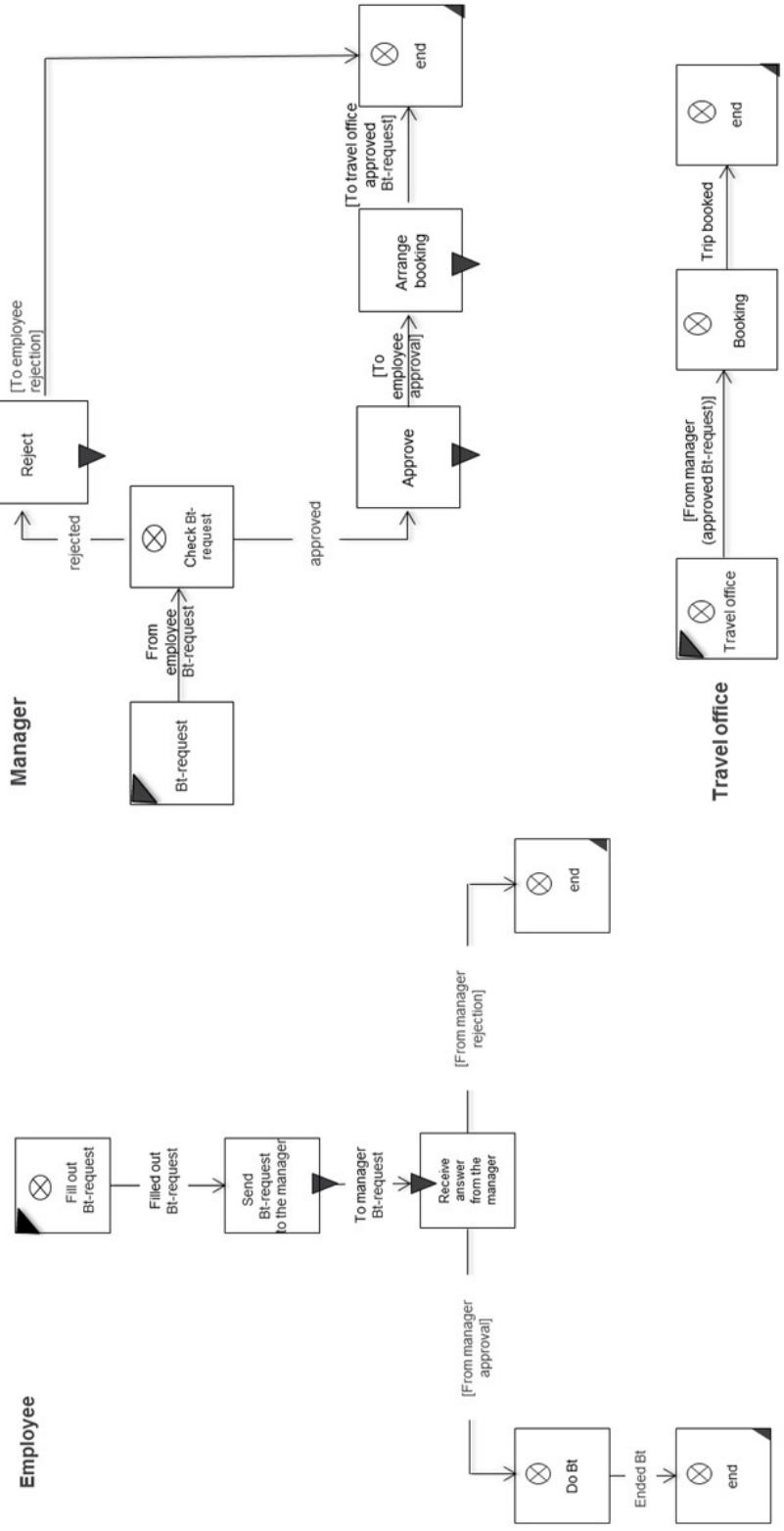


Figure 3.7: Subject behavior diagram for the subjects 'employee', 'manager', and 'travel office'

### 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.

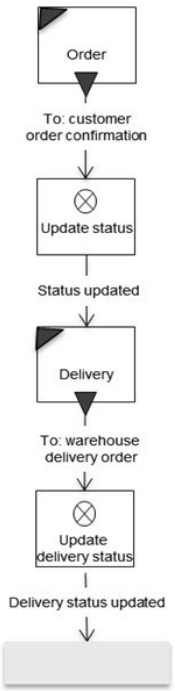


Figure 3.8: Behavior macro class 'request for approval'

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'.

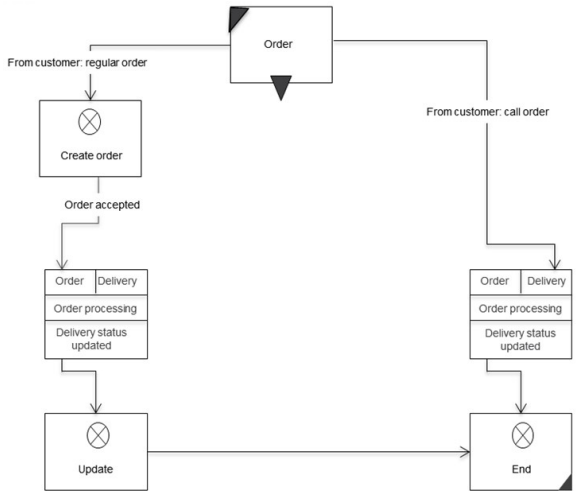


Figure 3.9: Subject behavior for order processing with macro integration

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.

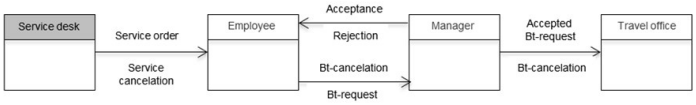


Figure 3.10: Communication structure diagram (CSD) of the business trip application

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

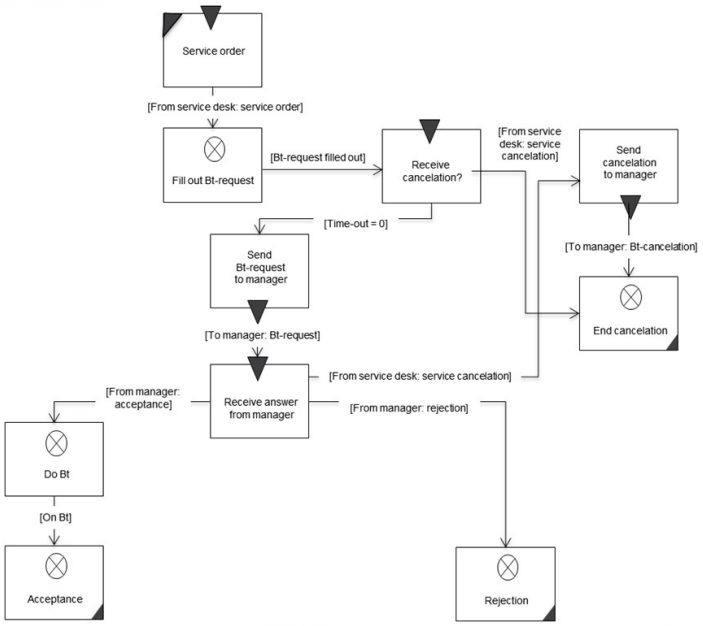


Figure 3.11: Handling the cancelation message using existing constructs

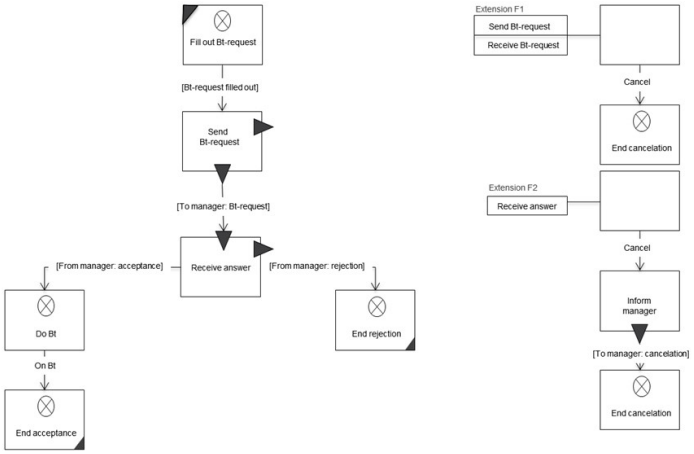


Figure 3.12: Behavior of subject 'employee' with exception handling

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.

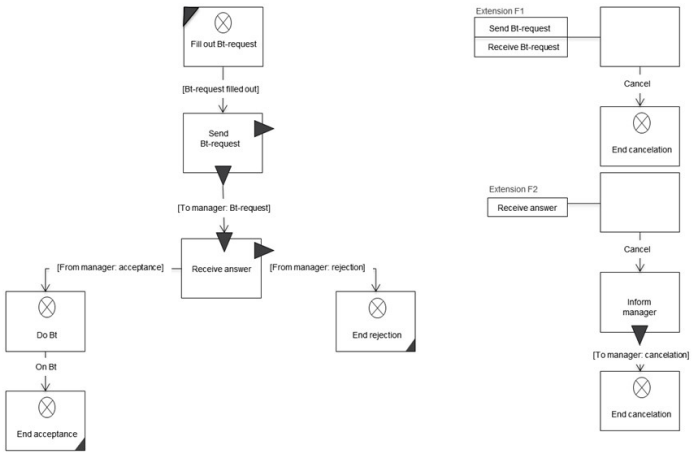


Figure 3.13: Subject behavior of employee with behavior extensions

*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.

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).

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

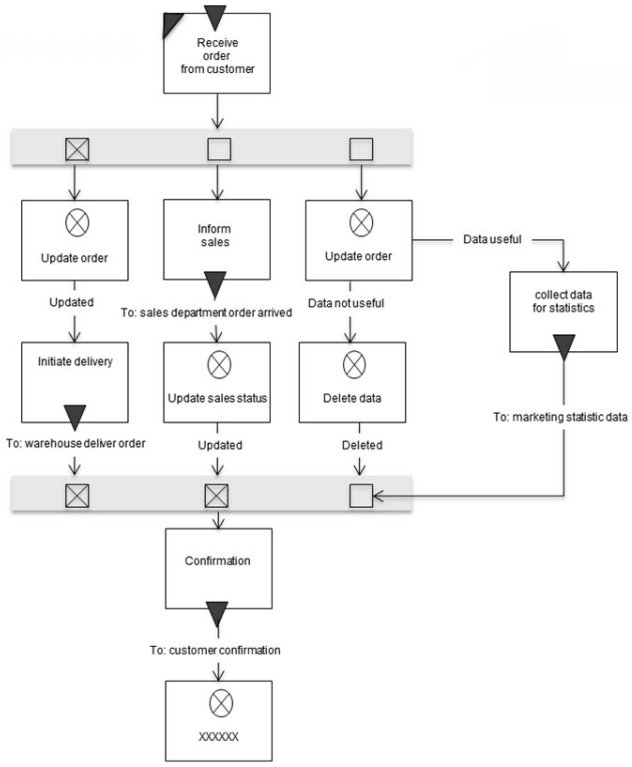


Figure 3.14: Example of Process Alternatives

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).

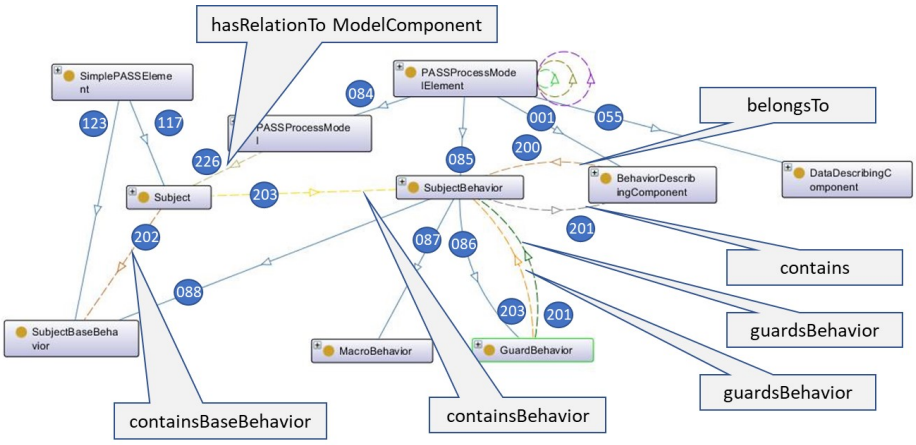


Figure 3.15: Structure of Subject Behavior Specification

### 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 transitions (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

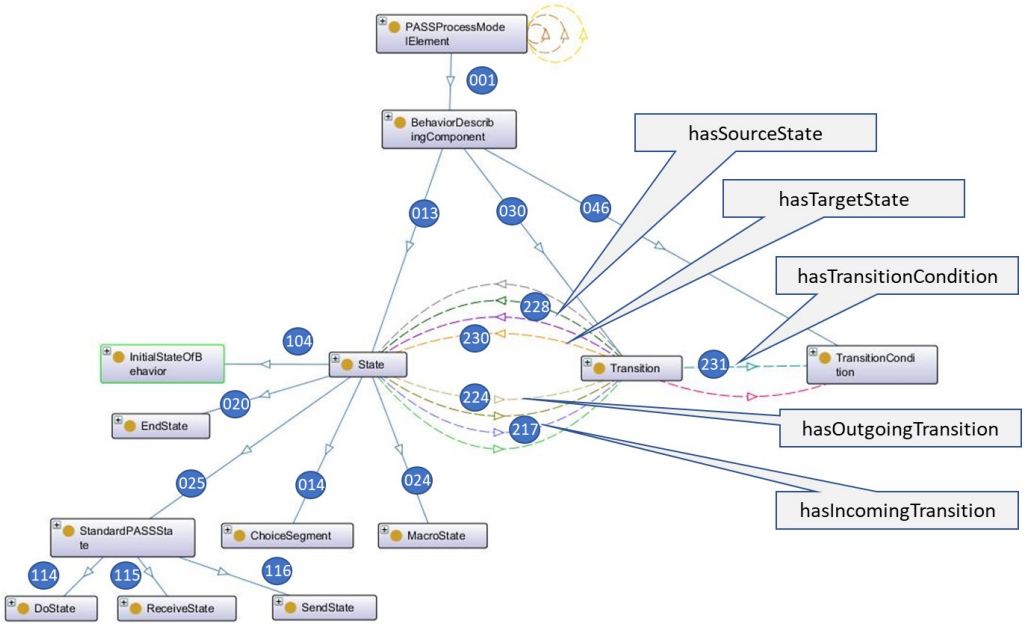


Figure 3.16: Subject Behavior describingComponent

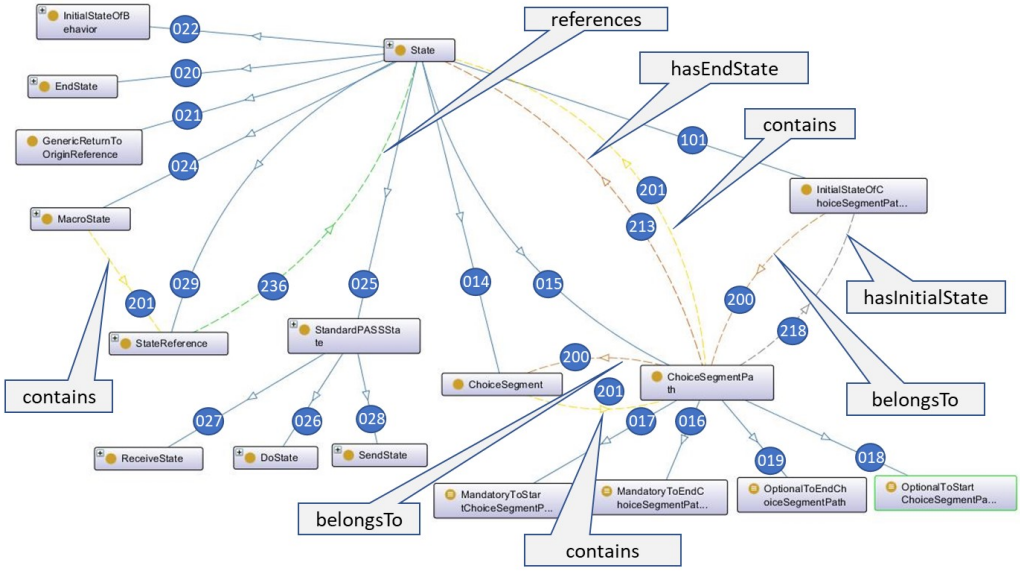


Figure 3.17: Details of States

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.

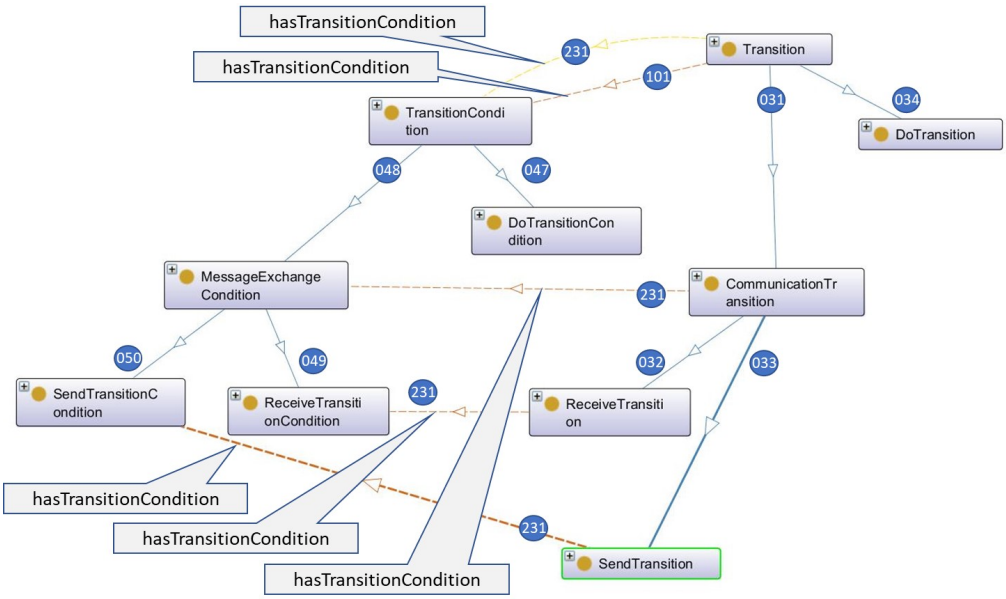


Figure 3.18: Details of transitions

## 3.3 ASM Definition of Subject Execution

This section provides a commented overview of the CoreASM PASS Reference Implementation provided in Appendix C. Only a reduced set of language elements are shown here, advanced Functions and implementation details are left out, which allowed it to shorten various rules for a focus on their core semantics. Therefore all contents of this section are a non-normative introduction to the topic of formal execution semantics.

There are some conceptional differences between the reference implementation and the OWL description. The End State is a distinguished state instead a property on a Do State or Receive State. Choise Segment Paths are always both mandatory to start and end, also they have to be started with the Modal Split Function and joined with the Modal Join Function. Furthermore the reference implementation supports advanced features that are not covered by the OWL

description, for example the Mobility of Channels concept and CorrelationIDs. A complete comparision of the conceptual differences is given in C.1.

### 3.3.1 Architecture

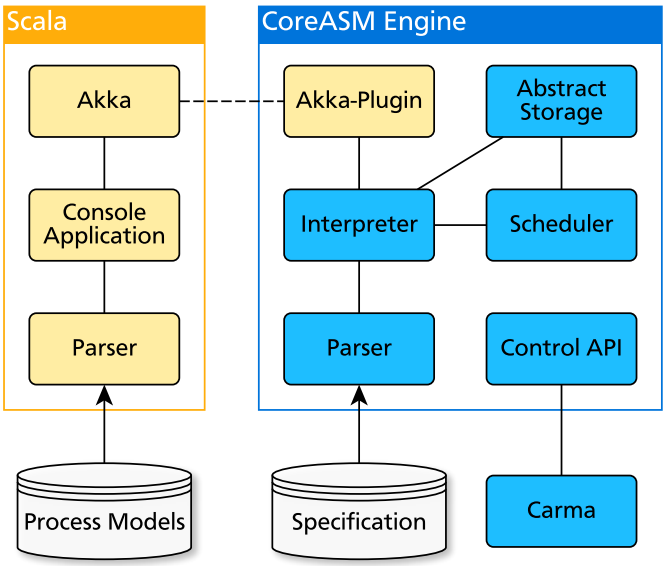


Figure 3.19: Architecture of the reference implementation

The overall architecture is shown in figure 3.19 and consists on the top level of a console application as user interface and the CoreASM engine as execution environment for the PASS-interpreter specification. Both high level components are running in parallel in separate Java Virtual Machine processes. The PASS-interpreter specification file includes an ASM specification of asynchronous multi-agent ASMs, executing each Subject by one ASM-agent each.

The CoreASM engine parses the specification file and executes the rules in the interpreter in cooperation with the abstract storage and the scheduler. The abstract storage stores the state of the ASM and is also used to support TurboASMs and their submachine states.

The scheduler is used to support asynchronous multi-agent ASMs; with the *default* scheduling policy it selects a random subset of the running ASM agents and calls the interpreter for the parallel evaluation of their main rules. If the resultant update sets are consistent they are applied in the abstract storage, otherwise the scheduler selects a different subset of the running ASM agents for a re-evaluation. Supplementary the *onebyone* scheduling policy can be used to evaluate only a single agent per step and the *allfirst* policy can be used to attempt to evaluate all agents at once before falling back to the default scheduling policy in case of inconsistencies. We use the *allfirst* scheduling policy in our work for performance reasons but could use the *default* policy as well.

The Carma application is a simple Java application provided by the CoreASM framework to allow the usage of the CoreASM engine as standalone application in a command line.

The console application is developed in Scala. A user can load, start and execute locally stored S-BPM Process Models with it. The interaction between the console application and the CoreASM engine is realized by Akka actors. The

Akka actor of the engine has full read and write access to all CoreASM functions by using a Plug-In interface.

### 3.3.2 Foundation

The interpreter uses asynchronous multi-agent Turbo ASMs. It supports the concurrent execution of multiple process models and multiple instances of each process model. Each instance has a unique *ProcessInstanceID* (short: `PI`) assigned. Within a process instance multiple instances of a subject can occur (MultiSubject concept). Each subject instance has an agent assigned, to distinguish the subject instances. This agent is identified by it name. We therefore identify a subject instance by the tuple (Process Model ID, Process Instance ID, Subject ID, Agent Name). This tuple is called *Channel* (short: `ch`), as it used in the mobility of channels concept in order to support distributed communication patterns.

In the interpreter, each subject instance has an ASM Agent assigned, that keeps track of its current state, where we mean state in the sense of which subject data is present, the content of the inputpool queues and also the active SBD states. This state is stored in ASM functions and assigned to the *Channel*.

```
// ASM Agent -> Channel
function channelFor : Agents -> LIST

derived processIDFor(a)       = processIDOf(channelFor(a))
derived processInstanceFor(a) = processInstanceOf(channelFor(a))
derived subjectIDFor(a)       = subjectIDOf(channelFor(a))
derived agentFor(a)           = agentOf(channelFor(a))

derived processIDOf(ch)       = nth(ch, 1)
derived processInstanceOf(ch) = nth(ch, 2)
derived subjectIDOf(ch)       = nth(ch, 3)
derived agentOf(ch)           = nth(ch, 4)
```

Listing 1: Channel definitions

In the function `channelFor` the assignment from the ASM Agents to their *Channels* is stored. The derived functions are used to lookup certain tuple elements of the channel.

Analogous to programming languages we call Subject Data *Variables*. Their scope is either bound to a Subject or a Macro Instance and can only be accessed by the Subject. Variables are identified by their name and have an explicit data type and a value.

Variables are stored in the `variable` function which maps from the Subject's Channel, the scoped Macro Instance ID and the Variable's name to a pair of the data type and value. For Variables that are not scoped to a Macro Instance, and are therefore accessible for any state in any Macro Instance of the Subject, the unused Macro Instance ID 0 is used.

The function `variableDefined` is used to keep track of Variables that are in use, so that their content can be reset upon the termination of a Macro Instance or the Subject, respectively.

```
// Channel * macroInstanceNumber * varname -> [vartype, content]
function variable : LIST * NUMBER * STRING -> LIST

// Channel -> Set[(macroInstanceNumber, varname)]
function variableDefined : LIST -> SET
```

Listing 2: variable

### 3.3.3 Interaction Definitions

We implement the interaction between Subjects as asynchronous Message exchange. Messages are placed into the Inputpool of the receiver where they are then retrieved from when the receiver is in a Receive state.

*Messages*

The message content consists of the actual payload and its data type. As this reference implementation is intended only for an abstract process execution we abstract the payload / business objects to be just a text given as string.

For the communication with MultiSubjects, i.e. sending the same Message to multiple Agents of one Subject, we use an *all-or-none* strategy. This is accomplished by separating the sending of a Message into two phases: first a reservation Message is placed at each receiver into the Inputpool. Only after all reservations could be placed they are then replaced with the actual Message.

*Inputpool*

The Inputpool is structured into multiple FIFO queues per Subject ID of the sender, the Messagetype and the CorrelationID.

The Inputpool can be limited to allow a maximum number of Messages and reservations per sender Subject ID and Messagetype; in case the Inputpool Limit is reached no additional reservation can be placed. To support the proper termination of a Subject a specific queue (and also the complete Inputpool) can be closed, in which case no additional reservation can be placed into it, too.

```
// Channel * senderSubjID * msgType * correlationID
//   -> [msg1, msg2, ...]
function inputPool : LIST * STRING * STRING * NUMBER -> LIST

/* stores all locations where an inputPool was defined */
// Channel -> {[senderSubjID, msgType, correlationID], ...}
function inputPoolDefined : LIST -> SET

// Channel * senderSubjID * msgType * correlationID
function inputPoolClosed : LIST * STRING * STRING * NUMBER
  -> BOOLEAN
```

Listing 3: inputPool

The queues of the Inputpool are stored in the `inputPool` function. The function `inputPoolDefined` is used to keep track of the locations of the queues that are in use, so that their content can be checked upon termination. The function `inputPoolClosed` is used to store whether a queue is closed. The

special location `inputPoolClosed(ch, undef, undef, undef)` is used to store whether the complete Inputpool is closed.

We define the function `derived availableMessages(receiverChannel, senderSubjectID, msgType, ipCorrelationID)` to return the Messages from the location `inputPool(receiverChannel, senderSubjectID, msgType, ipCorrelationID)` that can be received, i.e. that it filters out reservations and reduces Messages from the same sender to only the oldest one.

### 3.3.4 Subject Behavior

As depicted in figure **??**, a Subject Behavior consists at least of one *Main Macro* and might have an arbitrary number of minor Macros, called *Additional Macros*.

```
rule SubjectBehavior =
  MacroBehavior(1)
```

Listing 4: SubjectBehavior

The `MacroBehavior` rule controls the evaluation of all active states for the given Macro Instance ID `MI`.

```
rule MacroBehavior(MI) =
  let ch = channelFor(self) in
  choose stateNumber in activeStates(ch, MI) do
    Behavior(MI, stateNumber)
```

Listing 5: MacroBehavior

From that list a state `stateNumber` is chosen to be evaluated with the `Behavior` rule.

The evaluation of a state is structured into three main phases: initialization, the state function and an optional transition behavior.

The state function is responsible for the selection of an outgoing transition and has to supervise the Timeout. It also has to enable and disable its outgoing transitions, meaning that transitions can be available depending on some dynamic state, for example whether a certain Message is present in the Inputpool. Usually the outgoing transition will be selected by the environment, however with auto-transitions it is possible that such an transition is automatically selected as soon as it becomes enabled and as long as there are no other transitions to select from.

In the beginning the `Behavior` rule initializes the state with the `StartState` rule, which will set `initializedState` to `true`. If the function should not be aborted the `Perform` rule calls the state behavior of the underlying function until it is `completed`. In the next phase the selected transition will be initialized by the `StartSelectedTransition` rule and the transition behavior will be performed with the `PerformTransition` rule until it is completed as well. As last step the `Proceed` rule removes the current state and adds the selected transition's target state.

The environment has full read access to all functions of this semantics and knows therefore each running Subject, their Macro Instances and their active states.

```
rule Behavior(MI, currentStateNumber) =
 let s = currentStateNumber,
     ch = channelFor(self) in
  if (initializedState(ch, MI, s) != true) then
    StartState(MI, s)
  else if (abortState(MI, s) = true) then
    AbortState(MI, s)
  else if (completed(ch, MI, s) != true) then
    Perform(MI, s)
  else if (initializedSelectedTransition(ch, MI, s) != true) then
    StartSelectedTransition(MI, s)
  else
   let t = selectedTransition(ch, MI, s) in
    if (transitionCompleted(ch, MI, t) != true) then
      PerformTransition(MI, s, t)
    else
      Proceed(MI, s, targetStateNumber(processIDFor(self), t))
```

Listing 6: Behavior

To define a homogeneous interface between the Function semantics and the environment we define the function `wantInput` to be written by an Function when it requires an external input, for example if an outgoing transition has to be chosen.

```
// Channel * MacroInstanceNumber * StateNumber -> Set[String]
function wantInput : LIST * NUMBER * NUMBER -> SET
```

Listing 7: wantInput

This function is read by our console application for all active states to present the user a list of possible decisions that can be made.

The environment then writes its external input in a corresponding function, for example for a transition decision into the `selectedTransition` function, and clears the `wantInput` function of that state.

The `SelectTransition` rule adds the `"TransitionDecision"` requirement into the `wantInput` function.

```
rule SelectTransition(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
  if (|outgoingEnabledTransitions(ch, MI, s)| = 0) then
    skip // BLOCKED: none to select
  else if (not(contains(wantInput(ch, MI, s),
                        "TransitionDecision"))) then
    add "TransitionDecision" to wantInput(ch, MI, s)
  else
    skip // waiting for selectedTransition
```

Listing 8: SelectTransition

If the `wantInput` function already contains the `"TransitionDecision"`

requirement nothing needs to be done and another state can be evaluated by the `MacroBehavior` rule. The same applies if there are no outgoing transitions enabled. Otherwise the requirement is added to the `wantInput` function.

### 3.3.5 Internal Action

The *Internal Action* is used to model DoStates, Subject-internal activities and decisions. It is labeled with a textual description of the activity that the Agent should perform. The outgoing transitions are labeled with a textual description of the possible execution results. Since the activity is performed outside of the interpreter all outgoing transitions are enabled from the beginning on and no transition rule has to be defined. Therefore the state function only consists of the timeout check and transition selection.

```
rule StartInternalAction(MI, currentStateNumber) = {
  StartTimeout(MI, currentStateNumber)

  EnableAllTransitions(MI, currentStateNumber)
}
```

Listing 9: StartInternalAction

The initialization of the InternalAction starts the Timeout and enables all outgoing transitions.

```
rule PerformInternalAction(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
    if (shouldTimeout(ch, MI, s) = true) then {
      SetCompleted(MI, s)
      ActivateTimeout(MI, s)
    }
    else if (selectedTransition(ch, MI, s) != undef) then
      SetCompleted(MI, s)
    else
      SelectTransition(MI, s)
```

Listing 10: PerformInternalAction

The state function checks if a Timeout should be activated; otherwise the `SelectTransition` rule is called until the `selectedTransition` function has been set.

### 3.3.6 Send Function

The Send Function sends a Message. Disregarding the optional Timeout and Cancel transitions it must have exactly one outgoing transition which has to have parameters that define at least the Messagetype and the receiver's Subject ID.

We use an *all-or-none* strategy to send Messages to MultiSubjects, which means that the actual Message is only send when all receivers are able to store it in their Inputpool. Therefore the Send Function is structured into two phases: in the first phase, realized as state function, reservation Messages are placed and in

the second phase, realized as transition function, these reservations are replaced with the actual Message.

To place a reservation in a receiver's Inputpool there must be space available in the corresponding queue and the receiver must not be *non-proper* terminated.

```
// Channel * MacroInstanceNumber * StateNumber -> Set[Messages]
function receivedMessages : LIST * NUMBER * NUMBER -> SET

// Channel * MacroInstanceNumber * StateNumber -> Set[Channel]
function receivers : LIST * NUMBER * NUMBER -> SET

// Channel * MacroInstanceNumber * StateNumber
function messageContent       : LIST * NUMBER * NUMBER -> LIST
function messageCorrelationID : LIST * NUMBER * NUMBER -> NUMBER
function messageReceiverCorrelationID : LIST * NUMBER * NUMBER
  -> NUMBER

// Channel * MacroInstanceNumber * StateNumber -> Set[Channel]
function reservationsDone : LIST * NUMBER * NUMBER -> SET

function nextCorrelationID : -> NUMBER
function nextCorrelationIDUsedBy : NUMBER -> Agents
```

Listing 11: receivedMessages

The Send Function stores the message content it has to send in the `messageContent` function. The `receivers` function is used to store the required receivers. When a reservation message has been placed at a receiver its Channel is added to the `reservationsDone` function.

The global function `nextCorrelationID` is used to increment CorrelationIDs. To ensure the uniqueness of a CorrelationID the `nextCorrelationIDUsedBy` function is used to store the ASM agent that used the given CorrelationIDs.

The initialization of the Send Function resets / initializes the `receivers` and `reservationsDone` functions. If the communication transition has a `messageContentVar` parameter given the content of that Variable is loaded and stored in the `messageContent` function. If it has a `messageNewCorrelationVar` parameter given a new CorrelationID is created and stored in the `messageCorrelationID` function. It will be stored in the Variable after the messages have been send. If the message that will be send correlates to a previous message from the receiver the CorrelationID from the Variable in `messageWithCorrelationVar` will be loaded so that the message is stored in the corresponding inputpool queue of the receiver.

In the state function the `SelectReceivers` rule is called until the `receivers` have been selected. The `SelectReceivers` rule interacts with the environment through the Selection and SelectAgent Functions in order to either select existing Channels from a Variable, given as parameter on the communication edge, or to select new assignments of Agents for the receiver Subject.

The Timeout is started only when the `receivers` and the `messageContent` functions are defined. Until all reservations are placed, and if no Timeout occurs, the `DoReservations` rule attempts to place further reservation messages. If all reservations are placed the `TryCompletePerformSend` rule completes the state function, depending on whether no receiver is *non-proper* terminated.

```
rule StartSend(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  // there must be exactly one transition
  let  t = first_outgoingNormalTransition(pID, s) in {
    receivers(ch, MI, s) := undef
    reservationsDone(ch, MI, s) := {}
    let mcVName = messageContentVar(pID, t) in
      messageContent(ch, MI, s) := loadVar(MI, mcVName)

    // generate new CorrelationID now, it will be stored
    // in a Variable once the message(s) are send
    let cIDVName = messageNewCorrelationVar(pID, t) in
      if (cIDVName != undef and cIDVName != "") then {
        messageCorrelationID(ch, MI, s) := nextCorrelationID
        nextCorrelationID := nextCorrelationID + 1
        // ensure no other agent uses this same correlationID
        nextCorrelationIDUsedBy(nextCorrelationID) := self
      }
      else
        messageCorrelationID(ch, MI, s) := 0

    // load receiver IP CorrelationID now, to avoid
    // influences of any changes of that variable
    let cIDVName = messageWithCorrelationVar(pID, t) in
    let cID = loadCorrelationID(MI, cIDVName) in
      messageReceiverCorrelationID(ch, MI, s) := cID
  }
```

Listing 12: StartSend

```
rule PerformSend(MI, currentStateNumber) =
  let ch = channelFor(self),
       s = currentStateNumber in
  if (receivers(ch, MI, s) = undef) then
    SelectReceivers(MI, s)
  else if (messageContent(ch, MI, s) = undef) then
    SetMessageContent(MI, s)
  else if (startTime(ch, MI, s) = undef) then
    StartTimeout(MI, s)
  else if (|receivers(ch, MI, s)| =
           |reservationsDone(ch, MI, s)|) then
    TryCompletePerformSend(MI, s)
  else if (shouldTimeout(ch, MI, s) = true) then {
    SetCompleted(MI, s)
    ActivateTimeout(MI, s)
  }
  else
    DoReservations(MI, s)
```

Listing 13: PerformSend

```
rule SetMessageContent(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
    if not(contains(wantInput(ch, MI, ch),
                    "MessageContentDecision")) then
      add "MessageContentDecision" to wantInput(ch, MI, ch)
    else
      skip // waiting for messageContent
```

Listing 14: SetMessageContent

The `SetMessageContent` rule is called if no message content is given as parameter on the communication transition until the `messageContent` function is written by the environment.

```
// handle all receivers
rule DoReservations(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
  let receiversTodo = (receivers(ch, MI, s) diff
                       reservationsDone(ch, MI, s)) in
  foreach receiver in receiversTodo do
    DoReservation(MI, s, receiver)
```

Listing 15: DoReservations

The `DoReservations` rule iterates over all receivers that did not already receive a reservation message. The `DoReservation` rule then tries to place a reservation message for such receiver.

The `DoReservation` rule does not try to place a reservation message if the receiver is *non-proper* terminated. Otherwise it determines the necessary parameters of the queue to use and builds the reservation message. To support sending to an External Subject a translation of the sender's original Subject ID to the Subject ID used in the External Process is performed by the `searchSenderSubjectID` function.

When the queue at `inputPool(rCh, xSID, msgType, dstCorr)` was not created yet a new queue is assigned to that location and remembered in the `inputPoolDefined` function on the receiver's side. If the queue is either closed or full no reservation message can be placed, otherwise it is enqueued at the end.

After all reservations could be placed the `TryCompletePerformSend` rule has to ensure that no receiver terminated *non-proper* in the meantime. In that case the Send Function blocks and can only be left by a Timeout or Cancel transition. Otherwise the state function is completed and the behavior can continue with the transition function.

The transition function calls the `ReplaceReservation` rule to replace all reservations with the actual Message. The `EnsureRunning` rule (re)starts a receiver if it is not already running. If the communication transition has the parameter `messageStoreReceiverVar` defined the used receivers of the Message are stored in that Variable. Also, if the communication transition has the parameter `messageNewCorrelationVar` defined the CorrelationID that was created

```
// handle single reservation
// result true if hasPlacedReservation, adds to reservationsDone
rule DoReservation(MI, currentStateNumber, receiverChannel) =
  if (properTerminated(receiverChannel) = true) then
    let ch = channelFor(self),
       pID = processIDFor(self),
       sID = subjectIDFor(self),
         s = currentStateNumber in
    let Rch = receiverChannel,
       RpID = processIDOf(receiverChannel) in
    let sIDX = searchSenderSubjectID(pID, sID, RpID) in
    let msgCID = messageCorrelationID(ch, MI, s),
        RCID   = messageReceiverCorrelationID(ch, MI, s) in
    // there must be exactly one transition
    let t  = first_outgoingNormalTransition(pID, s) in
    let mT = messageType(pID, t) in
    let resMsg = [ch, mT, {}, msgCID, true] in
      seq
        // prepare receiver IP
        if (inputPool(Rch, sIDX, mT, RCID) = undef) then {
          add [sIDX, mT, RCID] to inputPoolDefined(Rch)
          inputPool(Rch, sIDX, mT, RCID) := []
        }
      next
        if (inputPoolIsClosed(Rch, sIDX, mT, RCID) != true) then
          if (inputPoolGetFreeSpace(Rch, sIDX, mT) > 0) then {
            enqueue resMsg into inputPool(Rch, sIDX, mT, RCID)
            add Rch to reservationsDone(ch, MI, s)
          }
          else
            skip // BLOCKED: no free space!
        else
          skip // BLOCKED: inputPoolIsClosed
  else
    skip // BLOCKED: non-properTerminated
```

Listing 16: DoReservation

in `StartSend` and used for the send messages will be stored in the given Variable.

Analogous to the `DoReservation` rule the `ReplaceReservation` rule has to determine the queue and build both the reservation message and the actual Message. It then can replace the reservation in the queue with the actual message.

To abort the Send Function all placed reservations have to be removed.

Just like the `ReplaceReservation` rule the `CancelReservation` rule determines the location of the queue and rebuilds the reservation message. It then removes the reservation from the queue.

### 3.3.7 Receive Function

The Receive Function retrieves Messages from the Inputpool. The state function updates the enabled outgoing transitions according to the available Messages in

```
rule TryCompletePerformSend(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  if (anyNonProperTerminated(receivers(ch, MI, s)) = true) then
    if (shouldTimeout(ch, MI, s) = true) then {
      SetCompleted(MI, s)
      ActivateTimeout(MI, s)
    }
    else
      // BLOCKED: a receiver where a reservation was placed has
      // terminated non-proper in the meantime
      skip
  else {
    // there must be exactly one transition
    let t = first_outgoingNormalTransition(pID, s) in
      selectedTransition(ch, MI, s) := t

    SetCompleted(MI, s)
  }
```

Listing 17: TryCompletePerformSend

```
rule PerformTransitionSend(MI, currentStateNumber, t) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in {
    foreach r in reservationsDone(ch, MI, s) do {
      ReplaceReservation(MI, s, r)

      EnsureRunning(r)
    }

    let storeVName = messageStoreReceiverVar(pID, t) in
      if (storeVName != undef and storeVName != "") then
        SetVar(MI, storeVName, "ChannelInformation",
               reservationsDone(ch, MI, s))

    let cIDVName = messageNewCorrelationVar(pID, t) in
      if (cIDVName != undef and cIDVName != "") then
        SetVar(MI, cIDVName, "CorrelationID",
               messageCorrelationID(ch, MI, s))

    SetCompletedTransition(MI, s, t)
  }
```

Listing 18: PerformTransitionSend

```
rule ReplaceReservation(MI, currentStateNumber, receiverChannel) =
  let ch = channelFor(self),
     pID = processIDFor(self),
     sID = subjectIDFor(self),
       s = currentStateNumber in
  let Rch  = receiverChannel,
      RpID = processIDOf(receiverChannel) in
  let t  = first_outgoingNormalTransition(pID, s) in
  let mT = messageType(pID, t) in
  let sIDX   = searchSenderSubjectID(pID, sID, RpID),
      msgCID = messageCorrelationID(ch, MI, s),
      RCID   = messageReceiverCorrelationID(ch, MI, s) in
  let resMsg = [ch, mT, {}, msgCID, true],
      msg    = [ch, mT, messageContent(ch, MI, s), msgCID, false],
      IPold = inputPool(Rch, sIDX, mT, RCID) in
  let IPnew = setnth(IPold, head(indexes(IPold, resMsg)), msg) in
      inputPool(Rch, sIDX, mT, RCID) := IPnew
```

Listing 19: ReplaceReservation

```
rule AbortSend(MI, currentStateNumber) =
  let ch = channelFor(self),
       s = currentStateNumber in {
    foreach r in reservationsDone(ch, MI, s) do
      CancelReservation(MI, s, r)

    SetAbortionCompleted(MI, s)
  }
```

Listing 20: AbortSend

```
rule CancelReservation(MI, currentStateNumber, receiverChannel) =
  let ch = channelFor(self),
     pID = processIDFor(self),
     sID = subjectIDFor(self),
       s = currentStateNumber in
  let Rch  = receiverChannel,
      RpID = processIDOf(receiverChannel) in
  let t  = first_outgoingNormalTransition(pID, s) in
  let mT = messageType(pID, t) in
  let sIDX   = searchSenderSubjectID(pID, sID, RpID),
      msgCID = messageCorrelationID(ch, MI, s),
      RCID   = messageReceiverCorrelationID(ch, MI, s) in
  let resMsg = [ch, mT, {}, msgCID, true],
      IPold = inputPool(Rch, sIDX, mT, RCID) in
  let IPnew = dropnth(IPold, head(indexes(IPold, resMsg))) in
    inputPool(Rch, sIDX, mT, RCID) := IPnew
```

Listing 21: CancelReservation

the corresponding queue. When an outgoing transition is selected the Messages are removed from the queue by the transition function.

In the beginning of the state function the Timeout is started and then checked each time. If the Timeout doesn't need to be activated each outgoing transition is either set to be enabled or disabled by the `CheckIP` rule.

```
rule PerformReceive(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  // startTime must be the time of the first attempt to receive
  // in order to support receiving with timeout=0
  if (startTime(ch, MI, s) = undef) then
    StartTimeout(MI, s)
  else if (shouldTimeout(ch, MI, s) = true) then {
    SetCompleted(MI, s)
    ActivateTimeout(MI, s)
  }
  else
    seq
      forall t in outgoingNormalTransitions(pID, s) do
        CheckIP(MI, s, t)
    next
      let enabledT = outgoingEnabledTransitions(ch, MI, s) in
      if (|enabledT| > 0) then
        seq
          if (selectedTransition(ch, MI, s) != undef) then
            skip // there is already an transition selected
          else if (|enabledT| = 1) then
            let t = firstFromSet(enabledT) in
            if (transitionIsAuto(pID, t) = true) then
              // make automatic decision
              selectedTransition(ch, MI, s) := t
            else skip // can not make automatic decision
          else skip // can not make automatic decision
        next
          if (selectedTransition(ch, MI, s) != undef) then
            // the decision was made
            SetCompleted(MI, s)
          else
            // no decision made, waiting for selectedTransition
            SelectTransition(MI, s)
      else
        skip // BLOCKED: no messages
```

Listing 22: PerformReceive

When exactly one auto transition is enabled, and no transition had been selected, an automatic selection for that transition is made. Otherwise a transtion decision from the environment is required.

The `CheckIP` rule loads the parameters from the communication transition attributes to determine the corresponding queue and the available Messages in it. The `messageSubjectCountMin` argument defines the minimal required number of different sender Agents of a MultiSubject. If the optional parameter

```
rule CheckIP(MI, currentStateNumber, t) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  let sID      = messageSubjectId         (pID, t),
      mT       = messageType              (pID, t),
      cIDVName = messageWithCorrelationVar(pID, t),
      countMin = messageSubjectCountMin   (pID, t) in
  let cID  = loadCorrelationID(MI, cIDVName) in
  let msgs = availableMessages(ch, sID, mT, cID) in
    if (|msgs| >= countMin) then
      EnableTransition(MI, t)
    else
      DisableTransition(MI, s, t)
```

Listing 23: CheckIP

`messageWithCorrelationVar` is not set the CorrelationID 0 is used.

When there are sufficient Messages, from different Agents of a MultiSubject, available the transition is enabled, otherwise it is set to be disabled.

```
rule PerformTransitionReceive(MI, currentStateNumber, t) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  let sID      = messageSubjectId         (pID, t),
      mT       = messageType              (pID, t),
      cIDVName = messageWithCorrelationVar(pID, t),
      countMax = messageSubjectCountMax   (pID, t) in
  let cID = loadCorrelationID(MI, cIDVName) in {
    seq
      // stores the messages in receivedMessages
      InputPool_Pop(MI, s, sID, mT, cID, countMax)
    next
      if (messageStoreMessagesVar(pID, t) != undef and
          messageStoreMessagesVar(pID, t) != "") then
        let msgs  = receivedMessages(ch, MI, s),
            vName = messageStoreMessagesVar(pID, t) in
          SetVar(MI, vName, "MessageSet", msgs)

    SetCompletedTransition(MI, s, t)
  }
```

Listing 24: PerformTransitionReceive

The transition function removes the available Messages from the Inputpool and optionally stores them in the Variable given as transition parameter `messageStoreMessagesVar`.

It first determines the location of the queue and passes them to the `InputPool_Pop` rule which removes up to `countMax` of the oldest Messages from the queue at the location `inputPool(ch, xSID, msgType, ipCorr)` and stores them temporarily in the `receivedMessages` function.

### 3.3.8 Modal Split and Modal Join Functions

The ModalSplit Function initiates parallel execution paths that will be joined again in a ModalJoin Function.

```
rule ModalSplit(MI, currentStateNumber, args) =
  let pID = processIDFor(self),
      s = currentStateNumber in {
  // start all following states
  foreach t in outgoingNormalTransitions(pID, s) do
    let sNew = targetStateNumber(pID, t) in
      AddState(MI, s, MI, sNew)

  // remove self
  RemoveState(MI, s, MI, s)
}
```

Listing 25: ModalSplit

It adds the target states of all outgoing transitions to the active states of its Macro Instance and removes itself.

```
// Channel * MacroInstanceNumber * joinState -> Number
function joinCount : LIST * NUMBER * NUMBER -> NUMBER

// number of execution paths have to be provided as argument
rule ModalJoin(MI, currentStateNumber, args) =
  let ch = channelFor(self),
      s = currentStateNumber,
      numSplits = nth(args, 1) in
  seq // count how often this join has been called
    if (joinCount(ch, MI, s) = undef) then
      joinCount(ch, MI, s) := 1
    else
      joinCount(ch, MI, s) := joinCount(ch, MI, s) + 1
  next
    // can we continue, or remove self and will be called again?
    if (joinCount(ch, MI, s) < numSplits) then {
      // drop this execution path
      RemoveState(MI, s, MI, s)
    }
    else {
      // reset for next iteration
      joinCount(ch, MI, s) := undef
      SetCompletedFunction(MI, s, undef)
    }
```

Listing 26: ModalJoin

The ModalJoin Function takes the number of execution paths to join as parameter, which doesn't need to be modelled explicitly as it could be determined when the Process Model is parsed. The `joinCount` function is used to count how many times an execution path was already joined and is incremented each time an execution path leads to this state.

Until all but one execution paths are joined the current state is removed from the list of active states of the current Macro Instance. If the last execution path reached this state the `joinCount` function is reset for the next iteration and the state function is set to completed, so that the Macro Behavior proceeds regularly to the next state.

### 3.3.9 CallMacro Function

The CallMacro Function creates a new Macro Instance for the Macro ID given as first parameter. It is responsible for the evaluation of that Macro Instance and therefore calls the `MacroBehavior` rule with the created Macro Instance ID.

```
// Channel * macroInstanceNumber -> result
function macroTerminationResult : LIST * NUMBER -> ELEMENT

// Channel * macroInstanceNumber -> MacroNumber
function macroNumberOfMI : LIST * NUMBER -> NUMBER

// Channel * macroInstanceNumber * StateNumber -> MacroInstance
function callMacroChildInstance : LIST * NUMBER * NUMBER -> NUMBER
```

Listing 27: macroTerminationResult

The `callMacroChildInstance` function stores the Macro Instance ID of the created Macro Instance. The `macroTerminationResult` function is written, either with the boolean `true` or a string to indicate which outgoing transition of the MacroState should be selected, when the Macro Instance terminates.

The state function has two phases: in the beginning the Macro Instance is created and then in the main phase evaluated.

In the first phase the value of the `nextMacroInstanceNumber` function is stored in the `callMacroChildInstance` function and incremented. The `activeStates` for the new Macro Instance is initialized and the start state will be added later on in the `MacroBehavior` rule of the current Macro Instance.

The CallMacro Function can have an optional list of Variable names whose values are passed into Macro Instance-local Variables of the called Macro Instance with the `InitializeMacroArguments` rule.

The second phase evaluates the created Macro Instance.

When the Macro Instance has terminated its result is used to select the outgoing transition of the CallMacro state and the `callMacroChildInstance` function is reset for the next iteration. Otherwise the `MacroBehavior` rule is called for the created Macro Instance ID.

The `InitializeMacroArguments` rule iterates over all required macro parameters. For each parameter it loads the value of the Variable in the current Macro Instance and stores it in the new Macro Instance.

### 3.3.10 End Function

The End Function is used to terminate the current Macro Instance and has no outgoing transitions. If the current Macro Instance is the Main Macro Instance the End Function terminates the Subject.

The `PerformEnd` rule calls the `AbortMacroInstance` rule until no other states are active in the current Macro Instance.

```
rule CallMacro(MI, currentStateNumber, args) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  let childInstance = callMacroChildInstance(ch, MI, s) in
  if (childInstance = undef) then
    // start new Macro Instance
    let mIDNew = searchMacro(head(args)),
        MINew  = nextMacroInstanceNumber(ch) in
    seqblock
      nextMacroInstanceNumber(ch) := MINew + 1
      macroNumberOfMI(ch, MINew) := mIDNew
      callMacroChildInstance(ch, MI, s) := MINew

      if (|macroArguments(ch, mIDNew)| > 0) then
        InitializeMacroArguments(MI, mIDNew, MINew, tail(args))

      StartMacro(MI, s, mIDNew, MINew)
    endseqblock
  else
    let childResult = macroTerminationResult(ch, childInstance) in
    if (childResult != undef) then {
      callMacroChildInstance(ch, MI, s) := undef

      // transport result, if present
      if (childResult = true) then
        SetCompletedFunction(MI, s, undef)
      else
        SetCompletedFunction(MI, s, childResult)
    }
    else
      // Macro Instance is active, call it
      MacroBehavior(childInstance)
```

Listing 28: CallMacro

```
rule InitializeMacroArguments(MI, mIDNew, MINew, givenSrcVNames) =
  local
      dstVNames := macroArguments(processIDFor(self), mIDNew),
      srcVNames := givenSrcVNames in
    while (|dstVNames| > 0) do {
      let dstVName = head(dstVNames),
          srcVName = head(srcVNames) in
      let var = loadVar(MI, srcVName) in
        SetVar(MINew, dstVName, nth(var, 1), nth(var, 2))

      dstVNames := tail(dstVNames)
      srcVNames := tail(srcVNames)
    }
```

Listing 29: InitializeMacroArguments

```
rule PerformEnd(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  if (|activeStates(ch, MI)| > 1) then
    AbortMacroInstance(MI, s)
  else {
    if (MI = 1) then { // terminate subject
      ClearAllVarInMI(ch, 0)
      ClearAllVarInMI(ch, 1)

      FinalizeInteraction()

      program(self) := undef
      remove self from asmAgents
    }
    else { // terminate only Macro Instance
      ClearAllVarInMI(ch, MI)

      let res = head(stateFunctionArguments(pID, s)) in
      if (res != undef) then
        // use parameter as result for CallMacro State
        macroTerminationResult(ch, MI) := res
      else
        // just indicate termination
        macroTerminationResult(ch, MI) := true
    }

    // remove self
    RemoveState(MI, s, MI, s)
  }
```

Listing 30: PerformEnd

If the current Macro Instance is the Main Macro Instance the Subject terminates. To do so the End Function resets all Variables of the Subject and terminates the ASM agent. Additionally, the `FinalizeInteraction` rule is called ...

Otherwise the Macro Instance was created by a CallMacro Function and only the Variables of the current Macro Instance are reset. If the End Function has a parameter it is stored in the function `macroTerminationResult`, so that the CallMacro Function proceeds with the corresponding outgoing transition. If no parameter is given the value is just set to `true` to indicate a termination without any particular result.

CHAPTER 4

# Implementation of Subject Oriented Models

SUbject oriented models address the internal aspects and structures of a system. They are essentially models of the internal structure of a system and cover organizational and technical aspects. When implementing the models, it is now necessary to establish the relationship between the process model and the available resources. Figure 4.1 shows the individual steps from a process model to the executable process instance.

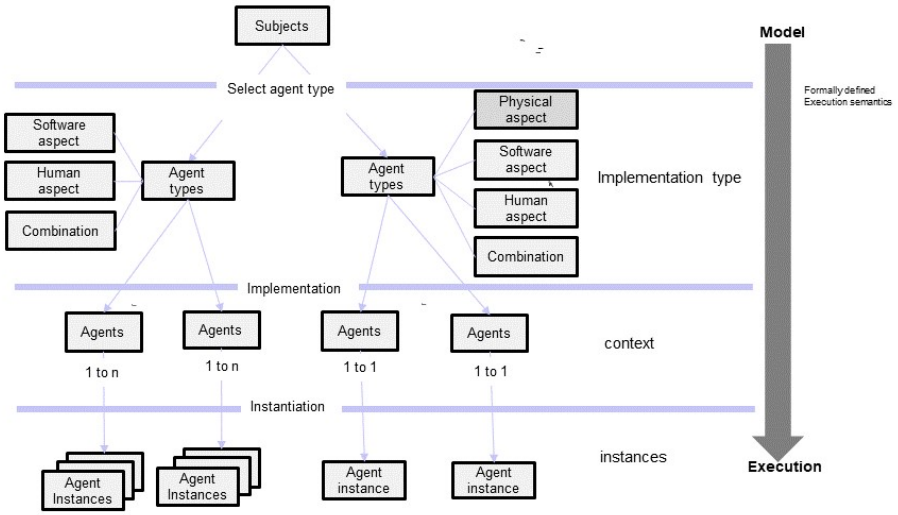


Figure 4.1: Implementation steps

In a system model, the actors, the actions, their sequences and the objects manipulated by the actions are described. Actions (activities) can be performed by humans, software systems, physical systems or a combination of these basic types of actors. We call them the task holders. For example, a software system can automatically perform the "tax rate calculation" action, while a person uses a software program to perform the "order entry" activity. The person enters the order data via a screen mask. The software checks the entered data for plausibility and saves it. However, activities can also be carried out purely manually,

for example when a warehouse worker receives a picking order on paper, executes it, marks it as executed on the order form and returns it to the warehouse manager.
When creating a system model, it is often not yet known which types of actors execute which actions. Therefore, it can be useful to abstract from said model when starting to describe processes by introducing abstract actors. A modeling language should allow the use of such abstractions. This means that when defining the process logic, no assertion should have to be made about what type of actor is realized. In S-BPM, the subjects represent abstract actors.
In the description of the control logic of a process, the individual activities are also described independently of their implementation. For example, for the action "create a picking order" it is not specified whether a human actor fills in a paper form or a screen mask, or whether a software system generates this form automatically. Thus, with activities the means by which something happens is not described, but rather only what happens.
The means are of course related to the implementation type of the actor. As soon as it has been defined which types of actors are assigned to the individual actions, the manner of realization of an activity has also been defined. In addition, the logical or physical object on which an action is executed also needs to be determined. Logical objects are data structures whose data is manipulated by activities. Paper forms represent a mixture between logical and physical objects, while a workpiece on which the "deburring" action takes place is a purely physical object. Therefore, there is a close relationship between the type of task holder, the actions and the associated objects actors manipulate or use when performing actions.
A system model can be used in different areas. The process logic is applied unchanged in the respective areas. However, it may be necessary to implement the individual actors and actions differently. Thus, in one environment certain actions could be performed by humans and in another the same actions could be performed by software systems. In the following, we refer to such different environments of use for a system model as context. Hence, for a process model, varying contexts can exist, in which there are different realization types for actors and actions.
In Subject Oriented Modeling, actors are not assigned to individual activities, but rather the actor type is assigned to an entire subject. This assignment is not part of the process logic, but in the most simple way it is done instead for each process in a separate two-column table. The left column contains the subject name and the right col-umn the implementation type. If there are several contexts for a model, a separate assignment table is created for each of them. The assignment of the implementation type forms the transition between the system logic and its implementation. Subsequently, it has to be defined which persons, software systems and physical systems represent the actors and how the individual actions are concretely realized. These aspects are described in detail in the following subsections.

## 4.1 People and organizations

## 4.2 Physical infrastructure

## 4.3 IT-Systems and Software

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:<br><br><br><br>Range: | PASSProcessModelElement<br><br><br><br>PASSProcessModelElement | Generic ObjectProperty that links two process elements, where one is contained in the other (inverse of contains). | 200 |
| contains | Domain:<br><br><br><br>Range: | PASSProcessModelElement<br><br><br><br>PASSProcessModelElement | Generic ObjectProperty that links two model elements where one contains another (possible multiple) | 201 |
| containsBaseBehavior | Domain:<br>Range: | Subject<br>SubjectBehavior | | 202 |
| containsBehavior | Domain:<br>Range: | Subject<br>SubjectBehavior | | 203 |
| containsPayload-Description | Domain:<br>Range: | MessageSpecification<br>PayloadDescription | | 204 |
| guardedBy | Domain:<br>Range: | State, Action<br>GuardBehavior | | 205 |
| guardsBehavior | Domain:<br><br><br><br><br><br>Range: | GuardBehavior<br><br><br><br><br><br>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:<br>Range: | State, Action<br>guardedBy | | 207 |
| hasAdditionalAttribute | Domain:<br>Range: | PASSProcessModelElement<br>AdditionalAttribute | | 208 |
| hasCorrespondent | Domain:<br><br><br><br><br>Range: | <br><br><br><br><br>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:<br>Range: | DataObjectDefinition | | 210 |
| hasDataMapping-Function | Domain:<br>Range: | state, SendTransition, ReceiveTransition<br>DataMappingFunction | | 211 |
| hasDataType | Domain:<br>Range: | PayloadDescription or DataObjectDefinition<br>DataTypeDefinition | | 212 |
| hasEndState | Domain:<br>Range: | SubjectBehavior or ChoiceSegmentPath<br>State, not SendState | | 213 |
| hasFunction-Specification | Domain:<br>Range: | State<br>FunctionSpecification | | 214 |
| hasHandlingStrategy | Domain:<br>Range: | InputPoolConstraint<br>InputPoolContstraint-HandlingStrategy | | 215 |
| hasIncomingMessage-Exchange | Domain:<br>Range: | Subject<br>MessageExchange | | 216 |
| hasIncomingTransition | Domain:<br>Range: | State<br>Transition | | 217 |
| hasInitialState | Domain:<br>Range: | SubjectBehavior or ChoiceSegmentPath<br>State | | 218 |
| hasInputPoolConstraint | Domain:<br>Range: | Subject<br>InputPoolConstraint | | 219 |
| hasKeyValuePair | Domain:<br>Range: | | | 220 |
| hasMessageExchange | Domain:<br>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:<br><br><br><br>Range: | MessageTypeConstraint or MessageSenderTypeConstraint or MessageExchange<br><br>MessageSpecification | | 222 |
| hasOutgoingMessage-Exchange | Domain:<br><br>Range: | Subject<br><br>MessageExchange | | 223 |
| hasOutgoingTransition | Domain:<br>Range: | State<br>Transition | | 224 |
| hasReceiver | Domain:<br>Range: | MessageExchange<br>Subject | | 225 |
| hasRelationToModel-Component | Domain:<br><br><br><br>Range: | PASSProcessModelElement<br><br><br><br>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:<br>Range: | MessageExchange<br>Subject | | 227 |
| hasSourceState | Domain:<br>Range: | Transition<br>State | | 228 |
| hasStartSubject | Domain:<br>Range: | PASSProcessModel<br>StartSubject | | 229 |
| hasTargetState | Domain<br>Range | Transition<br>State | | 230 |
| hasTransitionCondition | Domain<br>Range | Transition<br>TransitionCondition | | 231 |
| isBaseBehaviorOf | Domain:<br><br><br><br>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:<br>Range: | State and not SendState<br>SubjectBehavior or ChoiceSegmentPath | | 233 |
| isInitialStateOf | Domain:<br>Range: | State<br>SubjectBehavior or ChoiceSegmentPath | | 234 |
| isReferencedBy | Domain:<br>Range: | | | 235 |
| references | Domain:<br>Range: | | | 236 |
| referencesMacroBehavior | Domain:<br>Range: | MacroState<br>MacroBehavior | | 237 |
| refersTo | Domain:<br><br><br><br>Range: | CommunicationTransition<br><br><br><br>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:<br><br>Range: | ReceiveTransitionCondition<br><br>MessageSpecification | | 239 |
| requiresPerformed-MessageExchange | Domain:<br><br>Range: | MessageExchangeCondition<br><br>MessageExchange | | 240 |
| SimplePASSObject-Propertie | Domain:<br><br><br><br>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:<br>Range: | | | |
| hasCalendarBasedFrequencyOrDate | Domain:<br>Range: | | | |
| hasDataMappingString | Domain:<br>Range: | | | |
| hasDayTimeDurationTimeOutTime | Domain:<br>Range: | | | |
| hasDurationTimeOutTime | Domain:<br>Range: | | | |
| hasFeelExpressionAsDataMapping | Domain:<br><br><br><br><br><br><br><br>Range: | | See `https://www.omg.org/spec/DMN` for specification of Feel-Statement-Strings<br>The idea of these expression is to map data fields from and to the internal Data storage of a subject | |
| hasGraphicalRepresentation | Domain:<br><br><br><br><br><br><br><br>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:<br>Range: | | | |
| hasLimit | Domain:<br>Range: | | | |
| hasMaximumSubjectInstanceRestriction | Domain:<br>Range: | | | |
| hasMetaData | Domain:<br>Range: | | | |
| hasModelComponentComment | Domain:<br>Range: | | equivalent to rdfs:comment | |
| hasModelComponentID | Domain:<br><br>Range: | | The unique ID of a PASSProcessModelComponent | |
| hasModelComponentLabel | Domain:<br><br><br>Range: | | The human legible label or description of a model element. | |

| Property name | | Domain-Range | Comments | Reference |
|---|---|---|---|---|
| hasPriorityNumber | Domain: | | Transitions or Behaviors have numbers that denote their execution priority in situations where two or more options could be executed.<br>This is important for automated execution.<br>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.<br>Similarly, SubjectBehaviors with higher priority (lower priority number) are to be executed before Behaviors with lower priority. | |
| | Range: | | | |

| Property name | | Domain-Range | Comments | Reference |
|---|---|---|---|---|
| hasReoccuranceFrequenyOrDate | Domain:<br><br>Range: | | A data field meant for the two classes Reoccurance-TimeOutTransition and ReoccuranceTimeOutExit-Condition.<br>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:<br><br>Range: | | The Scalable Vector Graphic (SVG) XML format is a text based standard to describe vector graphics.<br>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:<br>Range: | | | |

| Property name | | Domain-Range | Comments | Reference |
|---|---|---|---|---|
| hasTimeValue | Domain:<br><br>Range: | | Generic super class for all data properties of time based transitions. | |
| hasToolSpecificDefinition | Domain:<br><br>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:<br>Range: | | | |
| hasYearMonthDurationTimeOutTime | Domain:<br>Range: | | | |
| isOptionalToEndChoiceSegmentPath | Domain:<br>Range: | | | |
| isOptionalToStartChoiceSegmentPath | Domain:<br>Range: | | | |
| owl:topDataProperty | Domain:<br>Range: | | | |
| PASSModelDataProperty | Domain:<br><br>Range: | | Generic super class of all DataProperties that PASS process model elements may have. | |

| Property name | | Domain-Range | Comments | Reference |
|---|---|---|---|---|
| SimplePASSDataProperties | Domain:<br><br><br><br><br>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

# 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 entity and relation names in the ontology to the names used in the ASMs.

## B.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**<br>**SendState**<br>**ReceiveState** | *function state,*<br>*send state,*<br>*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)<br>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**)<br>***DefaultFunctionReceive1_EnvironmentChoice***<br>***DefaultFunctionReceive2_AutoReceiveEarliest***<br>***DefaultFunctionSend*** | *ComAct* | Specialized version of Perform-ASM Rule for communication, either send or receive. These rules distinguish internally between send and receive. |

### B.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**<br>Specialized in:<br>**DoFunction** and<br>**CommunicationActs** with<br>**ReceiveFunction**<br>**SendFunction**<br>There exist a few default activities:<br>**DefaultFunctionDo1_EnvoironmentChoice**<br>**DefaultFunctionDo2_AutomaticEvaluation** | *PERFORM(subj ; service(state); state)* | Main interpreter ASM-rule/Method |
| **CommunicationActs** with<br>**ReceiveFunction**<br>**SendFunction**<br>**DefaultFunctionReceive1_EnvironmentChoice**<br>**DefaultFunctionReceive2_AutoReceiveEarliest**<br>**DefaultFunctionSend** | *PERFORM(subj ;ComAct; state)* | ASM-Rule specifying the execution of a Comunication act in an according state) |

Table B.2: Main Execution/Interpreting Rules

### B.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 B.3: Derived Functions

B.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 B.4: Refinrments places

B.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**<br>And their individual defaultinstances:<br>**InputPoolConstraintStrategy-Blocking**<br>**InputPoolConstraintStrategy-DeleteLatest**<br>**InputPoolConstraintStrategy-DeleteOldest**<br>**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 B.5: Input Pool Handling

B.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 B.6: Other Functions

### B.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 B.7: Other Functions

APPENDIX C

# CoreASM PASS Reference Implementation

## C.1 Conceptual Differences to the OWL Description

This reference implementation has some conceptual differences to the OWL description. For example it provides less flexible InputPool constraints and does not support synchronous communication, however it supports advanced features that are required for distributed inter-company business processes and provides concrete implementations for the usage of Subject Data.

| Concept | Reference Implementation | OWL Description |
|---|---|---|
| InputPool | Exactly one limit has to be defined for each Subject, the value 0 means the InputPool is not limited. The limit applies to each pair of (Sender, MessageType). Only **InputPoolConstraintStrategy-Blocking** is supported, synchronous send & receive is not supported. | Allows exact limits for different Subjects and MessageTypes. The combination **InputPoolConstraintStrategy-Blocking** with the limit of 0 means synchronous send & receive. Additionally defines the strategies DeleteLatest, DeleteOldest and Drop. |
| Subject Restart | Implicit when an additional message arrives on a Subject that is *proper terminated*. | Explicit via **EndState** on a **ReceiveState** and outgoing transitions. |
| Proper Termination | When a Subject terminates and its InputPool is empty it is *proper terminated* and can be restarted. | *Absent*. |
| End State | Implemented as a distinguished state. It cannot be a **ReceiveState** and forbids any outgoing transitions. | Must either be **DoState** or **ReceiveState**. Allows outgoing transitions to restart the Subject. |
| Choice Segments | A Choice Segment begins with a Modal Split state. The paths are joined in a Modal Join state. The target State of every outgoing Transition from the Modal Split State is an **InitialStateOfChoiceSegmentPath**. Every Choice Segment Path is mandatory to start and mandatory to end. | A Choice Segment is a single State. Choice Segment Paths can be both optional to start and optional to end. Choice Segment Paths are not connected by Transitions to the **ChoiceSegment** state. |

Table C.1: Key Conceptual Differences

The most important differences are listed in table C.1: while the OWL description offers both synchronous and asynchronous communication, different InputPool limits and handling strategies this reference implementation adresses only asynchronous communication with a blocking strategy. Conceptionally the End State is not a property of a state, but provides a distinguished function specification to terminate a Subject and determine the *proper terminaion*, which is required for the verification on interaction soundness. Next, the execution of Choice Segment Paths is not controlled by a Choice Segment State, but those paths have to be started with a Modal Split Function and joined in a Modal Join Function.

| Concept | Reference Implementation | OWL Description |
|---|---|---|
| CorrelationID | Concept to correlate messages, e.g. responses to requests, so that messages can be send and received reliable in a loop. | *Absent*. |
| InputPool Functions | InputPool Functions perform operations on the Subject's InputPool, e.g. close it for some message types or sender Subjects. | *Absent*. |
| Timer Transition | The Timer Transition Condition has to be defined in seconds. | It is possible to define the timeout in arbitrary time durations, including in business days. |
| Reminder Transition | *Absent*. | A **ReminderTransition** can be traversed if a certain time based event or frequency has been reached. |
| Mobility of Channels | Concept to store and forward at runtime references to other subjects and their agents. | *Absent*. |
| Macro Exit Parameter | Attribute on the End State of a Macro Behavior that selects a corresponding Transition on the **MacroState**. | *Absent*. |
| Maximum Subject Instances | *Absent*. Every Subject can have an arbitrary amount of instances. | The **hasMaximumSubjectInstanceRestriction** property limits the amount of instances that can be created at runtime of a Subject. |

Table C.2: Further Conceptual Differences

In table C.2 further important conceptual differences are listed: CorrelationIDs define a relation between messages and allow reliable looped communication patterns. CorrelationIDs appear as metadata of a message and as argument to an InputPool queue: A CorrelationID is created when a message is send and part of it. A later message, that relates to it, will be send to the InputPool queue of that CorrelationID, allowing a receive state to specify that only specific messages can be received. With the InputPool Functions queues of the InputPool can be opened and closed, forcing senders to block if they attempt to send to a closed queue. Also, the InputPool can be checked if it is empty or not, so that *non-proper termination* can be avoided. As this reference implementation targets an interactive process model validation only timeouts in seconds are considered. With the Mobility of Channels concept it is possible to store runtime references to agents as Subject Data and to communicate those references with other Subjects, which enables various distributed communication patterns. Macro Exit Parameters allow a Macro State to have multiple outgoing transitions. Within the Macro Behavior the End State determines which outgoing transition will be activated. Limiting the number of Subject instances is not possible, which also means that SingleSubjects are not enforced.

Table C.3 lists differences in Subject Data concepts. Descriptions of the Payload are not supported. The Data Types are hardcoded: "MessageSet" contains a set of Messages, "CorrelationID" contains a single CorrelationID, "ChannelInformation" stores Channels and "Text" is used for arbitrary contents, that are entered by a user in the abstract process evaluation. However, those Data Types are used only at runtime to describe the actual content of a Data Object or a Message.

This reference implementation adds a scope for Data Object Definitions, allowing a Data

Object to be accessible either "globally" for all states in all behaviors of a Subject or only within a single Macro Behavior instance, allowing macros to execute concurrently without influencing each other. Further, this enables copying Data Ojects as arguments from the scope of a Macro State into the scope of the called Macro instance.

While the OWL description merely anticipates Data Mapping Functions there are concrete implementations to use Data Objects in the Send and Receive Functions. Additionally this reference implementation provides Data Modification Functions to perform operations on Data Objects.

| Concept | Reference Implementation | OWL Description |
|---|---|---|
| Payload Description | *Absent*. | A **PayloadDescription** describes a **MessageSpecification** further. |
| Data Types | Hardcoded: MessageSet, CorrelationID, ChannelInformation and Text. The **hasDataType** property on Data Object definitions is ignored. | A **DataTypeDefinition** allows complex definitions, for example XSD Data Types or definitions in OWL or JSON. Data Object definitions and Payload descriptions must have a Data Type specified. |
| Subject Data Scope | A **DataObjectDefinition** can be scoped to either the Subject or a **SubjectBehavior**. When it is scoped to a Behavior its value is accessible only within instances of that Behavior, i.e. within a single instance of a Macro Behavior. When it is scoped to the Subject its value is accessible from every SubjectBehavior instance, given there is no locally scoped Data Object Definition with the same ID. | Is always scoped to the Subject. |
| Macro Data Arguments | A **MacroState** can copy the value of a **DataObjectDefinition** into the scope of another **DataObjectDefinition**, which is scoped to the referenced **MacroBehavior**. | *Absent*. |
| Data Mapping Functions | **StoreMessageFunction** stores the received messages in a local Data Object. **UseMessageContentFunction** uses the value of a Data Object as content of the outgoing message. **UseCorrelationIDFunction** uses the value of a Data Object as CorrelationID of the outgoing message. | *Abstract*. |
| Data Modification Functions | Data Modification Functions perform operations on Subject Data, e.g. concatenation of objects having a set-valued data type. | *Absent*. |

Table C.3: Subject Data Conceptual Differences

As listed in table C.4 a substantial conceptional difference to the OWL description is the lack of a discrete Guard Behavior. This is compensated by the introduction of State Priorities and the Cancel Function, which allows a transformation to an equivalent behavior: The guarded states are modelled in one ChoiceSegmentPath, a second ChoiceSegmentPath uses higher State Priorities and starts with the ReceiveState. Once a message can be received in this second path the execution of the first path is paused and the exception can be handled. After the exception handling the Subject can be terminated with the End State, the paused state can just continue or the paused state can be aborted with the Cancel Function.

| Concept | Reference Implementation | OWL Description |
|---|---|---|
| Cancel Function | Enables the **UserCancelTransition** of a referenced **State**. | *Absent*. |
| State Priority | A utility to support the Observer concept. When multiple states are active only the states with the highest priority can perform. The Function of such states can conditionally allow states with a lower priority to perform. | *Absent*. |
| Observer | Indirectly supported, requires a manual transformation with two **ChoiceSegmentPaths**, where one path starts with a **ReceiveState** and uses higher State Priorities than the other. Returning to the interrupted behavior is implicitly supported, alternatively it can be aborted with the Cancel Function. | Explicitly supported. The **GuardBehavior** has a reference to the observed States or Behaviors. A return to the interrupted state is possible with the **GenericReturnToOriginReference** State. |

Table C.4: Observer Concept Differences

## C.2 EDITORIAL NOTE

To fit the code onto the pages it has been compacted. Instead of writing:

```
rule Behavior(macroInstanceNumber, currentStateNumber) =
  if (initializedState(channelFor(self),
                       macroInstanceNumber,
                       currentStateNumber) != true) then
    StartState(macroInstanceNumber, currentStateNumber)
  else // ...
```

Listing 31: Behavior, spread-out snippet

the code blocks are transformed to a more compact notation:

```
rule Behavior(MI, currentStateNumber) =
 let s = currentStateNumber,
     ch = channelFor(self) in
  if (initializedState(ch, MI, s) != true) then
    StartState(MI, s)
  else // ...
```

Listing 32: Behavior, compact snippet

Some elements are intentionally left out in order to preserve space.

For example, the `rule InputPool_Pop` and the `derived availableMessages` function contain lengthy list traversals, and their definitions contain undocumented parameters used in a test environment for debugging.

Other functions like `derived searchMacro` or `derived hasTimeoutTransition` should be self-explanatory.

## C.3 Basic Definitions

```
function channelFor : Agents -> LIST

derived processIDFor(a)       = processIDOf(channelFor(a))
derived processInstanceFor(a) = processInstanceOf(channelFor(a))
derived subjectIDFor(a)       = subjectIDOf(channelFor(a))
derived agentFor(a)           = agentOf(channelFor(a))

derived processIDOf(ch)       = nth(ch, 1)
derived processInstanceOf(ch) = nth(ch, 2)
derived subjectIDOf(ch)       = nth(ch, 3)
derived agentOf(ch)           = nth(ch, 4)
```

Listing 33: channelFor

```
// Channel -> List[List[MI, StateNumber]]
function killStates : LIST -> LIST

// Channel * macroInstanceNumber -> List[StateNumber]
function activeStates : LIST * NUMBER -> LIST
```

Listing 34: activeStates

```
// -> PI
function nextPI : -> NUMBER
// PI -> Channel
function nextPIUsedBy : NUMBER -> Agents

// Channel -> Number
function nextMacroInstanceNumber : LIST -> NUMBER

// Channel -> Boolean
function properTerminated : LIST -> BOOLEAN

derived anyNonProperTerminated(chs) =
  exists ch in chs with (properTerminated(ch) = false)

// Channel * MacroInstanceNumber * StateNumber -> Set[String]
function wantInput : LIST * NUMBER * NUMBER -> SET
```

Listing 35: nextPI

```
// Channel * MacroInstanceNumber * StateNumber -> BOOLEAN
function initializedState : LIST * NUMBER * NUMBER -> BOOLEAN

// Channel * MacroInstanceNumber * StateNumber -> BOOLEAN
function completed : LIST * NUMBER * NUMBER -> BOOLEAN

// Channel * MacroInstanceNumber * StateNumber
function timeoutActive  : LIST * NUMBER * NUMBER -> BOOLEAN
function cancelDecision : LIST * NUMBER * NUMBER -> BOOLEAN

// Channel * MacroInstanceNumber * StateNumber -> BOOLEAN
function abortionCompleted : LIST * NUMBER * NUMBER -> BOOLEAN

// Channel * MacroInstanceNumber * StateNumber
function selectedTransition : LIST * NUMBER * NUMBER -> NUMBER
function initializedSelectedTransition : LIST * NUMBER * NUMBER
  -> BOOLEAN
function startTime  : LIST * NUMBER * NUMBER -> NUMBER

// Channel * MacroInstanceNumber * TransitionNumber -> BOOLEAN
function transitionEnabled : LIST * NUMBER * NUMBER -> BOOLEAN

// Channel * MacroInstanceNumber * TransitionNumber -> BOOLEAN
function transitionCompleted : LIST * NUMBER * NUMBER -> BOOLEAN
```

Listing 36: initializedState

```
derived shouldTimeout(ch, MI, stateNumber) = return boolres in
  let pID = processIDOf(ch) in
    if (hasTimeoutTransition(pID, stateNumber) = true
        and startTime(ch, MI, stateNumber) != undef) then
      let t = outgoingTimeoutTransition(pID, stateNumber) in
      let timeout = (transitionTimeout(pID, t)
                     * 1000 * 1000 * 1000) in
      let runningTime = (nanoTime()
                         - startTime(ch, MI, stateNumber)) in
        boolres := (runningTime > timeout)
    else
      boolres := false
```

Listing 37: shouldTimeout

```
// Channel * macroInstanceNumber * varname -> [vartype, content]
function variable : LIST * NUMBER * STRING -> LIST

// Channel -> Set[(macroInstanceNumber, varname)]
function variableDefined : LIST -> SET
```

Listing 38: variable

```
// Channel * macroInstanceNumber -> result
function macroTerminationResult : LIST * NUMBER -> ELEMENT

// Channel * macroInstanceNumber -> MacroNumber
function macroNumberOfMI : LIST * NUMBER -> NUMBER

// Channel * macroInstanceNumber * StateNumber -> MacroInstance
function callMacroChildInstance : LIST * NUMBER * NUMBER -> NUMBER
```

Listing 39: macroTerminationResult

## C.4 Interaction Definitions

```
// Channel * senderSubjID * msgType * correlationID
//   -> [msg1, msg2, ...]
function inputPool : LIST * STRING * STRING * NUMBER -> LIST

/* stores all locations where an inputPool was defined */
// Channel -> {[senderSubjID, msgType, correlationID], ...}
function inputPoolDefined : LIST -> SET

// Channel * senderSubjID * msgType * correlationID
function inputPoolClosed : LIST * STRING * STRING * NUMBER
  -> BOOLEAN

derived inputPoolIsClosed(ch, senderSubjID, msgType, cID) =
return boolres in
let isClosed = inputPoolClosed(ch, senderSubjID, msgType, cID) in
  if (isClosed = undef) then // default: global state
    boolres := inputPoolClosed(ch, undef, undef, undef)
  else
    boolres := isClosed
```

Listing 40: inputPool

```
// Channel * MacroInstanceNumber * StateNumber -> Set[Messages]
function receivedMessages : LIST * NUMBER * NUMBER -> SET

// Channel * MacroInstanceNumber * StateNumber -> Set[Channel]
function receivers : LIST * NUMBER * NUMBER -> SET

// Channel * MacroInstanceNumber * StateNumber
function messageContent       : LIST * NUMBER * NUMBER -> LIST
function messageCorrelationID : LIST * NUMBER * NUMBER -> NUMBER
function messageReceiverCorrelationID : LIST * NUMBER * NUMBER
  -> NUMBER

// Channel * MacroInstanceNumber * StateNumber -> Set[Channel]
function reservationsDone : LIST * NUMBER * NUMBER -> SET

function nextCorrelationID : -> NUMBER
function nextCorrelationIDUsedBy : NUMBER -> Agents
```

Listing 41: receivedMessages

C.5 SUBJECT RULES

```
rule GenerateUniqueProcessInstanceID = {
  nextPI := nextPI + 1
  result := nextPI
  // ensure that `nextPI` is not used by multiple agents in the
  // same ASM step. A collision would occur when multiple agents
  // try to call this rule in the same asm step
  nextPIUsedBy(nextPI) := self
}
```

Listing 42: GenerateUniqueProcessInstanceID

```
rule StartProcess(processID, additionalInitializationSubject,
                              additionalInitializationAgent) =
  let pID = processID in
  local PI in
    seq
      PI <- GenerateUniqueProcessInstanceID()
    next {
      result := PI

      foreach sID in keySet(processSubjects(pID)) do
        if (sID = additionalInitializationSubject) then
          let ch = [pID, PI,
                    sID, additionalInitializationAgent] in
            InitializeSubject(ch)
        else if (isStartSubject(pID, sID) = true) then
          let agentSet = safeSet(predefinedAgents(pID, sID)) in
          if (|agentSet| = 1) then
            // shortcut: avoid user interaction
            let agentName = firstFromSet(agentSet) in
            let ch = [pID, PI, sID, agentName] in
              seq
                InitializeSubject(ch)
              next
                StartASMAgent(ch)
          else
            SelectAgentAndStartASMAgent(pID, PI, sID)
    }
```

Listing 43: StartProcess

```
// -> SET[ASMAgent]
function asmAgents : -> SET

derived running(ch) = exists a in asmAgents
                        with channelFor(a) = ch

rule EnsureRunning(ch) =
  if (running(ch) != true) then
    StartASMAgent(ch)

rule StartASMAgent(ch) =
  extend Agents with a do {
    add a to asmAgents
    channelFor(a) := ch
    program(a)    := @StartMainMacro
  }

rule PrepareReceptionOfMessages(ch) =
  // might be called multiple times, esp. via SelectAgents
  if (properTerminated(ch) = undef) then {
    properTerminated(ch) := true

    inputPoolDefined(ch) := {}
    inputPoolClosed(ch, undef, undef, undef) := false
  }

rule FinalizeInteraction =
  let ch = channelFor(self) in
  let proper = inputPoolIsEmpty(ch, undef, undef, undef) in
    properTerminated(ch) := proper

rule InitializeSubject(ch) = PrepareReceptionOfMessages(ch)
```

Listing 44: StartASMAgent

```
rule StartMainMacro =
  let ch = channelFor(self),
      pID = processIDFor(self) in {
    killStates(ch) := []

    let MI = 0 in { // 0 => global / predefined variables
      variableDefined(ch) := {[MI, "$self"], [MI, "$empty"]}
      variable(ch, MI, "$self") := ["ChannelInformation", {ch}]
      variable(ch, MI, "$empty") := ["Text", ""]
    }

    let MI  = 1, // 1 => MainMacro Instance
        mID = subjectMainMacro(pID, subjectIDFor(self)) in
    let startState = macroStartState(pID, mID) in {
      macroNumberOfMI(ch, MI) := mID

      nextMacroInstanceNumber(ch) := MI  + 1

      activeStates(ch, MI) := [startState]
    }

    program(self) := @SubjectBehavior
  }
```

Listing 45: StartMainMacro

```
rule StartMacro(MI, currentStateNumber, mIDNew, MINew) = {
  let pID = processIDFor(self) in
  let startState = macroStartState(pID, mIDNew) in {
    activeStates(channelFor(self), MINew) := []
    AddState(MI, currentStateNumber, MINew, startState)
  }
}
```

Listing 46: StartMacro

```
// called from PerformEnd and AbortCallMacro
rule AbortMacroInstance(MIAbort, ignoreState) =
  let ch = channelFor(self) in {
    foreach currentState in activeStates(ch, MIAbort) do {
      add [MIAbort, currentState] to killStates(ch)
    }

    ClearAllVarInMI(ch, MIAbort)
  }
```

Listing 47: AbortMacroInstance

```
/*
- REPEAT
  - Behavior should be executed again for this state
  - updates from this execution step will be merged with the
    following execution step in one global ASM update
  - no other states are allowed to be executed
- DONE
  - no other active states are allowed to be executed
  - new states are started
  - global ASM / LTS step will be done
- NEXT
  - nothing changed / waiting for input
  - other active states with the same priority can be executed
  - active states with lower priority can not be executed
- LOWER
  - nothing changed / waiting for input
  - other active states, even with lower priority, can be executed
*/

// Channel * MacroInstanceNumber * stateNumber
function executionState : LIST * NUMBER * NUMBER -> NUMBER

// Channel * MacroInstanceNumber
function macroExecutionState : LIST * NUMBER -> NUMBER

// Channel * MacroInstanceNumber * stateNumber -> List[(MI, s)]
function addStates : LIST * NUMBER * NUMBER -> LIST

// Channel * MacroInstanceNumber * stateNumber -> List[(MI, s)]
function removeStates : LIST * NUMBER * NUMBER -> LIST
```

Listing 48: SetExecutionState

```
rule AddState(MI, currentStateNumber, MINew, sNew) =
  add [MINew, sNew]
    to addStates(channelFor(self), MI, currentStateNumber)

rule RemoveState(MI, currentStateNumber, MIOld, sOld) =
  add [MIOld, sOld]
    to removeStates(channelFor(self), MI, currentStateNumber)
```

Listing 49: AddState

```
rule SubjectBehavior =
  choose x in killStates(channelFor(self)) do
    KillBehavior(nth(x, 1), nth(x, 2))
  ifnone
    seqblock
      MacroBehavior(1)
    next // reset
      macroExecutionState(channelFor(self), 1) := undef
```

Listing 50: SubjectBehavior

```
rule KillBehavior(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
  if (initializedState(ch, MI, s) != true) then {
    // state is not initialized,
    // remove without further abortion

    remove [MI, s] from killStates(ch)
    remove s from activeStates(ch, MI)
  }
  else if (abortionCompleted(ch, MI, s) = true) then {
    ResetState(MI, s)
    remove [MI, s] from killStates(ch)
    remove s from activeStates(ch, MI)
  }
  else seqblock
    executionState(ch, MI, s) := undef
    addStates(ch, MI, s)      := []
    removeStates(ch, MI, s)   := []

    // NOTE: no new state must be added.
    // Also, removeStates should be empty,
    // as those states should be added to killStates
    Abort(MI, s)
  endseqblock
```

Listing 51: KillBehavior

```
rule MacroBehavior(MI) =
  let ch = channelFor(self),
     pID = processIDFor(self) in
  local remainingStates := activeStates(ch, MI) in
    seq
      macroExecutionState(ch, MI) := undef
    next
      // NOTE: can not be done with foreach,
      // as remainingStates is modified
      while (|remainingStates| > 0) do
      let stateNumber
        = getAnyStateWithHighestPrio(pID, remainingStates) in
      seqblock
        executionState(ch, MI, stateNumber) := undef
        addStates(ch, MI, stateNumber)      := []
        removeStates(ch, MI, stateNumber)   := []

        Behavior(MI, stateNumber)

        // NOTE: mutates remainingStates!
        let state = executionState(ch, MI, stateNumber) in
          UpdateRemainingStates(MI, stateNumber, state, remainingStates)

        UpdateActiveStates(MI, stateNumber)
      endseqblock
```

Listing 52: MacroBehavior

```
rule UpdateRemainingStates(MI, stateNumber,
                           exState, remainingStates) =
  let ch = channelFor(self),
      pID = processIDFor(self) in
  if (exState = REPEAT) then {
    remainingStates := [stateNumber]

    macroExecutionState(ch, MI) := DONE

    // reset
    executionState(ch, MI, stateNumber) := undef
  }
  else if (exState = DONE) then {
    seq // end loop ...
      remainingStates := []
    next
      // ... but add new states of this MI to initialize them,
      // so that all states have the same start time
      foreach x in addStates(ch, MI, stateNumber)
        with nth(x, 1) = MI do {
        add nth(x, 2) to remainingStates
      }

    macroExecutionState(ch, MI) := DONE

    // quasi-reset
    executionState(ch, MI, stateNumber) := NEXT
  }
  else if (exState = NEXT) then {
    seq
      remove stateNumber from remainingStates // remove self
    next
      // reduce to states with same priority
      let prio = statePriority(pID, stateNumber) in
        remainingStates
          := filterStatesWithSamePrio(pID, remainingStates, prio)

    if (macroExecutionState(ch, MI) != DONE) then
      macroExecutionState(ch, MI) := NEXT
  }
  else if (exState = LOWER) then {
    remove stateNumber from remainingStates // remove self

    if (macroExecutionState(ch, MI) != DONE
        and macroExecutionState(ch, MI) != NEXT) then
      macroExecutionState(ch, MI) := LOWER
  }
```

Listing 53: UpdateRemainingStates

```
rule UpdateActiveStates(MI, stateNumber) =
  // NOTE: everything needs to be sequential,
  // as activeStates is a list and not a set
  let ch = channelFor(self) in seqblock
    foreach x in addStates(ch, MI, stateNumber) do
      let xMI = nth(x, 1),
          xN  = nth(x, 2) in {
        add xN to activeStates(ch, xMI)
      }

    addStates(ch, MI, stateNumber) := undef

    foreach x in removeStates(ch, MI, stateNumber) do
      let xMI = nth(x, 1),
          xN  = nth(x, 2) in {
        // remove one instance of xN, if any
        remove xN from activeStates(ch, xMI)

        ResetState(xMI, xN)
      }

    removeStates(ch, MI, stateNumber) := undef
  endseqblock
```

Listing 54: UpdateActiveStates

```
// whether the state should be aborted or not
derived abortState(MI, stateNumber) =
  ((timeoutActive(channelFor(self), MI, stateNumber) = true) or
  (cancelDecision(channelFor(self), MI, stateNumber) = true))
```

Listing 55: abortState

```
rule Behavior(MI, currentStateNumber) =
 let s = currentStateNumber,
    ch = channelFor(self) in
  if (initializedState(ch, MI, s) != true) then
    StartState(MI, s)
  else if (abortState(MI, s) = true) then
    AbortState(MI, s)
  else if (completed(ch, MI, s) != true) then
    Perform(MI, s)
  else if (initializedSelectedTransition(ch, MI, s) != true) then
    StartSelectedTransition(MI, s)
  else
   let t = selectedTransition(ch, MI, s) in
    if (transitionCompleted(ch, MI, t) != true) then
      PerformTransition(MI, s, t)
    else {
      Proceed(MI, s, targetStateNumber(processIDFor(self), t))
      SetExecutionState(MI, s, DONE)
    }
```

Listing 56: Behavior

```
rule AbortState(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  if (abortionCompleted(ch, MI, s) != true) then
    Abort(MI, s)
  else {
    if (cancelDecision(ch, MI, s) = true) then {
      let t = outgoingCancelTransition(pID, s) in
      let target = targetStateNumber(pID, t) in {
        Proceed(MI, s, target)
      }
    }
    else if (timeoutActive(ch, MI, s) = true) then {
      let t = outgoingTimeoutTransition(pID, s) in
      let target = targetStateNumber(pID, t) in {
        Proceed(MI, s, target)
      }
    }

    SetExecutionState(MI, s, DONE)
  }
```

Listing 57: AbortState

C.6 State Rules

```
rule StartState(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in seqblock
    InitializeCompletion(MI, s)
    abortionCompleted(ch, MI, s) := false

    ResetTimeout(MI, s)
    cancelDecision(ch, MI, s) := false

    DisableAllTransitions(MI, s)
    initializedSelectedTransition(ch, MI, s) := false

    wantInput(ch, MI, s) := {}

    case stateType(pID, s) of
      "function"       : StartFunction(MI, s)
      "internalAction" : StartInternalAction(MI, s)
      "send"           : StartSend(MI, s)
      "receive"        : SetExecutionState(MI, s, LOWER)

      // shortcut: directly to PerformEnd w/o ASM step
      "end"            : SetExecutionState(MI, s, REPEAT)
    endcase

    initializedState(ch, MI, s) := true
  endseqblock
```

Listing 58: StartState

```
rule ResetState(MI, stateNumber) =
  let ch = channelFor(self),
      s = stateNumber in {
    executionState(ch, MI, s) := undef

    initializedState(ch, MI, s) := undef

    completed(ch, MI, s) := undef
    abortionCompleted(ch, MI, s) := undef

    startTime(ch, MI, s) := undef
    timeoutActive(ch, MI, s) := undef

    cancelDecision(ch, MI, s) := undef

    selectedTransition(ch, MI, s) := undef

    let pID = processIDFor(self) in
    forall t in outgoingNormalTransitions(pID, s) do {
      transitionEnabled(ch, MI, t) := undef
      transitionCompleted(ch, MI, t) := undef
    }

    initializedSelectedTransition(ch, MI, s) := undef

    wantInput(ch, MI, s) := undef

    // from StartSend
    receivers                    (ch, MI, s) := undef
    reservationsDone             (ch, MI, s) := undef
    messageContent               (ch, MI, s) := undef
    messageCorrelationID         (ch, MI, s) := undef
    messageReceiverCorrelationID(ch, MI, s) := undef
  }
```

Listing 59: ResetState

```
rule Perform(MI, currentStateNumber) =
  let pID = processIDFor(self)
      s = currentStateNumber in
  case stateType(pID, s) of
    "function"       : PerformFunction(MI, s)
    "internalAction" : PerformInternalAction(MI, s)
    "send"           : PerformSend(MI, s)
    "receive"        : PerformReceive(MI, s)
    "end"            : PerformEnd(MI, s)
  endcase
```

Listing 60: Perform

```
rule SelectTransition(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
  if (|outgoingEnabledTransitions(ch, MI, s)| = 0) then
    // BLOCKED: none to select
    SetExecutionState(MI, s, NEXT)
  else if (not(contains(wantInput(ch, MI, s),
                        "TransitionDecision"))) then {
    add "TransitionDecision" to wantInput(ch, MI, s)
    SetExecutionState(MI, s, DONE)
  }
  else
    // waiting for selectedTransition
    SetExecutionState(MI, s, NEXT)
```

Listing 61: SelectTransition

```
rule StartSelectedTransition(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in {
    let t = selectedTransition(ch, MI, s) in {
      InitializeCompletionTransition(MI, t)
      initializedSelectedTransition(ch, MI, s) := true
    }

    SetExecutionState(MI, s, REPEAT)
  }
```

Listing 62: StartSelectedTransition

```
rule PerformTransition(MI, currentStateNumber, t) =
  let pID = processIDFor(self),
      s = currentStateNumber in
  case stateType(pID, s) of
    "function"       : PerformTransitionFunction(MI, s, t)
    "internalAction" : SetCompletedTransition(MI, s, t)
    "send"           : PerformTransitionSend(MI, s, t)
    "receive"        : PerformTransitionReceive(MI, s, t)
  endcase
```

Listing 63: PerformTransition

```
rule StartFunction(MI, currentStateNumber) =
  let pID = processIDFor(self) in
  let functionName = stateFunction(pID, currentStateNumber) in {
    StartTimeout(MI, currentStateNumber)

    if (startFunction(functionName) = undef) then
      skip
    else
      call startFunction(functionName) (MI, currentStateNumber)

    SetExecutionState(MI, currentStateNumber, LOWER)
  }
```

Listing 64: StartFunction

```
rule PerformFunction(MI, currentStateNumber) =
  let s = currentStateNumber in
    if (shouldTimeout(channelFor(self), MI, s) = true) then {
      SetCompleted(MI, s)
      ActivateTimeout(MI, s)
    }
    else
      let pID = processIDFor(self) in
      let functionName = stateFunction(pID, s),
          args         = stateFunctionArguments(pID, s) in
        call performFunction(functionName) (MI, s, args)
```

Listing 65: PerformFunction

```
rule AbortFunction(MI, currentStateNumber) =
  let pID = processIDFor(self) in
  let functionName = stateFunction(pID, currentStateNumber) in
    if (abortFunction(functionName) = undef) then
      SetAbortionCompleted(MI, currentStateNumber)
    else
      // must set abortionCompleted eventually
      call abortFunction(functionName) (MI, currentStateNumber)
```

Listing 66: AbortFunction

```
rule PerformTransitionFunction(MI, currentStateNumber, t) =
  let pID = processIDFor(self),
        s = currentStateNumber in
  let functionName = stateFunction(pID, s) in
    if (performTransitionFunction(functionName) = undef) then
      SetCompletedTransition(MI, s, t)
    else
      call performTransitionFunction(functionName) (MI, s, t)
```

Listing 67: PerformTransitionFunction

```
rule SetCompletedFunction(MI, currentStateNumber, res) =
  let ch = channelFor(self),
       s = currentStateNumber in {
    if (res = undef) then
      choose t in outgoingNormalTransitions(pID, s) do
        selectedTransition(ch, MI, s) := t
    else
      let t = getTransitionByLabel(pID, s, res) in
        selectedTransition(ch, MI, s) := t

    SetCompleted(MI, s)
  }
```

Listing 68: SetCompletedFunction

```
rule Proceed(MI, s_from, s_to) = {
  AddState(MI, s_from, MI, s_to)
  RemoveState(MI, s_from, MI, s_from)
}
```

Listing 69: Proceed

```
rule StartTimeout(MI, currentStateNumber) =
  let ch = channelFor(self) in {
    startTime(ch, MI, currentStateNumber) := nanoTime()
    timeoutActive(ch, MI, currentStateNumber) := false
  }

rule ResetTimeout(MI, currentStateNumber) =
  let ch = channelFor(self) in {
    startTime(ch, MI, currentStateNumber) := undef
    timeoutActive(ch, MI, currentStateNumber) := undef
  }

rule ActivateTimeout(MI, currentStateNumber) =
  let ch = channelFor(self) in
    timeoutActive(ch, MI, currentStateNumber) := true
```

Listing 70: StartTimeout

```
rule InitializeCompletion(MI, currentStateNumber) =
  let ch = channelFor(self) in
    completed(ch, MI, currentStateNumber) := false

rule SetCompleted(MI, currentStateNumber) =
  let ch = channelFor(self) in {
    SetExecutionState(MI, currentStateNumber, REPEAT)
    completed(ch, MI, currentStateNumber) := true
  }

rule SetAbortionCompleted(MI, currentStateNumber) =
  let ch = channelFor(self) in {
    SetExecutionState(MI, currentStateNumber, DONE)
    abortionCompleted(ch, MI, currentStateNumber) := true
  }
```

Listing 71: InitializeCompletion

```
rule EnableTransition(MI, t) =
  transitionEnabled(channelFor(self), MI, t) := true

rule EnableAllTransitions(MI, currentStateNumber) =
  let pID = processIDFor(self),
      s = currentStateNumber in
    forall t in outgoingNormalTransitions(pID, s) do
      EnableTransition(MI, t)

rule DisableTransition(MI, currentStateNumber, t) =
  transitionEnabled(channelFor(self), MI, t) := false

rule DisableAllTransitions(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in {
    forall t in outgoingNormalTransitions(ppID, s) do
      DisableTransition(MI, s, t)

    selectedTransition(ch, MI, s) := undef
  }
```

Listing 72: EnableTransition

```
rule InitializeCompletionTransition(MI, t) =
  transitionCompleted(channelFor(self), MI, t) := false

rule SetCompletedTransition(MI, currentStateNumber, t) = {
  SetExecutionState(MI, currentStateNumber, REPEAT)

  transitionCompleted(channelFor(self), MI, t) := true
}
```

Listing 73: InitializeCompletionTransition

### C.7 Internal Action

```
rule StartInternalAction(MI, currentStateNumber) = {
  StartTimeout(MI, currentStateNumber)

  EnableAllTransitions(MI, currentStateNumber)

  SetExecutionState(MI, currentStateNumber, LOWER)
}
```

Listing 74: StartInternalAction

```
rule PerformInternalAction(MI, currentStateNumber) =
  let ch = channelFor(self),
       s = currentStateNumber in
    if (shouldTimeout(ch, MI, s) = true) then {
      SetCompleted(MI, s)
      ActivateTimeout(MI, s)
    }
    else if (selectedTransition(ch, MI, s) != undef) then
      SetCompleted(MI, s)
    else
      SelectTransition(MI, s)
```

Listing 75: PerformInternalAction

## C.8 Send Function

```
rule StartSend(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  // there must be exactly one transition
  let  t = first_outgoingNormalTransition(pID, s) in {
    receivers(ch, MI, s) := undef
    reservationsDone(ch, MI, s) := {}
    let mcVName = messageContentVar(pID, t) in
      messageContent(ch, MI, s) := loadVar(MI, mcVName)

    // generate new CorrelationID now, it will be stored
    // in a Variable once the message(s) are send
    let cIDVName = messageNewCorrelationVar(pID, t) in
      if (cIDVName != undef and cIDVName != "") then {
        messageCorrelationID(ch, MI, s) := nextCorrelationID
        nextCorrelationID := nextCorrelationID + 1
        // ensure no other agent uses this same correlationID
        nextCorrelationIDUsedBy(nextCorrelationID) := self
      }
      else
        messageCorrelationID(ch, MI, s) := 0

    // load receiver IP CorrelationID now, to avoid
    // influences of any changes of that variable
    let cIDVName = messageWithCorrelationVar(pID, t) in
    let cID = loadCorrelationID(MI, cIDVName) in
      messageReceiverCorrelationID(ch, MI, s) := cID

    SetExecutionState(MI, s, LOWER)
  }
```

Listing 76: StartSend

```
rule PerformSend(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
  if (receivers(ch, MI, s) = undef) then
    SelectReceivers(MI, s)
  else if (messageContent(ch, MI, s) = undef) then
    SetMessageContent(MI, s)
  else if (startTime(ch, MI, s) = undef) then {
    StartTimeout(MI, s)
    SetExecutionState(MI, s, REPEAT)
  }
  else if (|receivers(ch, MI, s)| =
           |reservationsDone(ch, MI, s)|) then
    TryCompletePerformSend(MI, s)
  else if (shouldTimeout(ch, MI, s) = true) then {
    SetCompleted(MI, s)
    ActivateTimeout(MI, s)
  }
  else
    DoReservations(MI, s)
```

Listing 77: PerformSend

```
rule TryCompletePerformSend(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  if (anyNonProperTerminated(receivers(ch, MI, s)) = true) then
    // BLOCKED: a receiver where a reservation was placed has
    // terminated non-proper in the meantime
    if (shouldTimeout(ch, MI, s) = true) then {
      SetCompleted(MI, s)
      ActivateTimeout(MI, s)
    }
    else
      SetExecutionState(MI, s, NEXT)
  else {
    // there must be exactly one transition
    let t = first_outgoingNormalTransition(pID, s) in
      selectedTransition(ch, MI, s) := t

    SetCompleted(MI, s)
  }
```

Listing 78: TryCompletePerformSend

```
rule SelectReceivers(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  // there must be exactly one transition
  let t = first_outgoingNormalTransition(pID, s) in
  let min = messageSubjectCountMin(pID, t),   // 0 => ALL
      max = messageSubjectCountMax(pID, t) in // 0 => no limit
  let recVName = messageSubjectVar(pID, t),
      recSID   = messageSubjectId(pID, t) in
    if (recVName != undef and recVName != "") then
      let rChs = loadChannelsFromVariable(MI, recVName, recSID) in
        if (|rChs| = 0 or (min != 0 and |rChs| < min)) then
          // BLOCKED: not enough receivers given
          SetExecutionState(MI, s, NEXT)
        else if (max != 0 and |rChs| > max)  then
          // too many receivers given -> call VarMan-Selection
          SelectReceivers_Selection(MI, s, rChs, min, max)
        else {
          // receivers fit min/max -> use them
          receivers(ch, MI, s) := rChs
          SetExecutionState(MI, s, REPEAT)
        }
    else // no variable with receivers -> call SelectAgents
      if (selectAgentsResult(ch, MI, s) != undef) then
        let y = selectAgentsResult(ch, MI, s) in {
          receivers(ch, MI, s) := y
          selectAgentsResult(ch, MI, s) := undef // reset

          SetExecutionState(MI, s, REPEAT)
        }
      else
        SelectAgents(MI, s, recSID, min, max)
```

Listing 79: SelectReceivers

```
rule SelectReceivers_Selection(MI, currentStateNumber,
                               rChs, min, max) =
  let ch = channelFor(self),
       s = currentStateNumber in
  let res = selectionResult(ch, MI, s) in
  if (res = undef) then
    let src = ["ChannelInformation", rChs] in
      Selection(MI, s, src, min, max)
  else {
    selectionResult(ch, MI, s) := undef
    receivers(ch, MI, s) := res

    SetExecutionState(MI, s, REPEAT)
  }
```

Listing 80: SelectReceivers_Selection

```
rule AbortSend(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in {
    foreach r in reservationsDone(ch, MI, s) do {
      CancelReservation(MI, s, r)
    }

    SetAbortionCompleted(MI, s)
  }
```

Listing 81: AbortSend

```
rule PerformTransitionSend(MI, currentStateNumber, t) =
  let ch = channelFor(self),
     pID = processIDFor(self),
      s = currentStateNumber in {
    foreach r in reservationsDone(ch, MI, s) do {
      ReplaceReservation(MI, s, r)

      EnsureRunning(r)
    }

    let storeVName = messageStoreReceiverVar(pID, t) in
      if (storeVName != undef and storeVName != "") then
        SetVar(MI, storeVName, "ChannelInformation",
               reservationsDone(ch, MI, s))

    let cIDVName = messageNewCorrelationVar(pID, t) in
      if (cIDVName != undef and cIDVName != "") then
        SetVar(MI, cIDVName, "CorrelationID",
               messageCorrelationID(ch, MI, s))

    SetCompletedTransition(MI, s, t)
  }
```

Listing 82: PerformTransitionSend

```
rule SetMessageContent(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
    if not(contains(wantInput(ch, MI, ch),
                    "MessageContentDecision")) then {
      add "MessageContentDecision" to wantInput(ch, MI, ch)
      SetExecutionState(MI, ch, DONE)
    }
    else
      // waiting for messageContent
      SetExecutionState(MI, ch, NEXT)
```

Listing 83: SetMessageContent

```
// handle all receivers
rule DoReservations(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
  local hasPlacedReservation := false in
    seq
      let receiversTodo = (receivers(ch, MI, s) diff
                           reservationsDone(ch, MI, s)) in
      foreach receiver in receiversTodo do
        local tmp := false in
          seq
            // result is true if a reservation was made
            tmp <- DoReservation(MI, s, receiver)
          next if (tmp = true) then
            hasPlacedReservation := true
    next
      if (hasPlacedReservation = true) then
        // reservation(s) placed, make update
        SetExecutionState(MI, s, DONE)
      else
        // no reservations made, allow other states
        SetExecutionState(MI, s, NEXT)
```

Listing 84: DoReservations

```
// handle single reservation
// result true if hasPlacedReservation, adds to reservationsDone
rule DoReservation(MI, currentStateNumber, receiverChannel) =
  if (properTerminated(receiverChannel) = true) then
    let ch = channelFor(self),
       pID = processIDFor(self),
       sID = subjectIDFor(self),
         s = currentStateNumber in
    let Rch = receiverChannel,
       RpID = processIDOf(receiverChannel) in
    let sIDX = searchSenderSubjectID(pID, sID, RpID) in
    let msgCID = messageCorrelationID(ch, MI, s),
        RCID   = messageReceiverCorrelationID(ch, MI, s) in
    // there must be exactly one transition
    let t  = first_outgoingNormalTransition(pID, s) in
    let mT = messageType(pID, t) in
    let resMsg = [ch, mT, {}, msgCID, true] in
      seq
        // prepare receiver IP
        if (inputPool(Rch, sIDX, mT, RCID) = undef) then {
          add [sIDX, mT, RCID] to inputPoolDefined(Rch)
          inputPool(Rch, sIDX, mT, RCID) := []
        }
      next
        if (inputPoolIsClosed(Rch, sIDX, mT, RCID) != true) then
          if (inputPoolGetFreeSpace(Rch, sIDX, mT) > 0) then {
            enqueue resMsg into inputPool(Rch, sIDX, mT, RCID)
            add Rch to reservationsDone(ch, MI, s)
            result := true
          }
          else
            result := false // BLOCKED: no free space!
        else
          result := false // BLOCKED: inputPoolIsClosed
  else
    result := false // BLOCKED: non-properTerminated
```

Listing 85: DoReservation

```
rule CancelReservation(MI, currentStateNumber, receiverChannel) =
  let ch = channelFor(self),
     pID = processIDFor(self),
     sID = subjectIDFor(self),
       s = currentStateNumber in
  let Rch  = receiverChannel,
      RpID = processIDOf(receiverChannel) in
  let t  = first_outgoingNormalTransition(pID, s) in
  let mT = messageType(pID, t) in
  let sIDX   = searchSenderSubjectID(pID, sID, RpID),
      msgCID = messageCorrelationID(ch, MI, s),
      RCID   = messageReceiverCorrelationID(ch, MI, s) in
  let resMsg = [ch, mT, {}, msgCID, true],
      IPold = inputPool(Rch, sIDX, mT, RCID) in
  let IPnew = dropnth(IPold, head(indexes(IPold, resMsg))) in
    inputPool(Rch, sIDX, mT, RCID) := IPnew
```

Listing 86: CancelReservation

```
rule ReplaceReservation(MI, currentStateNumber, receiverChannel) =
  let ch = channelFor(self),
      pID = processIDFor(self),
      sID = subjectIDFor(self),
        s = currentStateNumber in
  let Rch  = receiverChannel,
      RpID = processIDOf(receiverChannel) in
  let t  = first_outgoingNormalTransition(pID, s) in
  let mT = messageType(pID, t) in
  let sIDX   = searchSenderSubjectID(pID, sID, RpID),
      msgCID = messageCorrelationID(ch, MI, s),
      RCID   = messageReceiverCorrelationID(ch, MI, s) in
  let resMsg = [ch, mT, {}, msgCID, true],
      msg    = [ch, mT, messageContent(ch, MI, s), msgCID, false],
      IPold = inputPool(Rch, sIDX, mT, RCID) in
  let IPnew = setnth(IPold, head(indexes(IPold, resMsg)), msg) in
      inputPool(Rch, sIDX, mT, RCID) := IPnew
```

Listing 87: ReplaceReservation

## C.9 Receive Function

```
rule PerformReceive(MI, currentStateNumber) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  // startTime must be the time of the first attempt to receive
  // in order to support receiving with timeout=0
  if (startTime(ch, MI, s) = undef) then {
    StartTimeout(MI, s)
    SetExecutionState(MI, s, REPEAT)
  }
  else if (shouldTimeout(ch, MI, s) = true) then {
    SetCompleted(MI, s)
    ActivateTimeout(MI, s)
  }
  else
    seq
      forall t in outgoingNormalTransitions(pID, s) do
        CheckIP(MI, s, t)
    next
      let enabledT = outgoingEnabledTransitions(ch, MI, s) in
      if (|enabledT| > 0) then
        seq
          if (selectedTransition(ch, MI, s) != undef) then
            skip // there is already an transition selected
          else if (|enabledT| = 1) then
            let t = firstFromSet(enabledT) in
            if (transitionIsAuto(pID, t) = true) then
              // make automatic decision
              selectedTransition(ch, MI, s) := t
            else skip // can not make automatic decision
          else skip // can not make automatic decision
        next
          if (selectedTransition(ch, MI, s) != undef) then
            // the decision was made
            SetCompleted(MI, s)
          else
            // no decision made, waiting for selectedTransition
            SelectTransition(MI, s)
      else
        SetExecutionState(MI, s, LOWER) // BLOCKED: no messages
```

Listing 88: PerformReceive

```
rule PerformTransitionReceive(MI, currentStateNumber, t) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  let sID      = messageSubjectId         (pID, t),
      mT       = messageType              (pID, t),
      cIDVName = messageWithCorrelationVar(pID, t),
      countMax = messageSubjectCountMax   (pID, t) in
  let cID = loadCorrelationID(MI, cIDVName) in {
    seq
      // stores the messages in receivedMessages
      InputPool_Pop(MI, s, sID, mT, cID, countMax)
    next
      if (messageStoreMessagesVar(pID, t) != undef and
          messageStoreMessagesVar(pID, t) != "") then
        let msgs  = receivedMessages(ch, MI, s),
            vName = messageStoreMessagesVar(pID, t) in
          SetVar(MI, vName, "MessageSet", msgs)

    SetCompletedTransition(MI, s, t)
  }
```

Listing 89: PerformTransitionReceive

```
rule CheckIP(MI, currentStateNumber, t) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  let sID      = messageSubjectId         (pID, t),
      mT       = messageType              (pID, t),
      cIDVName = messageWithCorrelationVar(pID, t),
      countMin = messageSubjectCountMin   (pID, t) in
  let cID  = loadCorrelationID(MI, cIDVName) in
  let msgs = availableMessages(ch, sID, mT, cID) in
    if (|msgs| >= countMin) then
      EnableTransition(MI, t)
    else
      DisableTransition(MI, s, t)
```

Listing 90: CheckIP

### C.10 End Function

```
rule PerformEnd(MI, currentStateNumber) =
  let ch = channelFor(self),
      pID = processIDFor(self),
        s = currentStateNumber in
  if (|activeStates(ch, MI)| > 1) then {
    // does not remove currentStateNumber
    AbortMacroInstance(MI, s)
    SetExecutionState(MI, s, DONE)
  }
  else {
    if (MI = 1) then { // terminate subject
      ClearAllVarInMI(ch, 0)
      ClearAllVarInMI(ch, 1)

      FinalizeInteraction()

      program(self) := undef
      remove self from asmAgents
    }
    else { // terminate only Macro Instance
      ClearAllVarInMI(ch, MI)

      let res = head(stateFunctionArguments(pID, s)) in
      if (res != undef) then
        // use parameter as result for CallMacro State
        macroTerminationResult(ch, MI) := res
      else
        // just indicate termination
        macroTerminationResult(ch, MI) := true
    }

    // remove self
    RemoveState(MI, s, MI, s)
    SetExecutionState(MI, s, DONE)
  }
```

Listing 91: PerformEnd

## C.11 Tau Function

```
rule StartTau(MI, currentStateNumber) =
  EnableAllTransitions(MI, currentStateNumber)

rule Tau(MI, currentStateNumber, args) =
  let ch = channelFor(self),
      pID = processIDFor(self),
        s = currentStateNumber in
    choose t in outgoingEnabledTransitions(t, MI, s)
        with (transitionIsAuto(pID, t) = true) do {
      selectedTransition(t, MI, s) := t
      SetCompleted(MI, s)
    }
    ifnone // WARN: unable to choose auto transition!
      if (selectedTransition(t, MI, s) != undef) then
        SetCompleted(MI, s)
      else
        SelectTransition(MI, s)
```

Listing 92: Tau

## C.12 VarMan Function

```
rule AbortVarMan(MI, currentStateNumber) = {
  ResetSelection(MI, currentStateNumber)
  SetAbortionCompleted(MI, currentStateNumber)
}

rule VarMan(MI, currentStateNumber, args) =
 let s = currentStateNumber,
     method = nth(args, 1),
     A = nth(args, 2),
     B = nth(args, 3),
     C = nth(args, 4),
     D = nth(args, 5) in
 case method of
  "assign"              : VarMan_Assign   (MI, s, A, B)
  "storeData"           : VarMan_StoreData(MI, s, A, B)
  "clear"               : VarMan_Clear    (MI, s, A)

  "concatenation"       : VarMan_Concatenation(MI, s, A, B, C)
  "intersection"        : VarMan_Intersection (MI, s, A, B, C)
  "difference"          : VarMan_Difference   (MI, s, A, B, C)

  "extractContent"      : VarMan_ExtractContent      (MI, s, A, B)
  "extractChannel"      : VarMan_ExtractChannel      (MI, s, A, B)
  "extractCorrelationID": VarMan_ExtractCorrelationID(MI, s, A, B)

  "selection"           : VarMan_Selection(MI, s, A, B, C, D)
 endcase
```

Listing 93: VarMan

```
rule VarMan_Assign(MI, currentStateNumber, A, X) =
  let a = loadVar(MI, A) in {
    SetVar(MI, X, head(a), last(a))

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

rule VarMan_StoreData(MI, currentStateNumber, X, A) = {
  SetVar(MI, X, "Data", A)

  SetCompletedFunction(MI, currentStateNumber, undef)
}

rule VarMan_Clear(MI, currentStateNumber, X) = {
  ClearVar(MI, X)

  SetCompletedFunction(MI, currentStateNumber, undef)
}
```

Listing 94: VarMan_Assign

```
rule VarMan_Concatenation(MI, currentStateNumber, A, B, X) =
  let a = loadVar(MI, A),
      b = loadVar(MI, B) in {
    if (a = undef and b = undef) then
      ClearVar(MI, X)
    else if (a = undef) then
      SetVar(MI, X, head(b), last(b))
    else if (b = undef) then
      SetVar(MI, X, head(a), last(a))
    else
      let x = (last(a) union last(b)) in
        SetVar(MI, X, head(a), x)

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

rule VarMan_Intersection(MI, currentStateNumber, A, B, X) =
  let a = loadVar(MI, A),
      b = loadVar(MI, B) in {
    if (a = undef or b = undef) then
      ClearVar(MI, X)
    else
      let x = (last(a) intersect last(b)) in
        SetVar(MI, X, head(a), x)

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

rule VarMan_Difference(MI, currentStateNumber, A, B, X) =
  let a = loadVar(MI, A),
      b = loadVar(MI, B) in {
    if (a = undef) then
      ClearVar(MI, X)
    else if (b = undef) then
      SetVar(MI, X, head(a), last(a))
    else
      let x = (last(a) diff last(b)) in
        SetVar(MI, X, head(a), x)

    SetCompletedFunction(MI, currentStateNumber, undef)
  }
```

Listing 95: VarMan_Concatenation

```
rule VarMan_ExtractContent(MI, currentStateNumber, A, X) =
  let a = loadVar(MI, A) in
  let msgs = last(a) in
  let msgsContent = map(msgs, @msgContent) in
  let varType = head(firstFromSet(msgsContent)) in {
    SetVar(MI, X, varType, flattenSet(map(msgsContent, @last)))

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

rule VarMan_ExtractChannel(MI, currentStateNumber, A, X) =
  let a = loadVar(MI, A) in
  let msgs = last(a) in
  let msgsChannel = map(msgs, @msgChannel) in {
    SetVar(MI, X, "ChannelInformation", msgsChannel)

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

rule VarMan_ExtractCorrelationID(MI, currentStateNumber, A, X) =
  let a = loadVar(MI, A) in
  let msgs = last(a) in
  let msgsCorrelationID = map(msgs, @msgCorrelation) in {
    SetVar(MI, X, "CorrelationID",
               firstFromSet(msgsCorrelationID))

    SetCompletedFunction(MI, currentStateNumber, undef)
  }
```

Listing 96: VarMan_ExtractContent

```
// Channel * MI * n
function selectionVartype  : LIST * NUMBER * NUMBER -> STRING
function selectionData     : LIST * NUMBER * NUMBER -> LIST
function selectionOptions  : LIST * NUMBER * NUMBER -> LIST
function selectionMin      : LIST * NUMBER * NUMBER -> NUMBER
function selectionMax      : LIST * NUMBER * NUMBER -> NUMBER
function selectionDecision : LIST * NUMBER * NUMBER -> SET

function selectionResult   : LIST * NUMBER * NUMBER -> SET

rule VarMan_Selection(MI, currentStateNumber, srcVName, dstVName,
                                                minimum, maximum) =
  let  ch = channelFor(self) in
  let src = loadVar(MI, srcVName),
      res = selectionResult(ch, MI, currentStateNumber) in
  if (res = undef) then
    Selection(MI, currentStateNumber, src, minimum, maximum)
  else {
    selectionResult(ch, MI, currentStateNumber) := undef

    SetVar(MI, dstVName, head(src), res)

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

rule ResetSelection(MI, currentStateNumber) =
  let ch = channelFor(self),
       s = currentStateNumber in {
  selectionVartype (ch, MI, s) := undef
  selectionData    (ch, MI, s) := undef
  selectionOptions (ch, MI, s) := undef
  selectionMin     (ch, MI, s) := undef
  selectionMax     (ch, MI, s) := undef
  selectionDecision(ch, MI, s) := undef
}
```

Listing 97: VarMan_Selection

```
rule Selection(MI, currentStateNumber, src, minimum, maximum) =
  let ch = channelFor(self),
      s = currentStateNumber in
  if (selectionData(ch, MI, s) = undef) then {
    let l = toList(last(src)) in
    if (head(src) = "MessageSet") then {
      selectionData   (ch, MI, s) := l
      selectionOptions(ch, MI, s) := map(l, @msgToString)
    }
    else if (head(src) = "ChannelInformation") then {
      selectionData   (ch, MI, s) := l
      selectionOptions(ch, MI, s) := map(l, @chToString)
    }

    selectionVartype (ch, MI, s) := head(src)
    selectionMin     (ch, MI, s) := minimum
    selectionMax     (ch, MI, s) := maximum
    selectionDecision(ch, MI, s) := undef

    SetExecutionState(MI, s, REPEAT)
  }
  else if (selectionDecision(ch, MI, s) = undef) then {
    if not(contains(wantInput(ch, MI, s),
                    "SelectionDecision")) then {
      add "SelectionDecision" to wantInput(ch, MI, s)

      SetExecutionState(MI, s, DONE)
    }
    else
      // waiting for selectionDecision
      SetExecutionState(MI, s, NEXT)
  }
  else {
    let res = pickItems(selectionData(ch, MI, s),
                        selectionDecision(ch, MI, s)) in
      selectionResult(ch, MI, s) := res

    ResetSelection(MI, s)
    SetExecutionState(MI, s, REPEAT)
  }
```

Listing 98: Selection

## C.13 Modal Split & Modal Join Functions

```
rule ModalSplit(MI, currentStateNumber, args) =
  let pID = processIDFor(self),
      s = currentStateNumber in {
  // start all following states
  foreach t in outgoingNormalTransitions(pID, s) do
    let sNew = targetStateNumber(pID, t) in
      AddState(MI, s, MI, sNew)

  // remove self
  RemoveState(MI, s, MI, s)

  SetExecutionState(MI, s, DONE)
}
```

Listing 99: ModalSplit

```
// Channel * MacroInstanceNumber * joinState -> Number
function joinCount : LIST * NUMBER * NUMBER -> NUMBER

// number of execution paths have to be provided as argument
rule ModalJoin(MI, currentStateNumber, args) =
  let ch = channelFor(self),
      s = currentStateNumber,
      numSplits = nth(args, 1) in
  seq // count how often this join has been called
    if (joinCount(ch, MI, s) = undef) then
      joinCount(ch, MI, s) := 1
    else
      joinCount(ch, MI, s) := joinCount(ch, MI, s) + 1
  next
    // can we continue, or remove self and will be called again?
    if (joinCount(ch, MI, s) < numSplits) then {
      // drop this execution path
      RemoveState(MI, s, MI, s)
      SetExecutionState(MI, s, DONE)
    }
    else {
      // reset for next iteration
      joinCount(ch, MI, s) := undef
      SetCompletedFunction(MI, s, undef)
    }
```

Listing 100: ModalJoin

### C.14 CallMacro Function

```
rule AbortCallMacro(MI, currentStateNumber) =
  let ch = channelFor(self),
      s = currentStateNumber in
  let childInstance = callMacroChildInstance(ch, MI, s) in
    if (|activeStates(ch, childInstance)| > 0) then {
      AbortMacroInstance(childInstance, undef)
      SetExecutionState(MI, s, DONE)
    }
    else {
      callMacroChildInstance(ch, MI, s) := undef
      SetAbortionCompleted(MI, s)
    }
```

Listing 101: AbortCallMacro

```
rule InitializeMacroArguments(MI, mIDNew, MINew, givenSrcVNames) =
  local
      dstVNames := macroArguments(processIDFor(self), mIDNew),
      srcVNames := givenSrcVNames in
    while (|dstVNames| > 0) do {
      let dstVName = head(dstVNames),
          srcVName = head(srcVNames) in
      let var = loadVar(MI, srcVName) in
        SetVar(MINew, dstVName, nth(var, 1), nth(var, 2))

      dstVNames := tail(dstVNames)
      srcVNames := tail(srcVNames)
    }
```

Listing 102: InitializeMacroArguments

```
rule CallMacro(MI, currentStateNumber, args) =
  let ch = channelFor(self),
      pID = processIDFor(self),
        s = currentStateNumber in
  let childInstance = callMacroChildInstance(ch, MI, s) in
  if (childInstance = undef) then
    // start new Macro Instance
    let mIDNew = searchMacro(head(args)),
        MINew  = nextMacroInstanceNumber(ch) in
    seqblock
      nextMacroInstanceNumber(ch) := MINew + 1
      macroNumberOfMI(ch, MINew) := mIDNew
      callMacroChildInstance(ch, MI, s) := MINew

      if (|macroArguments(ch, mIDNew)| > 0) then
        InitializeMacroArguments(MI, mIDNew, MINew, tail(args))

      SetExecutionState(MI, s, DONE)

      StartMacro(MI, s, mIDNew, MINew)
    endseqblock
  else
    // perform existing Macro Instance
    let childResult = macroTerminationResult(ch, childInstance) in
    if (childResult != undef) then {
      callMacroChildInstance(ch, MI, s) := undef

      // transport result, if present
      if (childResult = true) then
        SetCompletedFunction(MI, s, undef)
      else
        SetCompletedFunction(MI, s, childResult)
    }
    else seqblock
      // Macro Instance is active, call it ...
      MacroBehavior(childInstance)

      // ... and transport its execution state
      let mState = macroExecutionState(ch, childInstance) in
        SetExecutionState(MI, s, mState)

      // reset
      macroExecutionState(ch, childInstance) := undef
    endseqblock
```

Listing 103: CallMacro

### C.15 Cancel Function

```
rule CheckCancel(MI, currentStateNumber, t) =
  let ch = channelFor(self),
     pID = processIDFor(self) in
  let tName = transitionLabel(pID, t) in
  let nCancel = stateNumberFromID(pID, tName) in
  if (contains(activeStates(ch, MI), nCancel) = true) then
    // referenced state is active in this Macro Instance
    EnableTransition(MI, t)
  else
    DisableTransition(MI, currentStateNumber, t)
```

Listing 104: CheckCancel

```
rule Cancel(MI, currentStateNumber, args) =
  let ch = channelFor(self),
     pID = processIDFor(self),
       s = currentStateNumber in
  seq
    forall t in outgoingNormalTransitions(pID, s) do
      CheckCancel(MI, s, t)
  next
    let enabledT = outgoingEnabledTransitions(ch, MI, s) in
    if (|enabledT| > 0) then
      seq
        if (|enabledT| = 1) then
          let t = firstFromSet(enabledT) in
          if (transitionIsAuto(pID, t) = true) then
            selectedTransition(ch, MI, s) := t
      next
        if (selectedTransition(ch, MI, s) != undef) then
          let t = selectedTransition(ch, MI, s) in
            SetCompletedFunction(MI, s, transitionLabel(pID, t))
        else
          SelectTransition(MI, s)
    else // BLOCKED: no corresponding active states
      SetExecutionState(MI, s, LOWER)
```

Listing 105: Cancel

```
rule Abort(MI, currentStateNumber) =
  let pID = processIDFor(self),
       s = currentStateNumber in
    case stateType(pID, s) of
      "function"       : AbortFunction(MI, s)
      "internalAction" : SetAbortionCompleted(MI, s)
      "send"           : AbortSend(MI, s)
      "receive"        : SetAbortionCompleted(MI, s)
    endcase
```

Listing 106: Abort

```
rule PerformTransitionCancel(MI, currentStateNumber, t) =
  let ch = channelFor(self),
      pID = processIDFor(self),
        s = currentStateNumber in
    let tLabel  = transitionLabel(pID, t) in
    let nCancel = stateNumberFromID(pID, tLabel) in {
      cancelDecision(ch, MI, nCancel) := true
      SetCompletedTransition(MI, s, t)
    }
```

Listing 107: PerformTransitionCancel

C.16 IP FUNCTIONS

```
rule CloseIP(MI, currentStateNumber, args) =
  let ch = channelFor(self),
      senderSubjID = nth(args, 1),
      msgType      = nth(args, 2),
      cIDVName     = nth(args, 3) in
  let cID = loadCorrelationID(MI, cIDVName) in {
    inputPoolClosed(ch, senderSubjID, msgType, cID) := true

    if (inputPool(ch, senderSubjID, msgType, cID) = undef) then {
      add [senderSubjID, msgType, cID] to inputPoolDefined(ch)
      inputPool(ch, senderSubjID, msgType, cID) := []
    }

    SetCompletedFunction(MI, currentStateNumber, undef)
  }
```

Listing 108: CloseIP

```
rule OpenIP(MI, currentStateNumber, args) =
  let ch = channelFor(self),
      senderSubjID = nth(args, 1),
      msgType      = nth(args, 2),
      cIDVName     = nth(args, 3) in
  let cID = loadCorrelationID(MI, cIDVName) in {
    inputPoolClosed(ch, senderSubjID, msgType, cID) := false

    if (inputPool(ch, senderSubjID, msgType, cID) = undef) then {
        add [senderSubjID, msgType, cID] to inputPoolDefined(ch)
        inputPool(ch, senderSubjID, msgType, cID) := []
    }

    SetCompletedFunction(MI, currentStateNumber, undef)
  }
```

Listing 109: OpenIP

```
rule CloseAllIPs(MI, currentStateNumber, args) =
  let ch = channelFor(self),
       s = currentStateNumber in {
    inputPoolClosed(ch, undef, undef, undef) := true

    forall key in inputPoolDefined(ch) do
      let sID = nth(key, 1),
          mT  = nth(key, 2),
          cID = nth(key, 3) in {
        inputPoolClosed(ch, sID, mT, cID) := true
      }

    SetCompletedFunction(MI, s, undef)
  }
```

Listing 110: CloseAllIPs

```
rule OpenAllIPs(MI, currentStateNumber, args) =
  let ch = channelFor(self),
       s = currentStateNumber in {
    inputPoolClosed(ch, undef, undef, undef) := false

    forall key in inputPoolDefined(ch) do
      let sID = nth(key, 1),
          mT  = nth(key, 2),
          cID = nth(key, 3) in {
        inputPoolClosed(ch, sID, mT, cID) := false
      }

    SetCompletedFunction(MI, s, undef)
  }
```

Listing 111: OpenAllIPs

```
// correlation can be wildcard (*)
rule IsIPEmpty(MI, currentStateNumber, args) =
  let ch = channelFor(self),
       s = currentStateNumber,
     senderSubjID       = nth(args, 1),
     msgType            = nth(args, 2),
     correlationIDVName = nth(args, 3) in
  local cID in
    seq
      if (correlationIDVName = undef or
          correlationIDVName = 0 or
          correlationIDVName = "") then {
        cID := 0
      }
      else if (correlationIDVName = "*") then {
        cID := undef
      }
      else {
        cID := loadCorrelationID(MI, correlationIDVName)
      }
    next
      if inputPoolIsEmpty(ch, senderSubjID, msgType, cID) then
        SetCompletedFunction(MI, s, "true")
      else
        SetCompletedFunction(MI, s, "false")
```

Listing 112: IsIPEmpty

## C.17 SelectAgents Function

```
// Channel * MacroInstanceNumber * StateNumber -> BOOLEAN
function selectAgentsDecision : LIST * NUMBER * NUMBER -> SET

function selectAgentsProcessID : LIST * NUMBER * NUMBER -> STRING
function selectAgentsSubjectID : LIST * NUMBER * NUMBER -> STRING
function selectAgentsCountMin  : LIST * NUMBER * NUMBER -> NUMBER
function selectAgentsCountMax  : LIST * NUMBER * NUMBER -> NUMBER

function selectAgentsResult : LIST * NUMBER * NUMBER -> SET

rule SelectAgentsAction(MI, currentStateNumber, args) =
  let ch = channelFor(self),
       s = currentStateNumber,
     vName    = nth(args, 1),
     sIDLocal = nth(args, 2),
     countMin = nth(args, 3),
     countMax = nth(args, 4) in
    if (selectAgentsResult(ch, MI, s) != undef) then {
      SetVar(MI, vName, "ChannelInformation",
             selectAgentsResult(ch, MI, s))
      selectAgentsResult(ch, MI, s) := undef

      SetCompletedFunction(MI, s, undef)
    }
    else
      SelectAgents(MI, s, sIDLocal, countMin, countMax)
```

Listing 113: SelectAgentsAction

```
rule SelectAgents(MI, currentStateNumber, sIDLocal, min, max) =
  let ch = channelFor(self),
      pID = processIDFor(self),
        s = currentStateNumber,
       PI = processInstanceFor(self) in
  let predefAgents = predefinedAgents(pID, sIDLocal),
      resolvedInterface = resolveInterfaceSubject(sIDLocal) in
  let resolvedProcessID = nth(resolvedInterface, 1),
      resolvedSubjectID = nth(resolvedInterface, 2) in
  if (selectAgentsDecision(ch, MI, s) != undef) then {
      local createdChannels := {} in
      seq
        foreach agent in selectAgentsDecision(ch, MI, s) do
          if (resolvedProcessID = pID) then
            // local process, use own PI
            let ch = [pID, PI, sIDLocal, agent] in {
              InitializeSubject(ch)
              add ch to createdChannels
            }
          else // external process, create new PI
            local newPI in
            seq
              newPI <- StartProcess(resolvedProcessID,
                                    resolvedSubjectID, agent)
            next
              add [resolvedProcessID, newPI,
                   resolvedSubjectID, agent] to createdChannels
      next
        selectAgentsResult(ch, MI, s) := createdChannels

      // reset for next iteration
      selectAgentsDecision (ch, MI, s) := undef
      selectAgentsCountMin (ch, MI, s) := undef
      selectAgentsCountMax (ch, MI, s) := undef
      selectAgentsProcessID(ch, MI, s) := undef
      selectAgentsSubjectID(ch, MI, s) := undef

      SetExecutionState(MI, s, REPEAT)
  }
  else if(hasSizeWithin(predefAgents, min, max) = true) then {
    selectAgentsDecision(ch, MI, s) := predefAgents
    SetExecutionState(MI, s, REPEAT)
  }
  else if not(contains(wantInput(ch, MI, s),
                       "SelectAgentsDecision")) then {
    add "SelectAgentsDecision" to wantInput(ch, MI, s)

    selectAgentsProcessID(ch, MI, s) := resolvedProcessID
    selectAgentsSubjectID(ch, MI, s) := resolvedSubjectID
    selectAgentsCountMin (ch, MI, s) := min
    selectAgentsCountMax (ch, MI, s) := max
    selectAgentsResult   (ch, MI, s) := undef

    SetExecutionState(MI, s, DONE)
  }
  else // waiting for selectAgentsDecision
    SetExecutionState(MI, s, NEXT)
```

Listing 114: SelectAgents

# Bibliography

[Kee76] E. L. Keenan. 1976.