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

Reinhard Gniza
xyz

Herbert Kindermann
xyz

Florian Krenn
xyz

Werner Schmidt
xyz

Robert Singer
FH JOANNEUM–University of Applied Sciences

Christian Stary
xyz

Florian Strecker
xyz

André Wolski
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.

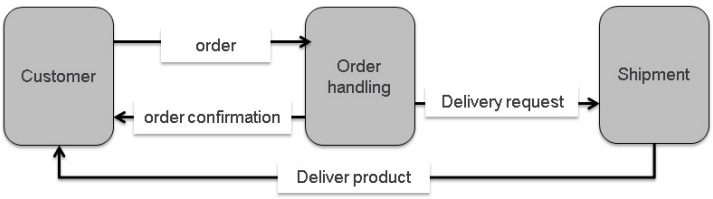

Figure 1.1: The Communication Structure in the Order Process

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.

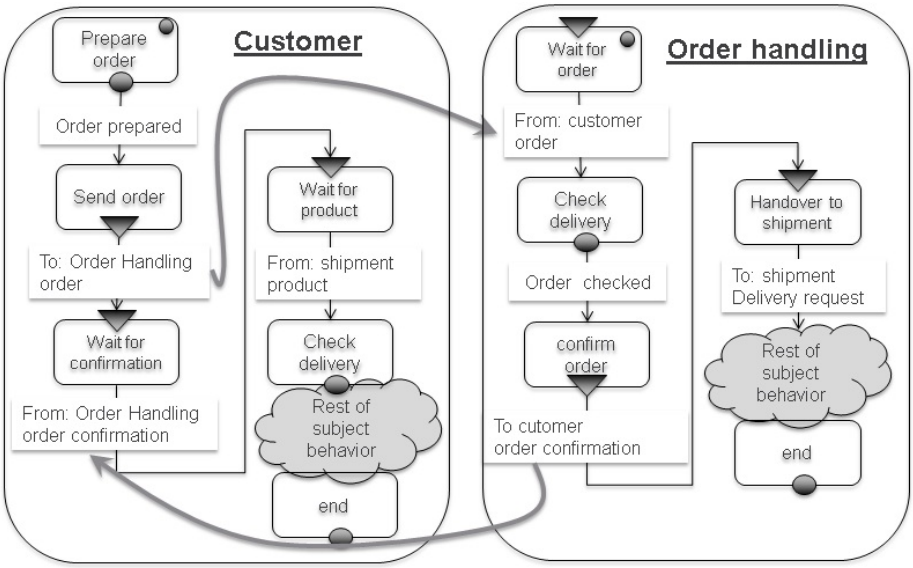

Figure 1.2: The Behavior of Subjects

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

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

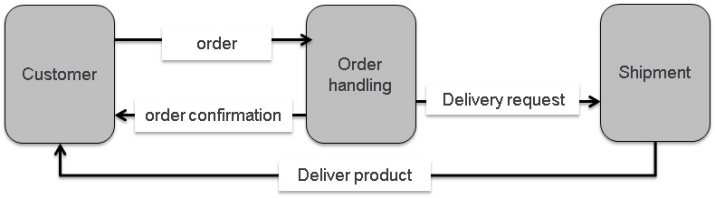

Figure 1.3: The Communication Structure with Change Message

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

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

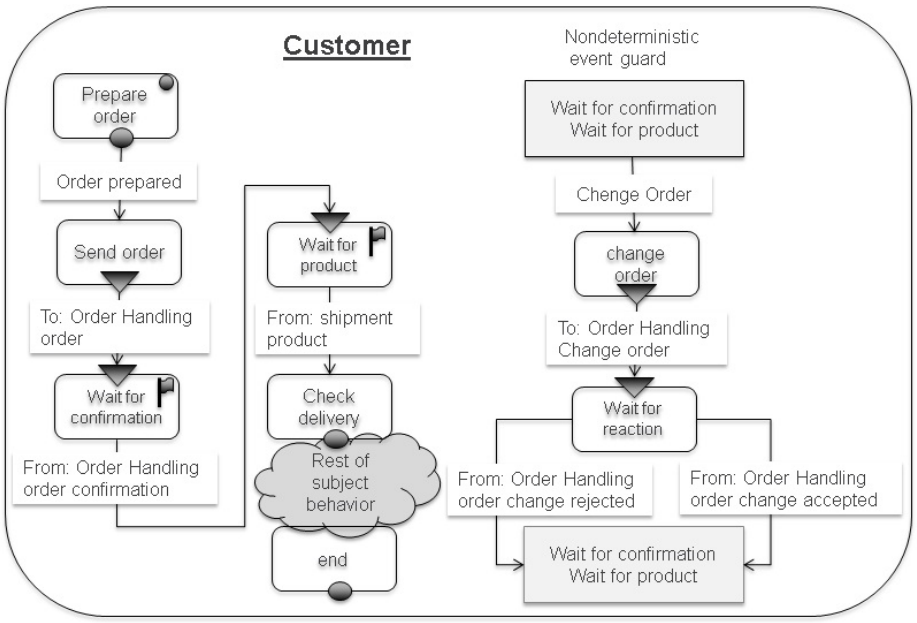

Figure 1.4: Customer is allowed to Change Orders

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

1.1.3 Subjects and Objects

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

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

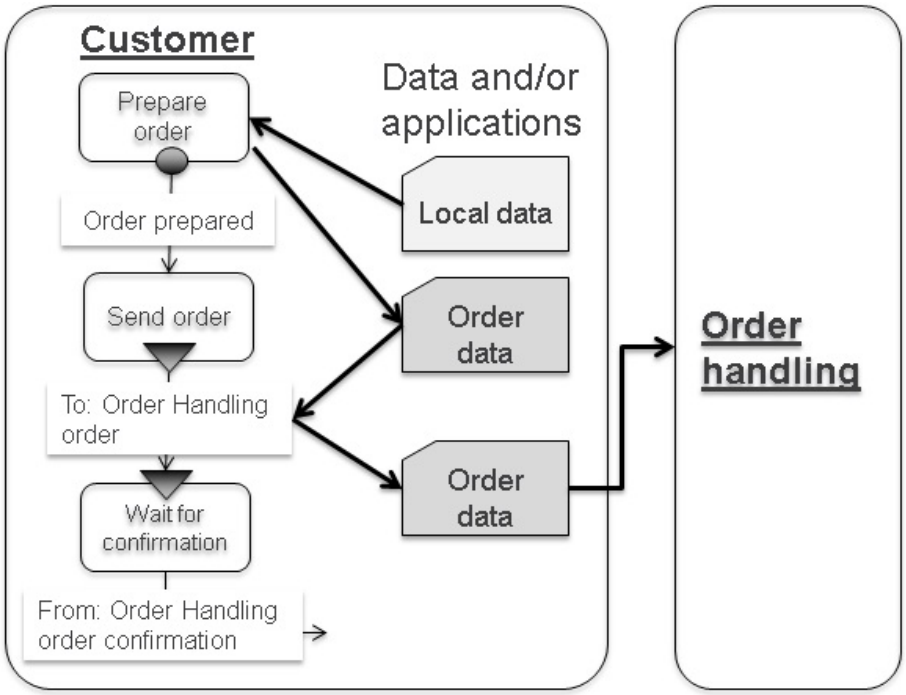

Figure 1.5: Subjects and Objects

1.2 Introduction to Ontologies and OWL

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

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

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

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

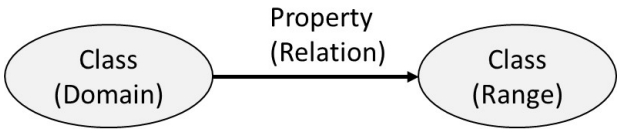

Figure 1.6: RDF graphic

RDF is based on the idea of making statements about resources (in particular web resources) in expressions of the form subject–predicate–object, known as

triples. The subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. In the context of ontology, the term subject expresses what the sentence is about (topic) (see 1.1).

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

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

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

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

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

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

Ontology statement examples:

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

1.3 Introduction to Abstract State Machines

An abstract state machine (ASM) is a state machine operating on states that are arbitrary data structures (structure in the sense of mathematical logic, that is a

nonempty set together with several functions (operations) and relations over the set).

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

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

with arbitrary *Condition* and *ACTION*. The latter is usually a finite set of assignments of the form *f (t1, ..., tn) := t*. The meaning of such a rule is to perform in any given state the indicated action if the indicated condition holds in this state.

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

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

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

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

instead of the equivalent ASM with two rules:

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

Another notation used below is

```
let x=t in M
```

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

For details of a mathematical definition of the semantics of ASMs which justifies their intuitive (rule-based or pseudo-code) understanding, we refer the reader to the AsmBook Börger, E., Stärk R. Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, 2003.

CHAPTER 2

Structure of a PASS Description

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

2.1 Informal Description

2.1.1 Subject

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

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

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

In general, there are the following types of subjects:

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

Fully specified Subjects

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

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

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

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

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

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

Multsubjects and Multiprocesses

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

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

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

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

Single subjects

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

Interface Subjects

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

2.1.2 Subject-to-Subject Communication

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

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

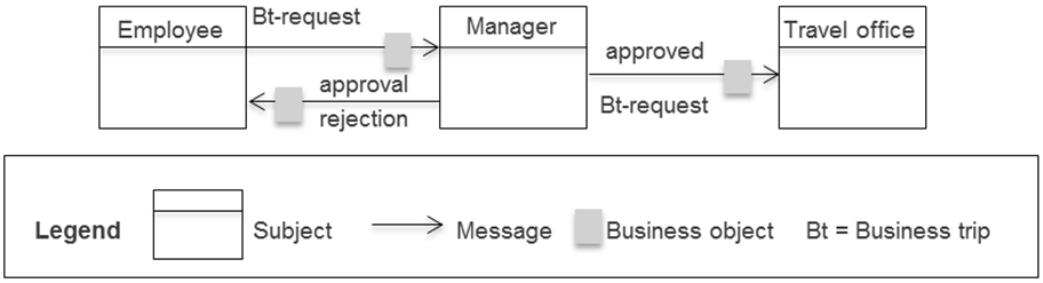

Figure 2.1: Subject interaction diagram for the process 'business trip application'

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

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

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

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

2.1.3 Message Exchange

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

Synchronous and Asynchronous Exchange of Messages

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

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

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

Exchange of Messages via the Input Pool

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

Consider the input pool as a mailbox of work performers, the operation of which is specified in detail. Each subject has its input pool. It serves as a message buffer to temporarily store messages received by the subject, independent of the sending communication partner. The input pools are therefore inboxes for flexible configuration of the message exchange between the subjects. In contrast

to the buffer in which only the front message can be seen and accepted, the pool solution enables picking up (i.e. removing from the buffer) any message. For a subject, all messages in its input pool are visible.

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

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

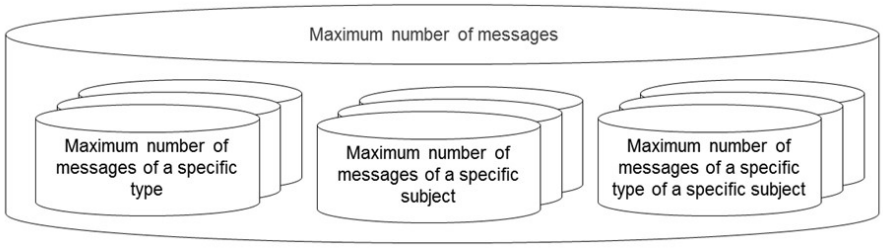

Figure 2.2: Configuration of Input Pool Parameters

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

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

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

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

2.2 OWL Description

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

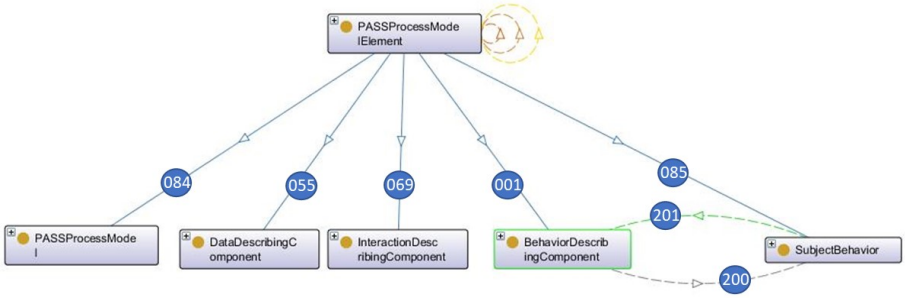

Figure 2.3: Elements of PASS Process Models

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

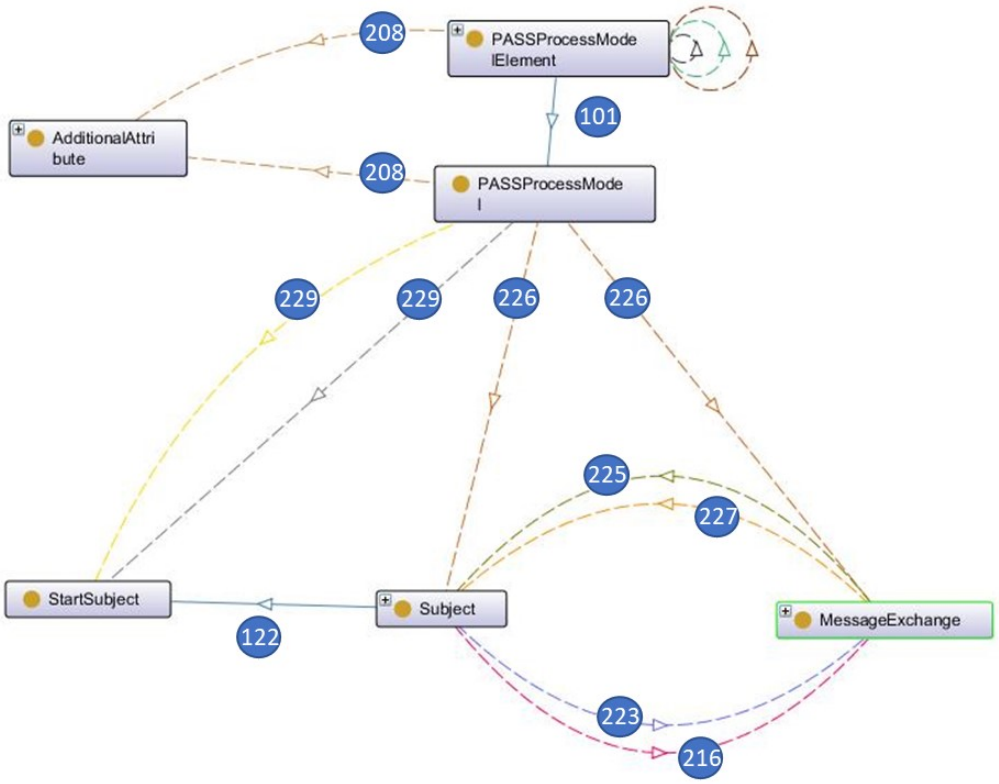

Figure 2.4: PASS Process Modell

define which messages are sent or received by a subject. Property `hasStartSubject` (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.

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 de-

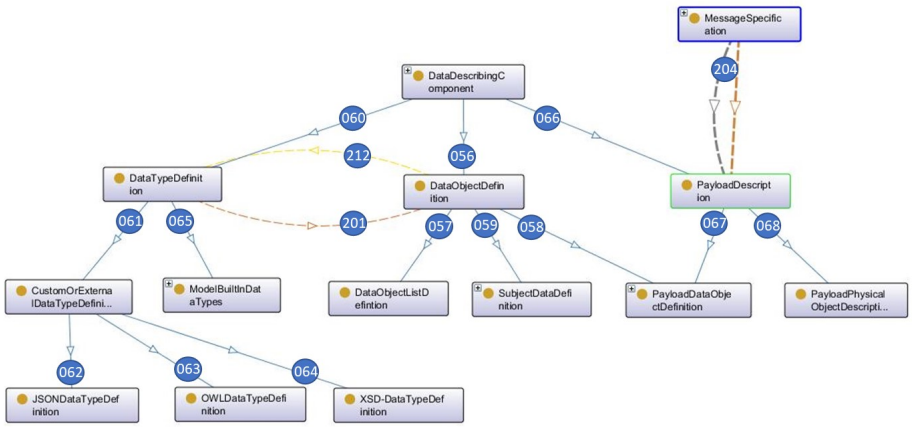

Figure 2.5: Data Description Component

fined data types whereas the class `CustomOfExternalDataTypeDefinition` is the superclass of JSON, OWL or XML based data type definitions(subclass relations 062, 63 and 064 in figure 2.5).

2.2.3 Interaction Describing Component

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

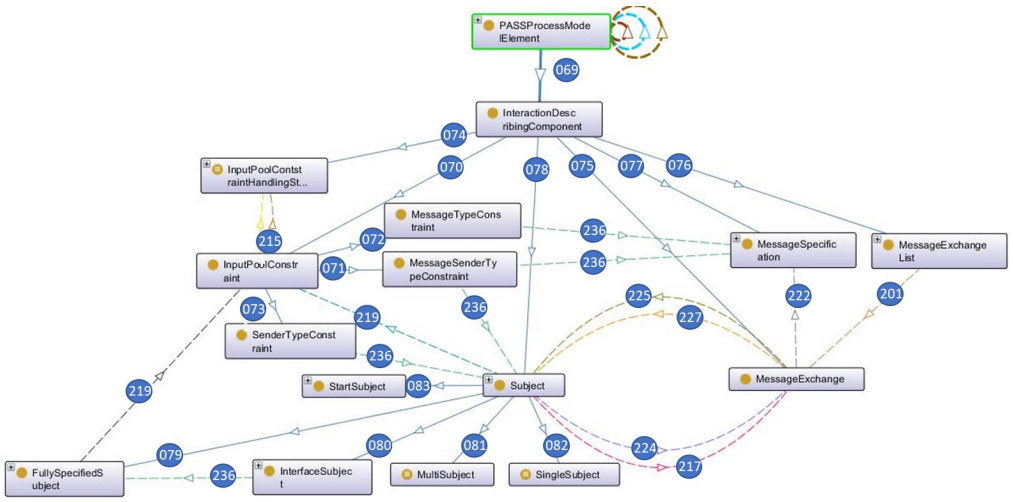

Figure 2.6: Subject Interaction Diagram

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

Each subject has an input pool. Input pools have three types of constraints (see section 2.1.3). This is expressed by the property references (in figure 2.6 number 236) and `InputPoolConstraints` (in figure 2.6 number 219). Constraints which are related to certain messages have references to the class `MessageSpecification`.

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

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

2.3 ASM Description

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

CHAPTER 3

Execution of a PASS Model

3.1 Informal Description of Subject Behavior and its Execution

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

3.1.1 Sending Messages

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

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

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

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

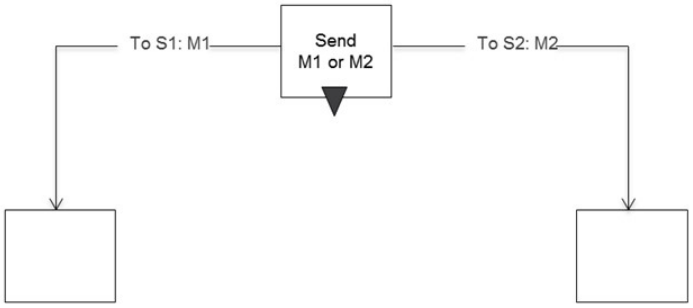

Figure 3.1: Example of alternative sending

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

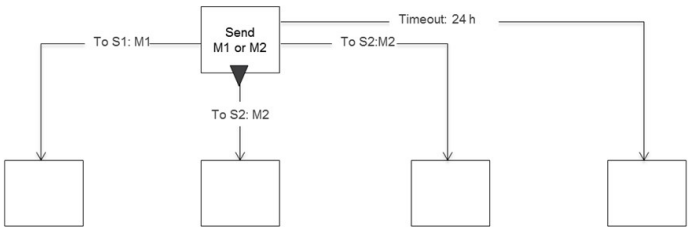

Figure 3.2: Send using time monitoring

3.1.2 Receiving Messages

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

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

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

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

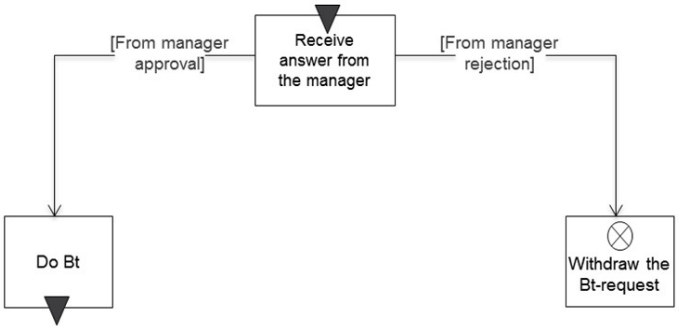

Figure 3.3: Example of alternative receiving

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

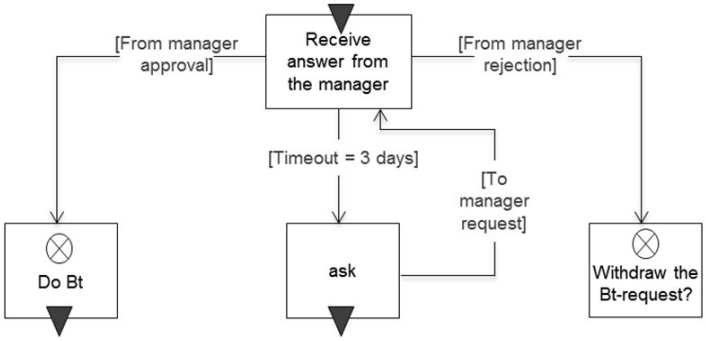

Figure 3.4: Time monitoring for message reception

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

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

3.1.3 Standard Subject Behavior

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

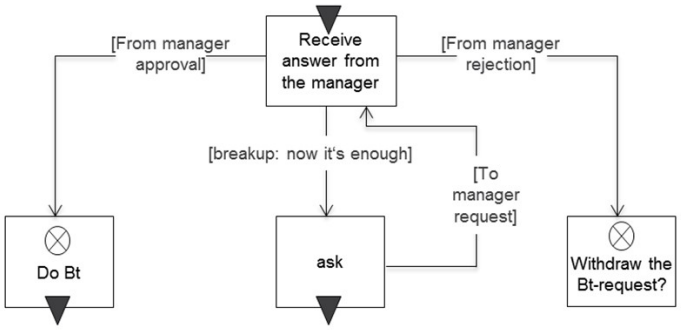

Figure 3.5: Message reception with manual interrupt

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

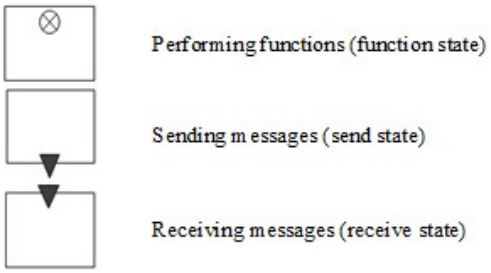

Figure 3.6: State types and coresponding symbols

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

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

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

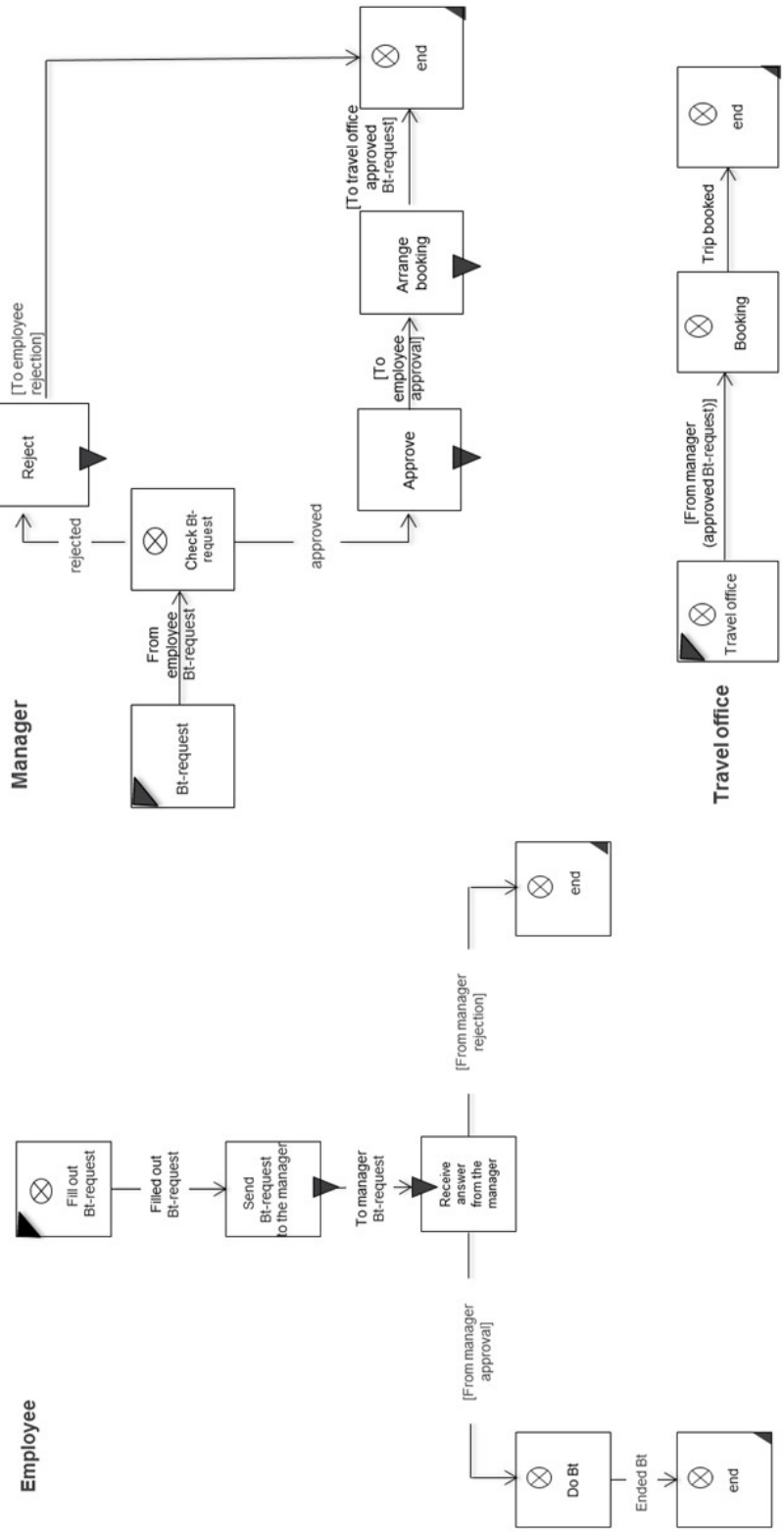

Figure 3.7: Subject behavior diagram for the subjects ‘employee’, ‘manager’, and ‘travel office’

3.1.4 Extended Behavior

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

Macros

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

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

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

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

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

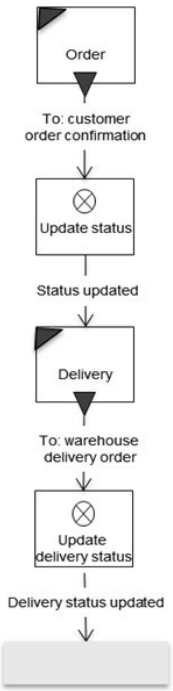

Figure 3.8: Behavior macro class 'request for approval'

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

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

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

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

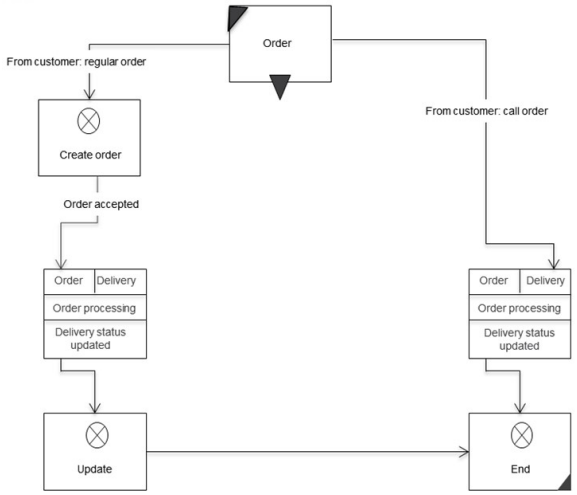

Figure 3.9: Subject behavior for order processing with macro integration

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

Guards: Exception Handling and Extensions

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

Exception handling is characterized by the fact that it can occur in a process in many behavior states of subjects. The receipt of certain messages, e.g., to abort the process, always results in the same processing pattern. This pattern would

have to be modeled for each state in which it is relevant. Exception handling causes high modeling effort and leads to complex process models since from each affected state a corresponding transition has to be specified. To prevent this situation, we introduce a concept similar to exception handling in programming languages or interrupt handling in operating systems.

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

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

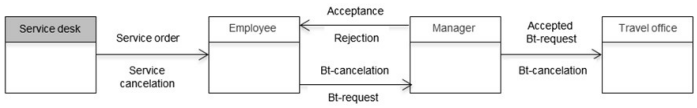

Figure 3.10: Communication structure diagram (CSD) of the business trip application

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

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

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

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

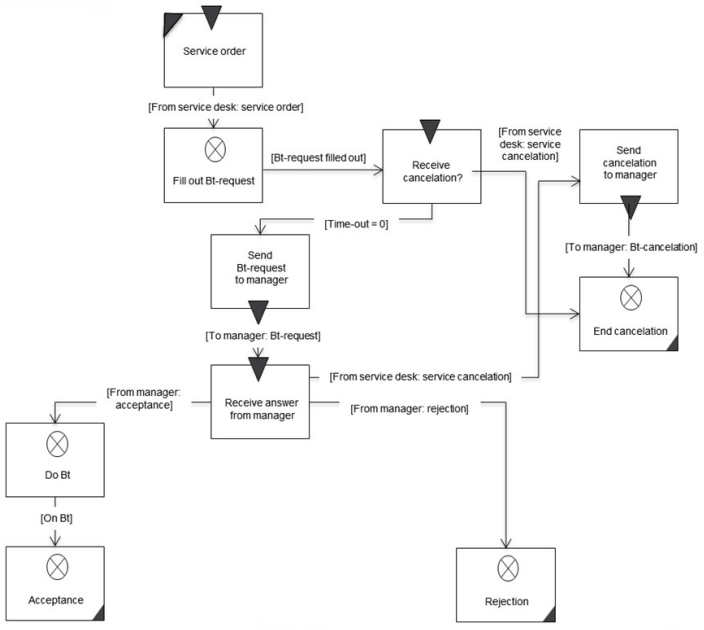

Figure 3.11: Handling the cancelation message using existing constructs

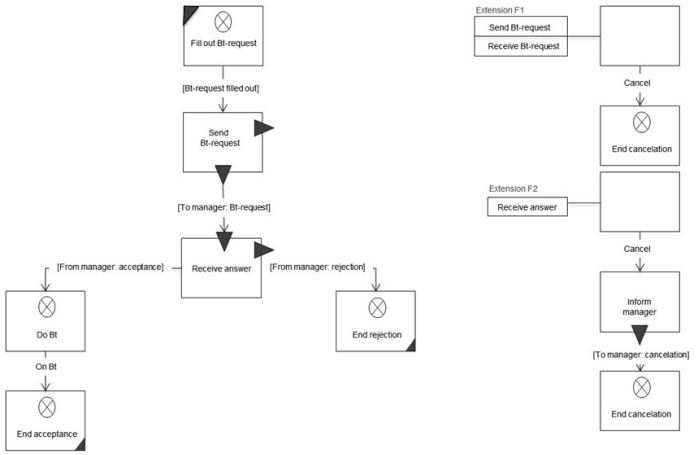

Figure 3.12: Behavior of subject 'employee' with exception handling

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

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

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

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

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

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

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

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

Alternative Actions (Freedom of Choice)

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

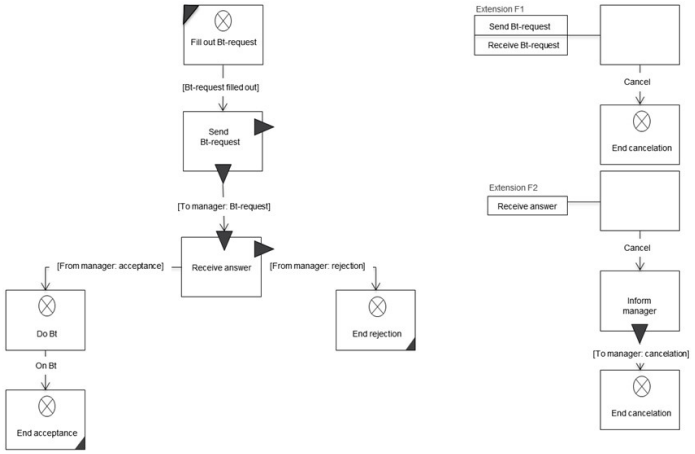

Figure 3.13: Subject behavior of employee with behavior extensions

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

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

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

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

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

Figure 3.14 shows an example for modeling alternative clauses. After receiving an order from the customer, three alternative behavioral sequences can be started, whereby the leftmost sequence, with the internal function 'update order' and sending the message 'deliver order' to the subject 'warehouse', must be started in any case. This is determined by the 'X' in the symbol for the start

of the alternative sequences (the gray bar is the starting point for alternatives). This sequence must be processed through to the end of the alternative because it is also marked in the end symbol of this alternative with an 'X' (gray bar as the endpoint of the alternative).

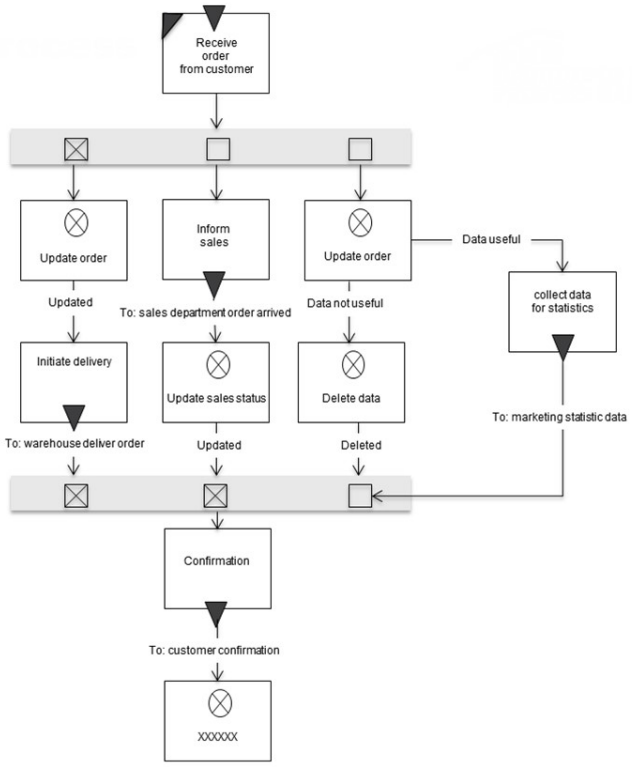

Figure 3.14: Example of Process Alternatives

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

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

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

The left alternative must be executed completely, and the middle alternative must also have been completed, if the first action ('inform sales' in the example) is executed. Only the left alternative can be processed because the middle one was never started. Alternatively, the sequence in the middle may have already reached its endpoint, while the left is not yet complete. In this case, the process

waits until the left one has reached its endpoint. Only then will the state 'confirmation' be reached in the alternative clause. The right branch neither needs to be started, nor to be completed. It is therefore irrelevant for the completion of the alternative construct.

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

3.2 Ontology of Subject Behavior Description

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

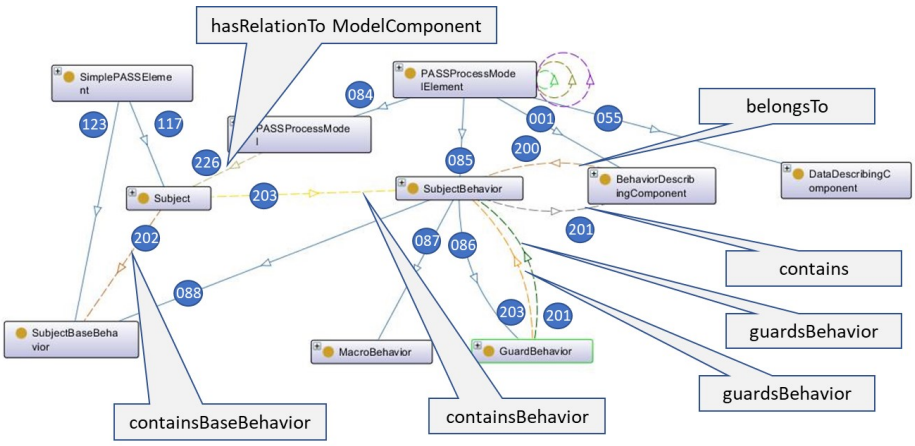

Figure 3.15: Structure of Subject Behavior Specification

3.2.1 Behavior Describing Component

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

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

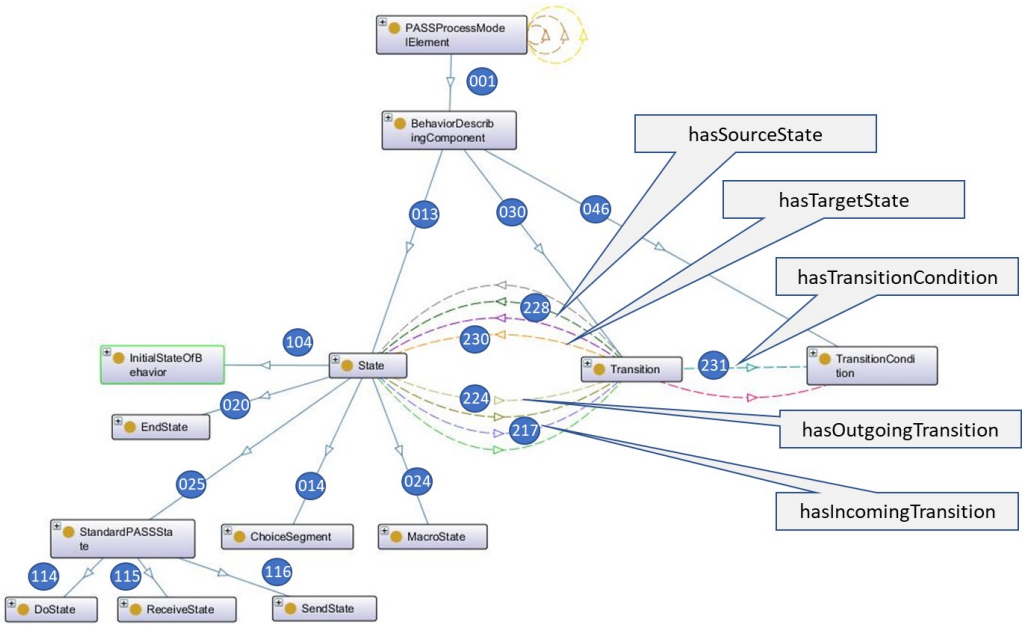

Figure 3.16: Subject Behavior describingComponent

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

States

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

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

Transitions

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

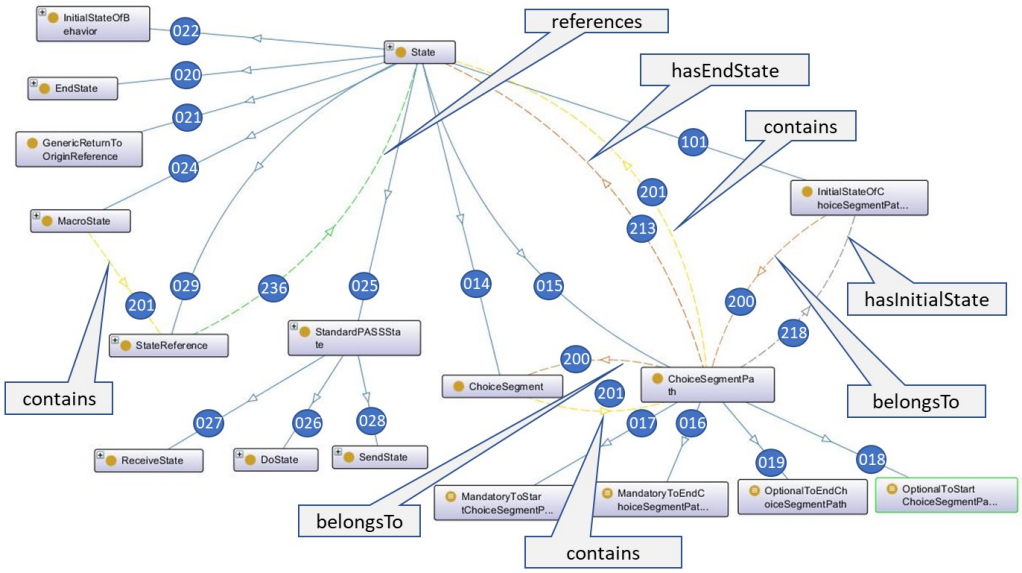

Figure 3.17: Details of States

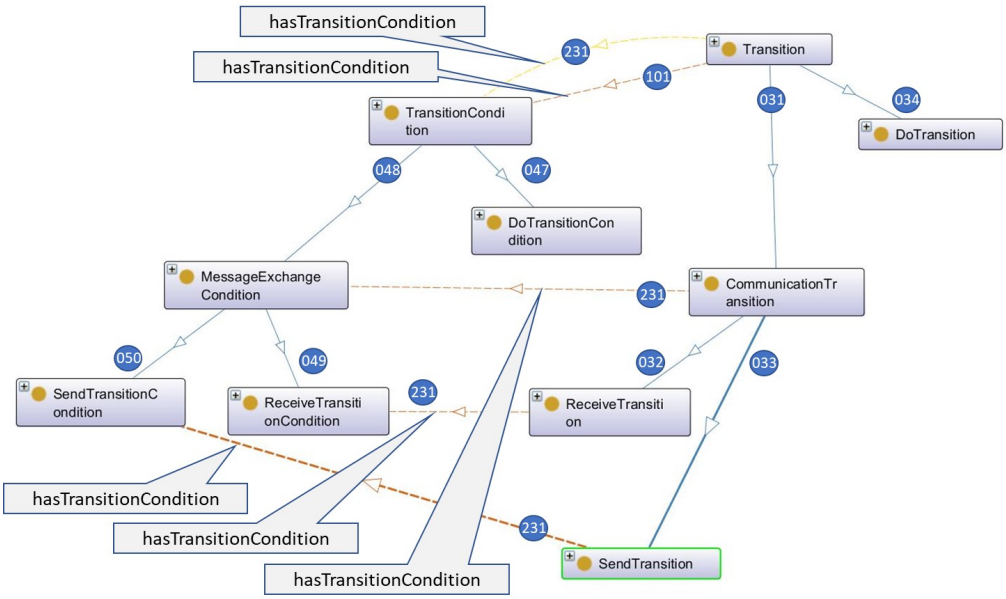

Figure 3.18: Details of transitions

3.3 ASM Definition of Subject Execution

This section provides a reduced overview of the ASM semantics given in Appendix C.

The goal of this section is to provide a minimal set of rules to establish a foundation to understand the ASM semantics, without going into their details. It is therefore just an informal introduction to the topic.

Rules under consideration to be presented:

- `rule Behavior`
- `rule PerformInternalAction`
- Inputpool & Send Definitions
- `rule PerformSend`
- `rule DoReservations`
- `rule DoReservation`
- `rule PerformEdgeSend`
- `rule PerformReceive`
- `rule CheckIP`
- `rule PerformEdgeReceive`

APPENDIX A

Classes and Properties of the PASS Ontology

A.1 All Classes (95)

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

- ReceiveFunction ; SRN: 008
  Specifications/descriptions for Receive-Functions describe in detail what the subject carrier is supposed to do in a state.
  DefaultFunctionReceive1_EnvoironmentChoice : present the surrounding execution environment with the given exit choices/conditions currently available depending on the current state of the subjects in-box. Waiting and not executing the receive action is an option.
  DefaultFunctionReceive2_AutoReceiveEarliest: automatically execute the according activity with the highest priority as soon as possible. In contrast to DefaultFunctionReceive1, it is not an option to prolong the reception and wait e.g. for another message.
- SendFunction ; SRN: 009
  Comments have to be added

- DoFunction ; SRN: 010
  Specifications or descriptions for Do-Functions describe in detail what the subject carrier is supposed to do in an according state. The default DoFunction
  1: present the surrounding execution environment with the given exit choices/-conditions and receive choice of one exit option –> define its Condition to be fulfilled in order to go to the next according state. The default DoFunction
  2: execute automatic rule evaluation (see DoTransitionCondition - ToDo) More specialized Do-Function Specifications may contain Data mappings denoting what of a subjects internal local Data can and should be:
  a) read: in order to simply see it or in order to send it of to an external function (e.g. a web service)
  b) write: in order to write incoming Data from e.g. a web Service or user input, to the local data fault

- ReceiveType ; SRN: 011
  Comments have to be added
- SendType ; SRN: 012
  Comments have to be added
- State ; SRN: 013
  A state in the behavior descriptions of a model
  - ChoiceSegment ; SRN: 014
    ChoiceSegments are groups of defined ChoiceSegementPaths. The paths may contain any amount of states. However, those states may not reach out of the bounds of the ChoiceSegmentPath.
  - ChoiceSegmentPath ; SRN: 015
    ChoiceSegments are groups of defined ChoiceSegementPaths. The paths may contain any amount of states. However, those states may not reach out of the bounds of the ChoiceSegmentPath.The path may contain any amount of states but may those states may not reach out of the bounds of the choice segment path. Similar to an initial state of a behavior a choice segment path must have one determined initial state. A transition within a choice segment path must not have a target state that is not inside the same choice segment path.
    - MandatoryToEndChoiceSegmentPath ; SRN: 016
      Comments have to be added
    - MandatoryToStartChoiceSegmentPath ; SRN: 017
      Comments have to be added
    - OptionalToEndChoiceSegmentPath ; SRN: 018
      Comments have to be added
    - OptionalToStartChoiceSegmentPath ; SRN: 019
      ChoiceSegmentPath and (isOptionalToEndChoiceSegmentPath value false)
  - EndState ; SRN: 020
    An end state a behavior. A subject behavior may have one or more end states. Only Do and Receive states may be end states. Send States cannot be end states.There are no individual end states that are not Do, Send, or Receive States at the same time.
  - GenericReturnToOriginReference ; SRN: 021
    Comments have to be added

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

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

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

- DayTimeTimerTransition ; SRN: 042
  Timer Transitions, denoting time outs for the state they originate from. The condition for a timer transition is that a certain amount of time has passed since the state it originates from has been entered.
  Day or Time Timers are measured in normal 24 hour days. Following the XML standard for time and day duration. They are to be differed from the timers that are timeout in units of years or months.
- YearMonthTimerTransition ; SRN: 044
  Timer transitions, denote time outs for the state they originate from. The condition for a timer transition is that a certain amount of time has passed since the state it originates from has been entered.
  Year or Month timers measure time in calendar years or months. The exact definitions for years and months depends on relevant or legal geographical location of the subject.

- UserCancelTransition ; SRN: 045
  A user cancel transition denotes the possibility to exit a receive state without the reception of a specific message.
  The user cancel allows for an arbitrary decision by a subject carrier/processor to abort a waiting process.

- TransitionCondition ; SRN: 046
  An exit condition belongs to alternatives which in turn is given for a state. An alternative (to leave the state) is only a real alternative if the exit condition is fulfilled (technically: if that according function returns "true")
  Note: Technically and during execution exit conditions belong to states. They define when it is allowed to leave that state. However, in PASS models exit conditions for states are defined and connected to the according transition edges. Therefore transition conditions are individual entities and not DataProperties.
  The according matching must be done by the model execution environment.
  By its existence, an edge/transition defines one possible follow up "state" for its state of origin. It is coupled with an "Exit Condition" that must be fulfilled in the originating state in order to leave the state.
  - DoTransitionCondition ; SRN: 047
    A TransitionCondition for the according DoTransitions and DoStates.
  - MessageExchangeCondition ; SRN: 048
    MessageExchangeConditon is the super class for Send End Receive Transition Conditions the both require either the sending or receiving (exchange) of a message to be fulfilled.
    - ReceiveTransitionCondition ; SRN: 049
      ReceiveTransitionConditions are conditions that state that a certain message must have been taken out of a subjects in-box to be fulfilled.
      These are the typical conditions defined by Receive Transitions.
    - SendTransitionCondition ; SRN: 050
      SendTransitionConditions are conditions that state that a certain message must have been successfully passed to another subjects in-box to be fulfilled.
      These are the typical conditions defined by Send transitions.
  - SendingFailedCondition ; SRN: 051
    Comments have to be added
  - TimeTransitionCondition ; SRN: 052
    A condition that is deemed 'true' and thus the according edge is gone, if: a surrounding execution system has deemed the time since entering the state and starting with the execution of the according action as too long (predefined by the outgoing edge)
    A condition that is true if a certain time defined has passed since the state this condition belongs to has been entered. (This is the standard TimeOut Exit condition)

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

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

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

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

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

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

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

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

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

    - ReceiveTransition ; SRN: 091
      Comments have to be added
    - SendTransition ; SRN: 092
      Comments have to be added
- DataMappingFunction ; SRN: 093
  Definitions of the ability/need to write or read data to and from a subject's personal data storage.
  DataMappingFunctions are behavior describing components since they define what the subject is supposed to do (mapping and translating data)
  Mapping may be done during reception of message, where data is taken from the message/Business Object (BO) and mapped/put into the local data field.
  It may be done during sending of a message where data is taken from the local vault and put into a BO.
  Or it may occur during executing a do function, where it is used to define read(get) and write (set) functions for the local data.
    - DataMappingIncomingToLocal ; SRN: 094
      A DataMapping that specifies how data is mapped from an an external source (message, function call etc.) to a subject's private defined data space.
    - DataMappingLocalToOutgoing ; SRN: 095
      A DataMapping that specifies how data is mapped from a subject's private data space to an an external destination (message, function call etc.)"
- DoTransition ; SRN: 096
  Comments have to be added
- DoTransitionCondition ; SRN: 097
  A TransitionCondition for the according DoTransitions and DoStates.
- EndState ; SRN: 098
  An end state a behavior. A subject behavior may have one or more end states. Only Do and Receive states may be end states. Send States cannot be end states.
  There are no individual end states that are not Do, Send, or Receive States at the same time.
- FunctionSpecification ; SRN: 099
  A function specification for state denotes
  Concept: Definitions of calls of (mostly technical) functions (e.g. Web-service, Scripts, Database access,) that are not part of the process model.
  Function Specifications are more than "Data Properties"? –> - If special function types (e.g. Defaults) are supposed to be reused, having them as explicit entities is a the better OWL-modeling choice.
    - CommunicationAct ; SRN: 100
      A super class for specialized FunctionSpecification of communication acts (send and receive)
        - ReceiveFunction ; SRN: 101
          Specifications/descriptions for Receive-Functions describe in detail what the subject carrier is supposed to do in a state.
          DefaultFunctionReceive1_EnvoironmentChoice : present the surrounding execution environment with the given exit choices/conditions currently available depending on the current state of the subjects in-box. Waiting and not executing the receive action is an option.
          DefaultFunctionReceive2_AutoReceiveEarliest: automatically execute the according activity with the highest priority as soon as possible. In contrast to DefaultFunctionReceive1, it is not an option to prolong the reception and wait e.g. for another message.
        - SendFunction ; SRN: 102
          Comments have to be added
    - DoFunction ; SRN: 103
      Specifications or descriptions for Do-Functions describe in detail what the subject carrier is supposed to do in an according state.
      The default DoFunction 1: present the surrounding execution environment with the given exit choices/conditions and receive choice of one exit option –> define its Condition to be fulfilled in order to go to the next according state.

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

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

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

- SubjectBaseBehavior ; SRN: 123
  The standard behavior model type

A.2 Object Properties (42)

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

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

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

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

A.3 Data Properties (27)

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

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

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

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

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

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

APPENDIX B

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

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

Table 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 And their individual defaultinstances: **InputPoolConstraintStrategy-Blocking** **InputPoolConstraintStrategy-DeleteLatest** **InputPoolConstraintStrategy-DeleteOldest** **InputPoolConstraintStrategy-Drop**	*/Blocking; DropYoungest; DropOldest; DropIncoming/*	Default Input Pool handling strategies for
Execution Concept – can be restricted by **InputPoolConstraint** – for its Input Pool	*P / inputPool*	The actual Input Pool
synchronous communication	Definition for an input pool **constraint set to 0** requiring sender and receiver interpreter to be in the corresponding send and receive states at the same time in order to actually communicate (as messages cannot be passed to an input pool)	

Table 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

PASS CoreASM Reference Implementation

C.1 Conceptual Differences ASM Semantic / OWL Model

This implementation provides more language elements than covered by the OWL description and also gives some concrete implementations.

Additionally to OWL:

- Internal Behavior: usage of Subject Data: DataModificationFunction (VarMan: Selection, Concatenation, ...), DataMappingFunction (StoreMessage, UseMessageContent, UseCorrelationID). Subject Data can be scoped to Macro Instance.
- Interaction: Subject Restart, Inputpool Functions (CloseIP, OpenIP, ...), CorrelationID, Mobility of Channel.

There are also some conceptional differences and limitations:

- only blocking asynchronous send, i.e. IP strategy blocking - no delete / drop. no sync send.
- modal split / join instead of ChoiceSegement. only mandatory to start and end / no optional start or optional end.
- timeout transitions: designed for interactive validation -> duration in seconds, no business days. also no reminders.
- Observer: different approach: no native support; possible via Modal Split + Receive State with State Priority + following Cancel Function

C.2 ACTUAL APPENDIX

```
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: channelFor

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

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

Listing 2: activeStates

```
// 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 // -> TransitionNumber
function initializedSelectedTransition : LIST * NUMBER * NUMBER -> BOOLEAN
function startTime  : LIST * NUMBER * NUMBER -> NUMBER

// can exit
// Channel * MacroInstanceNumber * TransitionNumber -> BOOLEAN
function exitCondition : LIST * NUMBER * NUMBER -> BOOLEAN
// TODO: may rename, possibly transitionEnabled ???

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

Listing 3: initializedState

```
derived shouldTimeout(ch, MI, stateNumber) = return boolres in {
  let processID = processIDOf(ch) in {
    if (hasTimeoutTransition(processID, stateNumber) = true and startTime(ch, MI, stateNu
      let transitionNumber = first_outgoingTimeoutTransition(processID, stateNumber) in
      let timeout = transitionTimeout(processID, transitionNumber) * 1000 * 1000 * 1000 i
        let runningTime = (nanoTime() - startTime(ch, MI, stateNumber)) in
          boolres := (runningTime > timeout)
      }
    }
    else {
      boolres := false
    }
  }
}
```

Listing 4: shouldTimeout

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

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

Listing 5: variable

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

/* store all locations where an inputPool was defined for to allow IPEmpty and receiving
// receiverChannel -> {[senderSubjID, messageType, correlationID], ..}
function inputPoolDefined : LIST -> SET
```

Listing 6: 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 -> STRING
function messageContent : LIST * NUMBER * NUMBER -> LIST

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

Listing 7: receivedMessages

```
// 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 8: macroTerminationResult

```
// called form PerformEnd (iff other states are active) and AbortCallMacro
rule AbortMacroInstance(MIAbort, ignoreState) = {
  foreach currentState in activeStates(channelFor(self), MIAbort) do {
    add [MIAbort, currentState] to killStates(channelFor(self))
  }

  ClearAllVarInMIForChannel(channelFor(self), MIAbort)
}
```

Listing 9: AbortMacroInstance

```
rule StartASMAgent(ch) = {
  extend Agents with a do seqblock
    channelFor(a) := ch

    add a to asmAgents

    program(a) := @StartMainMacro
  endseqblock
}
```

Listing 10: StartASMAgent

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

    variableDefined(ch) := {[0, "$self"], [0, "$empty"]}
    variable(ch, 0, "$self") := ["ChannelInformation", {ch}]
    variable(ch, 0, "$empty") := ["Text", ""]
  }

  let mID  = subjectMainMacro(processIDFor(self), subjectIDFor(self)) in
  let startState = macroStartState(processIDFor(self), mID) in
  let MI   = 1 in // 0 reserved for top-level variable manipulation; 1 mainmacro
  {
    macroNumberOfMI(channelFor(self), MI) := mID

    nextMacroInstanceNumber(channelFor(self)) := MI  + 1

    activeStates(channelFor(self), MI) := [startState]
  }

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

Listing 11: StartMainMacro

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

Listing 12: StartMacro

```
/*
0 - REPEAT
  -* Behaviour should be executed again for this state
  -* results of previous execution will be merged with following execution in one glo
  -* no other states are allowed to be executed
1 - DONE
  -* no other active states are allowed to be executed
  -* new states are started
  -* global ASM / LTS step should be done
2 - 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
3 - LOWER
  -* nothing changed / waiting for input
  -* other active states, even with lower priority, can be executed
*/

// ch * MacroInstanceNumber * stateNumber => Int
function executionState : LIST * NUMBER * NUMBER -> NUMBER


/*
1 - DONE
2 - NEXT
3 - LOWER
*/
// ch * MacroInstanceNumber => Int
function macroExecutionState : LIST * NUMBER -> NUMBER

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

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

Listing 13: 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 14: AddState

```
rule SubjectBehaviour = {
  choose x in killStates(channelFor(self)) do {
    KillBehaviour(nth(x, 1), nth(x, 2))
  }
  ifnone seqblock
    MacroBehaviour(1)

    // reset
    macroExecutionState(channelFor(self), 1) := undef
  endseqblock
}
```

Listing 15: SubjectBehaviour

```
rule KillBehaviour(MI, currentStateNumber) = {
  if (initializedState(channelFor(self), MI, currentStateNumber) != true) then {
    remove [MI, currentStateNumber] from killStates(channelFor(self))
    remove n from activeStates(channelFor(self), MI)
  }
  else seqblock
    executionState(channelFor(self), MI, currentStateNumber) := undef
    addStates(channelFor(self), MI, currentStateNumber)       := []
    removeStates(channelFor(self), MI, currentStateNumber)    := []

    if (abortionCompleted(channelFor(self), MI, currentStateNumber) != true) then {
      Abort(MI, currentStateNumber)
    }
    else {
      RemoveState(MI, currentStateNumber, MI, currentStateNumber)
      SetExecutionState(MI, currentStateNumber, DONE)
    }

    if (executionState(channelFor(self), MI, currentStateNumber) != 1) then {
      Crash()
    }
    else if (|addStates(channelFor(self), MI, currentStateNumber)| > 0) then {
      Crash()
    }
    else if (|removeStates(channelFor(self), MI, currentStateNumber)| > 0) then {
      if (removeStates(channelFor(self), MI, currentStateNumber) != [[MI, currentStat
        Crash()
      }
      else {
        foreach x in removeStates(channelFor(self), MI, currentStateNumber) do {
          let xMI = nth(x, 1),
            xN  = nth(x, 2) in {
            remove [xMI, xN] from killStates(channelFor(self))
            ResetState(xMI, xN)
            remove xN from activeStates(channelFor(self), xMI)
          }
        }
      }
    }
  endseqblock
}
```

Listing 16: KillBehaviour

```
rule MacroBehaviour(MI) = {
  let processID = processIDFor(self) in
  local listres := activeStates(channelFor(self), MI) in seqblock // remaining states
    macroExecutionState(channelFor(self), MI) := undef

    // can not be done with foreach as listres is modified depending on the executionStat
    while (|listres| > 0) do
    let stateNumber = getAnyStateWithHighestPrio(processID, listres) in {
      seqblock
        executionState(channelFor(self), MI, stateNumber) := undef
        addStates(channelFor(self), MI, stateNumber)      := []
        removeStates(channelFor(self), MI, stateNumber)   := []

        Behaviour(MI, stateNumber)

        // WARNING: mutates listres!
        let state = executionState(channelFor(self), MI, stateNumber) in
          UpdateRemainingStates(MI, stateNumber, state, listres)

        UpdateActiveStates(MI, stateNumber)
      endseqblock
    }
  endseqblock
}
```

Listing 17: MacroBehaviour

```
// WARNING: mutates listres!
rule UpdateRemainingStates(MI, stateNumber, exState, remainingStates) = {
  if (exState = REPEAT) then {
    remainingStates := [stateNumber]

    macroExecutionState(channelFor(self), MI) := DONE

    // quasi-reset
    executionState(channelFor(self), 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(channelFor(self), MI, stateNumber)
        with nth(x, 1) = MI do {
        add nth(x, 2) to remainingStates
      }

    macroExecutionState(channelFor(self), MI) := DONE // something changed and nothin

    // quasi-reset
    executionState(channelFor(self), MI, stateNumber) := NEXT
  }
  else if (exState = NEXT) then {
    seqblock
      remove stateNumber from remainingStates // remove self

      remainingStates := filterStatesWithSamePrio(processIDFor(self), remainingStates

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

    if (macroExecutionState(channelFor(self), MI) != DONE and macroExecutionState(cha
      macroExecutionState(channelFor(self), MI) := LOWER
    }
  }
}
```

Listing 18: UpdateRemainingStates

```
rule UpdateActiveStates(MI, stateNumber) = seqblock
  // NOTE: everything needs to be sequential, as activeStates is a list and not a set

  foreach x in addStates(channelFor(self), MI, stateNumber) do {
    let xMI = nth(x, 1),
      xN  = nth(x, 2) in {
      add xN to activeStates(channelFor(self), xMI)
    }
  }

  addStates(channelFor(self), MI, stateNumber) := undef

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

      ResetState(xMI, xN)
    }
  }

  removeStates(channelFor(self), MI, stateNumber) := undef
endseqblock
```

Listing 19: UpdateActiveStates

```
// whether to abort the state or not
derived abortState(MI, stateNumber) = return boolres in {
  boolres := ((timeoutActive(channelFor(self), MI, stateNumber) = true) or (cancelDecisic
}
```

Listing 20: abortState

```
rule Behaviour(MI, currentStateNumber) =
 let s  = currentStateNumber,
 let 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 transitionNumber = 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 21: Behaviour

```
rule AbortState(MI, currentStateNumber) = {
  if (abortionCompleted(channelFor(self), MI, currentStateNumber) != true) then {
    Abort(MI, currentStateNumber)
  }
  else {
    if (cancelDecision(channelFor(self), MI, currentStateNumber) = true) then {
      let transitionNumber = first_outgoingCancelTransition(processIDFor(self), curre
      let t = targetStateNumber(processIDFor(self), transitionNumber) in {
        Proceed(MI, currentStateNumber, t)
      }
    }
    else if (timeoutActive(channelFor(self), MI, currentStateNumber) = true) then {
      let transitionNumber = first_outgoingTimeoutTransition(processIDFor(self), curr
      let t = targetStateNumber(processIDFor(self), transitionNumber) in {
        Proceed(MI, currentStateNumber, t)
      }
    }

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

Listing 22: AbortState

```
rule StartState(MI, currentStateNumber) = {
  let processID = processIDFor(self) in
  let sType     = stateType(processID, currentStateNumber) in
  seqblock
    InitializeCompletion(MI, currentStateNumber)
    abortionCompleted(channelFor(self), MI, currentStateNumber) := false

    ResetTimeout(MI, currentStateNumber)
    cancelDecision(channelFor(self), MI, currentStateNumber) := false

    DisableAllTransitions(MI, currentStateNumber)
    initializedSelectedTransition(channelFor(self), MI, currentStateNumber) := false

    wantInput(channelFor(self), MI, currentStateNumber) := {}

    case sType of
      "function"       : StartFunction(MI, currentStateNumber)
      "internalAction" : StartInternalAction(MI, currentStateNumber)
      "send"           : StartSend(MI, currentStateNumber)
      "receive"        : SetExecutionState(MI, currentStateNumber, LOWER)
      "end"            : StartEnd(MI, currentStateNumber)
    endcase

    initializedState(channelFor(self), MI, currentStateNumber) := true
  endseqblock
}
```

Listing 23: StartState

```
rule ResetState(MI, stateNumber) = {
  executionState(channelFor(self), MI, stateNumber) := undef

  // StartState

  initializedState(channelFor(self), MI, stateNumber) := undef

  completed(channelFor(self), MI, stateNumber) := undef
  abortionCompleted(channelFor(self), MI, stateNumber) := undef

  startTime(channelFor(self), MI, stateNumber) := undef
  timeoutActive(channelFor(self), MI, stateNumber) := undef

  cancelDecision(channelFor(self), MI, stateNumber) := undef

  selectedTransition(channelFor(self), MI, stateNumber) := undef

  forall transitionNumber in outgoingNormalTransitions(processIDFor(self), stateNumbe
    exitCondition(channelFor(self), MI, transitionNumber) := undef
    transitionCompleted(channelFor(self), MI, transitionNumber) := undef
  }

  initializedSelectedTransition(channelFor(self), MI, stateNumber) := undef

  wantInput(channelFor(self), MI, stateNumber) := undef

  // StartSend
  receivers(channelFor(self), MI, stateNumber) := undef
  reservationsDone(channelFor(self), MI, stateNumber) := undef
  messageContent(channelFor(self), MI, stateNumber) := undef
}
```

Listing 24: ResetState

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

Listing 25: Perform

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

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

Listing 26: StartSelectedTransition

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

Listing 27: Proceed

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

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

rule ActivateTimeout(MI, currentStateNumber) = {
  timeoutActive(channelFor(self), MI, currentStateNumber) := true
}

rule InitializeCompletion(MI, currentStateNumber) = {
  completed(channelFor(self), MI, currentStateNumber) := false
}

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

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


rule EnableTransition(MI, transitionNumber) = {
  exitCondition(channelFor(self), MI, transitionNumber) := true
}

rule EnableAllTransitions(MI, currentStateNumber) = {
  forall transitionNumber in outgoingNormalTransitions(processIDFor(self), currentSta
    EnableTransition(MI, transitionNumber)
  }
}


rule DisableTransition(MI, currentStateNumber, transitionNumber) = {
  exitCondition(channelFor(self), MI, transitionNumber) := false
}

rule DisableAllTransitions(MI, currentStateNumber) = {
  selectedTransition(channelFor(self), MI, currentStateNumber) := undef

  forall transitionNumber in outgoingNormalTransitions(processIDFor(self), currentSta
    DisableTransition(MI, currentStateNumber, transitionNumber)
  }
}

rule InitializeCompletionTransition(MI, transitionNumber) = {
  transitionCompleted(channelFor(self), MI, transitionNumber) := false
}


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

  transitionCompleted(channelFor(self), MI, transitionNumber) := true
```

```
rule StartInternalAction(MI, currentStateNumber) = {
  let processID = processIDFor(self) in {
    StartTimeout(MI, currentStateNumber)

    EnableAllTransitions(MI, currentStateNumber)

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

Listing 29: StartInternalAction

```
rule PerformInternalAction(MI, currentStateNumber) = {
  if (shouldTimeout(channelFor(self), MI, currentStateNumber) = true) then {
    SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
    ActivateTimeout(MI, currentStateNumber)
  }
  else {
    if (selectedTransition(channelFor(self), MI, currentStateNumber) != undef) then {
      SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
    }
    else {
      SelectTransition(MI, currentStateNumber)
    }
  }
}
```

Listing 30: PerformInternalAction

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

  let processID = processIDFor(self) in
  let actionName = stateFunction(processID, currentStateNumber) in {
    if (startFunction(actionName) = undef) then {
      skip
    }
    else {
      call startFunction(actionName) (MI, currentStateNumber)
    }
  }

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

Listing 31: StartFunction

```
rule PerformFunction(MI, currentStateNumber) = {
  if (shouldTimeout(channelFor(self), MI, currentStateNumber) = true) then {
    SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
    ActivateTimeout(MI, currentStateNumber)
  }
  else {
    let processID = processIDFor(self) in
    let actionName = stateFunction(processID, currentStateNumber),
      args       = stateFunctionArguments(processID, currentStateNumber) in
        call performFunction(actionName) (MI, currentStateNumber, args)
  }
}
```

Listing 32: PerformFunction

```
rule AbortFunction(MI, currentStateNumber) = {
  let processID = processIDFor(self) in
  let functionName = stateFunction(processID, currentStateNumber) in {
    if (abortFunction(functionName) = undef) then {
      SetAbortionCompleted(MI, currentStateNumber)  // sets executionState to DONE
    }
    else {
      call abortFunction(functionName) (MI, currentStateNumber) // must set abortionC
    }
  }
}
```

Listing 33: AbortFunction

```
rule PerformTransitionFunction(MI, currentStateNumber, transitionNumber) = {
  let processID = processIDFor(self) in
  let functionName = stateFunction(processID, currentStateNumber) in {
    if (performTransitionFunction(functionName) = undef) then {
      SetCompletedTransition(MI, currentStateNumber, transitionNumber) // sets execut
    }
    else {
      call performTransitionFunction(functionName) (MI, currentStateNumber, transitio
    }
  }
}
```

Listing 34: PerformTransitionFunction

```
rule SetCompletedAction(MI, currentStateNumber, res) = {
  let processID = processIDFor(self) in {
    if (res = undef) then {
      choose transitionNumber in outgoingNormalTransitions(processID, currentStateNumber)
        selectedTransition(channelFor(self), MI, currentStateNumber) := transitionNumber
      }
    }
    else {
      let transitionNumber = getTransitionByLabel(processID, currentStateNumber, res) in
        selectedTransition(channelFor(self), MI, currentStateNumber) := transitionNumber
    }
  }

  SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
}
```

Listing 35: SetCompletedAction

```
rule StartSend(MI, currentStateNumber) = {
  let processID = processIDFor(self) in
  let transitionNumber = first_outgoingNormalTransition(processID, currentStateNumber) in
    receivers(channelFor(self), MI, currentStateNumber) := undef
    reservationsDone(channelFor(self), MI, currentStateNumber) := {}
    messageContent(channelFor(self), MI, currentStateNumber) := loadVar(MI, messageConten

    let cIDVarname = messageNewCorrelationVar(processID, transitionNumber) in
      if (cIDVarname != undef and cIDVarname != "") then {
        SetVar(MI, cIDVarname, "CorrelationID", nextCorrelationID)
        // FIXME: it's a bad idea to save the cID early and read this variable later: it
        nextCorrelationID := nextCorrelationID + 1
        nextCorrelationIDUsedBy(nextCorrelationID) := self // ensure no other agent incre
      }
  }

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

Listing 36: StartSend

```
rule PerformSend(MI, currentStateNumber) = {
  if (receivers(channelFor(self), MI, currentStateNumber) = undef) then {
    SelectReceivers(MI, currentStateNumber) // sets executionState to DONE / REPEAT /
  }
  else if (messageContent(channelFor(self), MI, currentStateNumber) = undef) then {
    SetMessageContent(MI, currentStateNumber) // sets executionState to DONE / NEXT
  }
  else if (startTime(channelFor(self), MI, currentStateNumber) = undef) then {
    StartTimeout(MI, currentStateNumber)

    SetExecutionState(MI, currentStateNumber, REPEAT)
  }
  else if ( | receivers(channelFor(self), MI, currentStateNumber) | = | reservationsD
    TryCompletePerformSend(MI, currentStateNumber) // sets executionState to NEXT / R
  }
  else if (shouldTimeout(channelFor(self), MI, currentStateNumber) = true) then {
    SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
    ActivateTimeout(MI, currentStateNumber)
  }
  else {
    DoReservations(MI, currentStateNumber) // sets executionState to DONE or NEXT
  }
}
```

Listing 37: PerformSend

```
rule TryCompletePerformSend(MI, currentStateNumber) = {
  if (anyNonProperTerminated(receivers(channelFor(self), MI, currentStateNumber)) = t
    debuginfo TryCompletePerformSend self + ": a receiver where a reservation was pla

    if (shouldTimeout(channelFor(self), MI, currentStateNumber) = true) then {
      debuginfo TryCompletePerformSend self + ": shouldTimeout"
      SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
      ActivateTimeout(MI, currentStateNumber)
    }
    else {
      SetExecutionState(MI, currentStateNumber, NEXT)
    }
  }
  else {
    // there must be only one transition
    let transitionNumber = first_outgoingNormalTransition(processIDFor(self), current
      selectedTransition(channelFor(self), MI, currentStateNumber) := transitionNumbe
    }

    SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
  }
}
```

Listing 38: TryCompletePerformSend

```
rule SelectReceivers(MI, currentStateNumber) = {
  let processID = processIDFor(self) in
  let transitionNumber = first_outgoingNormalTransition(processID, currentStateNumber) in
  let countMin = messageSubjectCountMin(processID, transitionNumber),
    countMax = messageSubjectCountMax(processID, transitionNumber) in {
    if (messageSubjectVar(processID, transitionNumber) != undef and messageSubjectVar(pro
      let rChs = loadChannelsFromVariable(MI, messageSubjectVar(processID, transitionNumb

        // countMin = 0 => ALL
        if (| rChs | = 0 or (countMin != 0 and | rChs | < countMin)) then {
          debuginfo SelectReceivers self + ": WARN: not enough receivers given in '" + me

          SetExecutionState(MI, currentStateNumber, NEXT)
        }
        // countMax = 0 => at least countMin, but unlimited
        else if (countMax != 0 and | rChs | > countMax)  then {
          debuginfo SelectReceivers self + ": too many receivers given -> Selection"

          SelectReceivers_Selection(MI, currentStateNumber, rChs, countMin, countMax) //
        }
        else {
          debuginfo SelectReceivers self + ": receivers fit min/max -> use them"

          receivers(channelFor(self), MI, currentStateNumber) := rChs

          SetExecutionState(MI, currentStateNumber, REPEAT)
        }
      }
    }
    else {
      if (selectAgentsResult(channelFor(self), MI, currentStateNumber) != undef) then {
        let y = selectAgentsResult(channelFor(self), MI, currentStateNumber) in
        {
          if (|y| < countMin or |y| > countMax) then {
            debuginfo SelectReceivers self + ": invalid receivers selected, must be betwe
            Crash()
          }

          receivers(channelFor(self), MI, currentStateNumber) := y

          selectAgentsResult(channelFor(self), MI, currentStateNumber) := undef

          SetExecutionState(MI, currentStateNumber, REPEAT)
        }
      }
      else {
        let sIDLocal = messageSubjectId(processID, transitionNumber) in {
          SelectAgents(MI, currentStateNumber, sIDLocal, countMin, countMax) // sets exec
        }
      }
    }
  }
}
```

Listing 39: SelectReceivers

```
rule SelectReceivers_Selection(MI, currentStateNumber, rChs, minimum, maximum) = {
  let res = selectionResult(channelFor(self), MI, currentStateNumber) in
  if (res = undef) then {
    let src = ["ChannelInformation", rChs] in {
      Selection(MI, currentStateNumber, src, minimum, maximum)
    }
  }
  else {
    selectionResult(channelFor(self), MI, currentStateNumber) := undef

    receivers(channelFor(self), MI, currentStateNumber) := res

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

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

  SetAbortionCompleted(MI, currentStateNumber)  // sets executionState to DONE
}
```

Listing 41: AbortSend

```
rule PerformTransitionSend(MI, currentStateNumber, transitionNumber) = {
  let processID = processIDFor(self) in {
    let storeReceiverVarname = messageStoreReceiverVar(processID, transitionNumber) i
      if (storeReceiverVarname != undef and storeReceiverVarname != "") then {
        SetVar(MI, storeReceiverVarname, "ChannelInformation", reservationsDone(chann
      }

    foreach r in reservationsDone(channelFor(self), MI, currentStateNumber) do {
      ReplaceReservation(MI, currentStateNumber, r)

      EnsureRunning(r)
    }

    SetCompletedTransition(MI, currentStateNumber, transitionNumber) // sets executio
  }
}
```

Listing 42: PerformTransitionSend

```
rule SetMessageContent(MI, currentStateNumber) = {
  if not(contains(wantInput(channelFor(self), MI, currentStateNumber), "MessageContentDec
    add "MessageContentDecision" to wantInput(channelFor(self), MI, currentStateNumber)

    SetExecutionState(MI, currentStateNumber, DONE)
  }
  else {
    debuginfo SetMessageContent self + ": waiting for messageContent"
    SetExecutionState(MI, currentStateNumber, NEXT)
  }
}
```

Listing 43: SetMessageContent

```
rule DoReservations(MI, currentStateNumber) = {
  local boolres := false in { // hasPlacedReservation
    seq
      let receiversTodo = (receivers(channelFor(self), MI, currentStateNumber) diff reser
      foreach receiver in receiversTodo do {
        local boolres1 := false in {
          seq
            boolres1 <- DoReservation(MI, currentStateNumber, receiver) // returns true
          next
            if (boolres1 = true) then {
              boolres := true
            }
        }
      }
    next
      if (boolres = true) then {
        debuginfo DoReservations self + ": reservation(s) placed, make update"
        SetExecutionState(MI, currentStateNumber, DONE)
      }
      else {
        debuginfo DoReservations self + ": no reservations made, allow other states"
        SetExecutionState(MI, currentStateNumber, NEXT)
      }
  }
}
```

Listing 44: DoReservations

```
// result = hasPlacedReservation
rule DoReservation(MI, currentStateNumber, receiverChannel) = {
  if (properTerminated(receiverChannel) = true) then {
      let processID          = processIDFor(self) in
      let transitionNumber   = first_outgoingNormalTransition(processID, currentState
        senderChannel      = channelFor(self),
        receiverProcessID  = processIDOf(receiverChannel) in
      let senderSubjectID    = searchSenderSubjectID(processID, subjectIDFor(self), r
        msgCorrelationID   = loadCorrelationID(MI, messageNewCorrelationVar(processID
        ipCorrelationID    = loadCorrelationID(MI, messageWithCorrelationVar(processI
      let reservationMessage = [senderChannel, messageType(processID, transitionNumbe
        seq
          if (inputPool(receiverChannel, senderSubjectID, messageType(processID, tran
              add [senderSubjectID, messageType(processID, transitionNumber), ipCorre
              inputPool(receiverChannel, senderSubjectID, messageType(processID, tran
          }
        next
          if (inputPoolIsClosed(receiverChannel, senderSubjectID, messageType(process
              if (inputPoolGetFreeSpace(receiverChannel, senderSubjectID, messageType
                enqueue reservationMessage into inputPool(receiverChannel, senderSubj
                add receiverChannel to reservationsDone(channelFor(self), MI, current
                debuginfo DoReservation self + ": added reservation to inputPool: " +
                result := true
              }
              else {
                debuginfo DoReservation self + ": no free space!"
                result := false
              }
          }
          else {
              debuginfo DoReservation self + ": inputPoolIsClosed"
              result := false
          }
      }
  }
  else {
      debuginfo DoReservation self + ": non-properTerminated, skipping receiver"
      result := false
  }
}
```

Listing 45: DoReservation

```
rule CancelReservation(MI, currentStateNumber, receiverChannel) = {
  let processID          = processIDFor(self) in
  let transitionNumber   = first_outgoingNormalTransition(processID, currentStateNumber),
      senderChannel      = channelFor(self),
      receiverProcessID  = processIDOf(receiverChannel) in
  let senderSubjectID    = searchSenderSubjectID(processID, subjectIDFor(self), receiverP
      msgCorrelationID   = loadCorrelationID(MI, messageNewCorrelationVar(processID, tran
      ipCorrelationID    = loadCorrelationID(MI, messageWithCorrelationVar(processID, tra
  let reservationMessage = [senderChannel, messageType(processID, transitionNumber), {},
      IP = inputPool(receiverChannel, senderSubjectID, messageType(processID, transitionN
      inputPool(receiverChannel, senderSubjectID, messageType(processID, transitionNumber
  }
}
```

Listing 46: CancelReservation

```
rule ReplaceReservation(MI, currentStateNumber, receiverChannel) = {
  let processID          = processIDFor(self) in
  let transitionNumber   = first_outgoingNormalTransition(processID, currentStateNumber),
      senderChannel      = channelFor(self),
      receiverProcessID  = processIDOf(receiverChannel) in
  let senderSubjectID    = searchSenderSubjectID(processID, subjectIDFor(self), receiverP
      msgCorrelationID   = loadCorrelationID(MI, messageNewCorrelationVar(processID, tran
      ipCorrelationID    = loadCorrelationID(MI, messageWithCorrelationVar(processID, tra
  let reservationMessage = [senderChannel, messageType(processID, transitionNumber), {},
      message            = [senderChannel, messageType(processID, transitionNumber), mess
      IP = inputPool(receiverChannel, senderSubjectID, messageType(processID, transitionN
      // TODO: discuss: setnth or dropnth & enqueue?
      inputPool(receiverChannel, senderSubjectID, messageType(processID, transitionNumber
  }
}
```

Listing 47: ReplaceReservation

```
rule PerformReceive(MI, currentStateNumber) = {
  // startTime must be the time of the first attempt to receive in order to support r
  if (startTime(channelFor(self), MI, currentStateNumber) = undef) then {
    StartTimeout(MI, currentStateNumber)

    SetExecutionState(MI, currentStateNumber, REPEAT)
  }
  else {
    if (shouldTimeout(channelFor(self), MI, currentStateNumber) = true) then {
      SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
      ActivateTimeout(MI, currentStateNumber)
    }
    else {
      let processID = processIDFor(self) in
      seq
        forall transitionNumber in outgoingNormalTransitions(processID, currentStateN
          CheckIP(MI, currentStateNumber, transitionNumber)
        }
      next
        let enabledOutgoingTransitions = outgoingEnabledTransitions(channelFor(self),
          if (|enabledOutgoingTransitions| > 0) then {
            seqblock
            debuginfo PerformReceive self + ": at least one IP with messages :)"

            if (selectedTransition(channelFor(self), MI, currentStateNumber) != undef
              debuginfo PerformReceive self + ": skipping automatic decision, there i
            }
            else if (|enabledOutgoingTransitions| = 1) then {
              let transitionNumber = firstFromSet(enabledOutgoingTransitions) in {
                if (transitionIsAuto(processID, transitionNumber) = true) then {
                  debuginfo PerformReceive self + ": making automatic decision for tr
                  selectedTransition(channelFor(self), MI, currentStateNumber) := tra
                }
                else {
                  debuginfo PerformReceive self + ": can not make automatic decision,
                }
              }
            }
            else {
              debuginfo PerformReceive self + ": can not make automatic decision, too
            }

            if (selectedTransition(channelFor(self), MI, currentStateNumber) != undef
              debuginfo PerformReceive self + ": the decision has been made.."
              SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
            }
            else {
              debuginfo PerformReceive self + ": no decision made, waiting for selecti

              SelectTransition(MI, currentStateNumber)
            }
            endseqblock
          }
          else {
            debuginfo PerformReceive self + ": no IP with messages, trying later.."
            SetExecutionState(MI, currentStateNumber, LOWER)
          }
        }
    }
  }
}
```

```
rule PerformTransitionReceive(MI, currentStateNumber, transitionNumber) = {
  ReceiveMessage(MI, currentStateNumber, transitionNumber)
}
```

Listing 49: PerformTransitionReceive

```
rule ReceiveMessage(MI, currentStateNumber, transitionNumber) = {
  let processID   = processIDFor(self) in
  let s           = messageSubjectId         (processID, transitionNumber),
      sChsVarname = messageSubjectVar        (processID, transitionNumber),
      mt          = messageType              (processID, transitionNumber),
      cIDVarname  = messageWithCorrelationVar(processID, transitionNumber) in
  // TODO 2019-02-22: local receivedMessages ? Or is that function used elsewhere?
  // Alternative: directly return as listres via result in InputPool_Pop?
  local stringres1, // subjectID
      setres1,    // subjectChannels
      stringres2, // messageType
      numres1 in  // correlationID
  seqblock

      debuginfo ReceiveMessage self + ": ReceiveMessage in state " + statePretty(MI,

      // TODO/NOTE: same structure as CheckIP. May refactor to reduce duplicated code

      if (s = "*") then {
        debuginfo ReceiveMessage self + ": wildcard for subject is ? and not *"
        Crash()
      }
      else if (s = "?") then {
        stringres1 := undef
      }
      else {
        stringres1 := s
      }


      if (sChsVarname = undef or sChsVarname = "") then {
        setres1 := undef
      }
      else {
        seq
          setres1 := loadChannelsFromVariable(MI, sChsVarname, s) // (stringres1 woul
        next
          debuginfo CheckIP self + ": considering only messages from the following ch
      }


      if (mt = "*") then {
        debuginfo ReceiveMessage self + ": wildcard for message type is ? and not *"
        Crash()
      }
      else if (mt = "?") then {
        stringres2 := undef
      }
      else {
        stringres2 := mt
      }


      if (cIDVarname = "*") then {
        debuginfo ReceiveMessage self + ": wildcard for cID type is ? and not *"
        Crash()
      }
      else if (cIDVarname = "?") then {
        numres1 := undef
      }
      else {
```

```
rule CheckIP(MI, currentStateNumber, transitionNumber) = {
  let processID   = processIDFor(self) in
  let sID         = messageSubjectId         (processID, transitionNumber),
      sChsVarname = messageSubjectVar        (processID, transitionNumber),
      mT          = messageType              (processID, transitionNumber),
      cIDVarname  = messageWithCorrelationVar(processID, transitionNumber) in
  local stringres1, // subjectID
      setres1,    // subjectChannels
      stringres2, // messageType
      numres1 in  // correlationID
  seqblock

      // TODO/NOTE: same structure as ReceiveMessage. May refactor to reduce duplicated c

      if (sID = "?") then {
        stringres1 := undef
      }
      else if (sID = "*") then {
        debuginfo CheckIP self + ": subject wildcard is '?'"
        Crash()
      }
      else {
        stringres1 := sID
      }


      if (sChsVarname = undef or sChsVarname = "") then {
        setres1 := undef
      }
      else {
        seq
          setres1 := loadChannelsFromVariable(MI, sChsVarname, sID) // (stringres1 would
        next
          debuginfo CheckIP self + ": considering only messages from the following channe
      }


      if (mT = "?") then {
        stringres2 := undef
      }
      else if (mT = "*") then {
        debuginfo CheckIP self + ": message type wildcard is '?'"
        Crash()
      }
      else {
        stringres2 := mT
      }


      if (cIDVarname = "?") then {
        numres1 := undef
      }
      else if (cIDVarname = "*") then {
        debuginfo CheckIP self + ": correlationID wildcard is '?'"
        Crash()
      }
      else {
        numres1 := loadCorrelationID(MI, messageWithCorrelationVar(processID, transitionN
      }
```

```
rule StartEnd(MI, currentStateNumber) = {
  SetExecutionState(MI, currentStateNumber, REPEAT)
}
```

Listing 52: StartEnd

```
rule PerformEnd(MI, currentStateNumber) = {
  if (|activeStates(channelFor(self), MI)| > 1) then {
      AbortMacroInstance(MI, currentStateNumber) // do not remove self. calls ClearAl

      SetExecutionState(MI, currentStateNumber, DONE)
  }
  else {
      if (MI = 1) then {
        let res = head(stateFunctionArguments(processIDFor(self), currentStateNumber)
          // just for debugging purposes, a termination of the Main Macro should not
          if (res = undef) then { // no parameters for End state
              debuginfo PerformEnd self + ": within mainMacro, terminate subject with
          }
          else {
              debuginfo PerformEnd self + ": within mainMacro. WARN: terminate subjec
          }
        }

        ClearAllVarInMIForChannel(channelFor(self), 0)
        ClearAllVarInMIForChannel(channelFor(self), 1)

        FinalizeInteraction()

        program(self) := undef
        remove self from asmAgents
      }
      else {
        ClearAllVarInMIForChannel(channelFor(self), MI)

        let res = head(stateFunctionArguments(processIDFor(self), currentStateNumber)
          if (res = undef) then { // no parameters for End state
              debuginfo PerformEnd self + ": terminated without result value"
              macroTerminationResult(channelFor(self), MI) := true
          }
          else {
              debuginfo PerformEnd self + ": terminated with result value: " + res
              macroTerminationResult(channelFor(self), MI) := res
          }
        }
      }

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

Listing 53: PerformEnd

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

rule Tau(MI, currentStateNumber, args) = {
  let processID = processIDFor(self) in {
      choose transitionNumber in outgoingEnabledTransitions(channelFor(self), MI, current
        debuginfo Tau self + ": transition chosen, it is normal and auto: " + transitionN

        selectedTransition(channelFor(self), MI, currentStateNumber) := transitionNumber

        SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
      }
      ifnone {
        debuginfo Tau self + ": unable to choose auto transition!"

        if (selectedTransition(channelFor(self), MI, currentStateNumber) != undef) then {
          debuginfo Tau self + ": selectedTransition had been set"
          SetCompleted(MI, currentStateNumber) // sets executionState to REPEAT
        }
        else {
          debuginfo Tau self + ": SelectTransition"
          SelectTransition(MI, currentStateNumber)
        }
      }
  }
}
```

Listing 54: Tau

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

rule VarMan(MI, currentStateNumber, args) = {
  let method = head(args) in {
      debuginfo VarMan self + ": method: " + method
      debuginfo VarMan self + ": args: " + tail(args)

      case method of
        "assign"                : VarMan_Assign               (MI, currentStateNumber
        "storeData"             : VarMan_StoreData            (MI, currentStateNumber
        "clear"                 : VarMan_Clear                (MI, currentStateNumber

        "concatenation"         : VarMan_Concatenation        (MI, currentStateNumber
        "intersection"          : VarMan_Intersection         (MI, currentStateNumber
        "difference"            : VarMan_Difference           (MI, currentStateNumber

        "extractContent"        : VarMan_ExtractContent       (MI, currentStateNumber
        "extractChannel"        : VarMan_ExtractChannel       (MI, currentStateNumber
        "extractCorrelationID"  : VarMan_ExtractCorrelationID (MI, currentStateNumber

        "selection"             : VarMan_Selection            (MI, currentStateNumber
      endcase
  }
}


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

      SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REP
  }
}

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

  SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REPEAT
}

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

  SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REPEAT
}

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)

        SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to R
      }
      else if (a = undef) then {
        SetVar(MI, X, head(b), last(b))

        SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to R
```

```
// CH * 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, srcVarname, dstVarname, minimum, maximum) =
  let src = loadVar(MI, srcVarname),
    res = selectionResult(channelFor(self), MI, currentStateNumber) in
  if (res = undef) then {
    // TODO: cancel / timeout transition?
    Selection(MI, currentStateNumber, src, minimum, maximum)
  }
  else {
    selectionResult(channelFor(self), MI, currentStateNumber) := undef

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

    SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REPEAT
  }
}

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

rule Selection(MI, currentStateNumber, src, minimum, maximum) = {
  if (selectionData(channelFor(self), MI, currentStateNumber) = undef) then {
      if (head(src) = "MessageSet") then {
        let l = toList(last(src)) in {
          selectionData   (channelFor(self), MI, currentStateNumber) := l
          selectionOptions(channelFor(self), MI, currentStateNumber) := map(l, @msgToStri
        }
      }
      else if (head(src) = "ChannelInformation") then {
        let l = toList(last(src)) in {
          selectionData   (channelFor(self), MI, currentStateNumber) := l
          selectionOptions(channelFor(self), MI, currentStateNumber) := map(l, @chToStrin
        }
      }
      else {
        debuginfo Selection self + ": can not perform selection on datatype '" + head(x)

        Crash()
      }

      selectionVartype (channelFor(self), MI, currentStateNumber) := head(src)
      selectionMin     (channelFor(self), MI, currentStateNumber) := minimum
      selectionMax     (channelFor(self), MI, currentStateNumber) := maximum
      selectionDecision(channelFor(self), MI, currentStateNumber) := undef // just to be

      SetExecutionState(MI, currentStateNumber, REPEAT)
```

```
rule ModalSplit(MI, currentStateNumber, args) = {
  seqblock
    // start each following state
    foreach transitionNumber in outgoingNormalTransitions(processIDFor(self), current
      let sNew = targetStateNumber(processIDFor(self), transitionNumber) in {
        AddState(MI, currentStateNumber, MI, sNew)
      }
    }

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

    SetExecutionState(MI, currentStateNumber, DONE)
  endseqblock
}
```

Listing 57: ModalSplit

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

rule ModalJoin(MI, currentStateNumber, args) = {
  let splitCount = nth(args, 1) in
  seqblock
    debuginfo ModalJoin self + ": state: " + statePretty(MI, currentStateNumber)
    debuginfo ModalJoin self + ": splitCount: " + splitCount

    if (joinCount(channelFor(self), MI, currentStateNumber) = undef) then {
      joinCount(channelFor(self), MI, currentStateNumber) := 1
    }
    else {
      joinCount(channelFor(self), MI, currentStateNumber) := joinCount(channelFor(sel
    }

    debuginfo ModalJoin self + ": joinCount_post: " + joinCount(channelFor(self), MI,

    // can we continue, or remove self and wait for next path joining?
    if (joinCount(channelFor(self), MI, currentStateNumber) < splitCount) then {
      // remove self
      RemoveState(MI, currentStateNumber, MI, currentStateNumber)

      SetExecutionState(MI, currentStateNumber, DONE)
    }
    else {
      joinCount(channelFor(self), MI, currentStateNumber) := undef
      SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REP
    }
  endseqblock
}
```

Listing 58: ModalJoin

```
rule AbortCallMacro(MI, currentStateNumber) = {
  let childInstance = callMacroChildInstance(channelFor(self), MI, currentStateNumber) in
    if (|activeStates(channelFor(self), childInstance)| > 0) then {
      AbortMacroInstance(childInstance, undef)

      SetExecutionState(MI, currentStateNumber, DONE)
    }
    else {
      callMacroChildInstance(channelFor(self), MI, currentStateNumber) := undef

      SetAbortionCompleted(MI, currentStateNumber)  // sets executionState to DONE
    }
  }
}
```

Listing 59: AbortCallMacro

```
rule InitializeMacroArguments(MI, currentStateNumber, mIDNew, MINew, macroArgumentsValues
  local
    listres1 := macroArguments(processIDFor(self), mIDNew),
    listres2 := macroArgumentsValues in
  {
    if (|listres1| != |listres2|) then {
      debuginfo CallMacro self + ": Macro '"+macroID(processIDFor(self), mIDNew)+"' takes
      Crash()
    }

    while (|listres1| > 0) do {
      let varnameDst = head(listres1),
        varnameSrc = head(listres2) in
      let var = loadVar(MI, varnameSrc) in
      {
        if (var = undef) then {
          debuginfo CallMacro self + ": skipped local variable '" + varnameDst + "' from
        }
        else {
          debuginfo CallMacro self + ": load local variable '" + varnameDst + "' from '"
          SetVar(MINew, varnameDst, nth(var, 1), nth(var, 2))
        }
      }

      listres1 := tail(listres1)
      listres2 := tail(listres2)
    }
  }
}
```

Listing 60: InitializeMacroArguments

```
rule CallMacro(MI, currentStateNumber, args) = {
  let childInstance = callMacroChildInstance(channelFor(self), MI, currentStateNumber
    // if Macro is not yet running..
    if (childInstance = undef) then {
      // TODO: consider to move this to a new rule StartCallMacro
      let mIDNew = searchMacro(head(args)),
        MINew  = nextMacroInstanceNumber(channelFor(self)) in seqblock
        nextMacroInstanceNumber(channelFor(self)) := MINew + 1
        macroNumberOfMI(channelFor(self), MINew) := mIDNew
        callMacroChildInstance(channelFor(self), MI, currentStateNumber) := MINew

        // NOTE: macroTerminationResult doesn't need to be initialized
        // as the MI part will be different in each iteration

        // if the Macro has parameters..
        if (|macroArguments(processIDFor(self), mIDNew)| > 0) then
        {
          InitializeMacroArguments(MI, currentStateNumber, mIDNew, MINew, tail(args))
        }

        SetExecutionState(MI, currentStateNumber, DONE)

        StartMacro(MI, currentStateNumber, mIDNew, MINew)
      endseqblock
    }
    else {
      debuginfo CallMacro self + ": childInstance: " + childInstance
      let childResult = macroTerminationResult(channelFor(self), childInstance) in {
        if (childResult != undef) then {
          debuginfo CallMacro self + ": childResult: " + childResult

          callMacroChildInstance(channelFor(self), MI, currentStateNumber) := undef

          if (childResult = true) then { // completed without result
            SetCompletedAction(MI, currentStateNumber, undef) // sets executionState
          }
          else {
            SetCompletedAction(MI, currentStateNumber, childResult) // sets execution
          }
        }
        else seqblock
          MacroBehaviour(childInstance)

          let mState = macroExecutionState(channelFor(self), childInstance) in
            SetExecutionState(MI, currentStateNumber, mState)

          // reset
          macroExecutionState(channelFor(self), childInstance) := undef
        endseqblock
      }
    }
  }
}
```

Listing 61: CallMacro

```
rule CheckCancel(MI, currentStateNumber, transitionNumber) = {
  let processID = processIDFor(self) in
  let tName = transitionLabel(processID, transitionNumber) in
  let nCancel = stateNumberFromID(processID, tName) in {
    if (contains(activeStates(channelFor(self), MI), nCancel) = true) then {
      debuginfo CheckCancel self + ": at least one state active!" // at least? if nothing
      EnableTransition(MI, transitionNumber)
    }
    else {
      debuginfo CheckCancel self + ": currently no state active!"
      DisableTransition(MI, currentStateNumber, transitionNumber)
    }
  }
}
```

Listing 62: CheckCancel

```
rule Cancel(MI, currentStateNumber, args) = {
  let processID = processIDFor(self) in
  seqblock
    forall transitionNumber in outgoingNormalTransitions(processID, currentStateNumbe
      CheckCancel(MI, currentStateNumber, transitionNumber)
    }

    let enabledOutgoingTransitions = outgoingEnabledTransitions(channelFor(self), MI,
      if (|enabledOutgoingTransitions| > 0) then {
        seqblock
        debuginfo Cancel self + ": at least one transition with active states :)"

        if (|enabledOutgoingTransitions| = 1) then {
          let transitionNumber = firstFromSet(enabledOutgoingTransitions) in {
            if (transitionIsAuto(processID, transitionNumber) = true) then {
              debuginfo Cancel self + ": making automatic decision for transition " +
              selectedTransition(channelFor(self), MI, currentStateNumber) := transit
            }
            else {
              debuginfo Cancel self + ": can not make automatic decision, not an auto
            }
          }
        }
        else {
          debuginfo Cancel self + ": can not make automatic decision, too much transi
        }

        if (selectedTransition(channelFor(self), MI, currentStateNumber) != undef) th
          debuginfo Cancel self + ": the decision has been made for: " + transitionPr

          SetCompletedAction(MI, currentStateNumber, transitionLabel(processID, selec
        }
        else {
          SelectTransition(MI, currentStateNumber)
        }
        endseqblock
      }
      else {
        debuginfo Cancel self + ": no transition with active states, trying later.."
        SetExecutionState(MI, currentStateNumber, LOWER)
      }
    }
  endseqblock
}
```

Listing 63: Cancel

```
rule PerformTransitionCancel(MI, currentStateNumber, transitionNumber) = {
  let processID = processIDFor(self) in {
    let tLabel = transitionLabel(processID, transitionNumber) in
    let nCancel = stateNumberFromID(processID, tLabel) in {
      cancelDecision(channelFor(self), MI, nCancel) := true

      SetCompletedTransition(MI, currentStateNumber, transitionNumber) // sets executionS
    }
  }
}
```

Listing 64: PerformTransitionCancel

```
// no wildcards allowed
rule CloseIP(MI, currentStateNumber, args) = {
  let senderSubjID         = nth(args, 1),
      messageType          = nth(args, 2),
      correlationIDVarname = nth(args, 3) in {
      if (messageType = "*" or messageType = "?" or senderSubjID = "*" or senderSubjID =
        debuginfo CloseIP self + ": no wildcards allowed. You may want to use CloseAllIPs
        Crash()
      }
      else {
        let correlationID = loadCorrelationID(MI, correlationIDVarname) in {
          inputPoolClosed(channelFor(self), senderSubjID, messageType, correlationID) :=
          if (inputPool(channelFor(self), senderSubjID, messageType, correlationID) = und
              add [senderSubjID, messageType, correlationID] to inputPoolDefined(channelF
              inputPool(channelFor(self), senderSubjID, messageType, correlationID) := []
          }
        }
      }

      SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REPEAT
  }
}
```

Listing 65: CloseIP

```
// no wildcards allowed
rule OpenIP(MI, currentStateNumber, args) = {
  let senderSubjID         = nth(args, 1),
      messageType          = nth(args, 2),
      correlationIDVarname = nth(args, 3) in {
      if (messageType = "*" or messageType = "?" or senderSubjID = "*" or senderSubjI
        debuginfo OpenIP self + ": no wildcards allowed. You may want to use CloseAll
        Crash()
      }
      else {
        let correlationID = loadCorrelationID(MI, correlationIDVarname) in {
          inputPoolClosed(channelFor(self), senderSubjID, messageType, correlationID)
          if (inputPool(channelFor(self), senderSubjID, messageType, correlationID) =
              add [senderSubjID, messageType, correlationID] to inputPoolDefined(chan
              inputPool(channelFor(self), senderSubjID, messageType, correlationID) :
          }
        }
      }

      SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REP
  }
}
```

Listing 66: OpenIP

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

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

  SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REPEAT
}
```

Listing 67: CloseAllIPs

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

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

  SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REPEAT
}
```

Listing 68: OpenAllIPs

```
// only correlation can be wildcard (*)
rule IsIPEmpty(MI, currentStateNumber, args) = {
  debuginfo IsIPEmpty self + ": args: " + args

  local numres in
  let senderSubjID         = nth(args, 1),
      messageType          = nth(args, 2),
      correlationIDVarname = nth(args, 3) in
  seqblock

      if (correlationIDVarname = undef or correlationIDVarname = 0 or correlationIDVarnam
        numres := 0
      }
      else if (correlationIDVarname = "*") then {
        numres := undef
      }
      else if (correlationIDVarname = "?") then {
        debuginfo OpenIP self + ": correlationIDVarname must not be '?'. wildcard is '*'"
        Crash()
      }
      else {
        numres := loadCorrelationID(MI, correlationIDVarname)
      }

      // receiverChannel * senderSubjID * messageType * correlationID
      if (inputPoolIsEmpty(channelFor(self), senderSubjID, messageType, numres) = true) t
        SetCompletedAction(MI, currentStateNumber, "true") // sets executionState to REPE
      }
      else {
        SetCompletedAction(MI, currentStateNumber, "false") // sets executionState to REP
      }
  endseqblock
}
```

Listing 69: IsIPEmpty

```
// 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
    varname  = nth(args, 1),
    sIDLocal = nth(args, 2),
    countMin = nth(args, 3),
    countMax = nth(args, 4) in
  {
    if (selectAgentsResult(channelFor(self), MI, currentStateNumber) != undef) then {
      SetVar(MI, varname, "ChannelInformation", selectAgentsResult(channelFor(self), I
      selectAgentsResult(channelFor(self), MI, currentStateNumber) := undef

      SetCompletedAction(MI, currentStateNumber, undef) // sets executionState to REP
    }
    else {
      SelectAgents(MI, currentStateNumber, sIDLocal, countMin, countMax)
    }
  }
}
```

Listing 70: SelectAgentsAction

```
rule SelectAgents(MI, currentStateNumber, sIDLocal, countMin, countMax) = {
  if (sIDLocal = undef or sIDLocal = "?") then {
    debuginfo SelectAgents self + ": sIDLocal must not be wildcard/undef"
    Crash()
  }

  let processID = processIDFor(self),
      PI        = processInstanceFor(self) in
  let resolvedInterface = resolveInterfaceSubject(sIDLocal) in
  let resolvedProcessID = nth(resolvedInterface, 1),
      resolvedSubjectID = nth(resolvedInterface, 2) in
  if (selectAgentsDecision(channelFor(self), MI, currentStateNumber) != undef) then {
      // validate min/max
      if (hasSizeWithin(selectAgentsDecision(channelFor(self), MI, currentStateNumber), c
        debuginfo SelectAgents self + ": selectAgentsDecision is not within countMin/coun
        Crash()
      }

      local setres1 := {} in // created channels
        seq
          foreach agent in selectAgentsDecision(channelFor(self), MI, currentStateNumber)
              if (resolvedProcessID = processID) then {
                // local process, use own PI
                let ch = [processID, PI, sIDLocal, agent] in {
                  InitializeSubject(ch)

                  add ch to setres1
                }
              }
              else {
                // external process, create new PI
                local numres1 in {
                  seq
                      numres1 <- StartProcess(resolvedProcessID, resolvedSubjectID, agent
                  next {
                      let ch = [resolvedProcessID, numres1, resolvedSubjectID, agent] in
                        add ch to setres1
                  }
                }
              }
          }
        next
          selectAgentsResult(channelFor(self), MI, currentStateNumber) := setres1

      selectAgentsDecision (channelFor(self), MI, currentStateNumber) := undef

      selectAgentsCountMin (channelFor(self), MI, currentStateNumber) := undef
      selectAgentsCountMax (channelFor(self), MI, currentStateNumber) := undef
      selectAgentsProcessID(channelFor(self), MI, currentStateNumber) := undef
      selectAgentsSubjectID(channelFor(self), MI, currentStateNumber) := undef

      SetExecutionState(MI, currentStateNumber, REPEAT)
  }
  else if(hasSizeWithin(predefinedAgents(processID, PI, sIDLocal), countMin, countMax) =
      debuginfo SelectAgents self + ": apply predefinedAgents: " + predefinedAgents(proce

      selectAgentsDecision(channelFor(self), MI, currentStateNumber) := predefinedAgents(

      SetExecutionState(MI, currentStateNumber, REPEAT)
  }
  else {
```

Bibliography

[Kee76] E. L. Keenan. 1976.