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

Thomas Schaller
xyz

Werner Schmidt
xyz

Robert Singer
FH JOANNEUM–University of Applied Sciences

Christian Stary
xyz

Florian Strecker
xyz

André Wolski
TU Darmstadt

Conny Zebold
Technische Hochschule Ingolstadt

Typeset by the authors

Production and publishing: XYZ

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

Short contents

Preface

Contents

CHAPTER 1

Foundation

Review CS: Chapter One should be a motivation including history of developements, rationale and targe

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). Additionally, 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 (CS: INTRODUCED FOR THE FIRST TIME and thus its meaning should be explained here) 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 PASS 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.

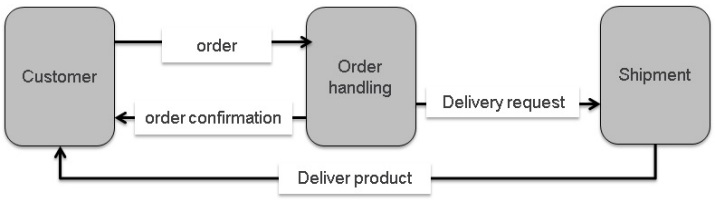


Figure 1.1: The Communication Structure in the Order Process

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.

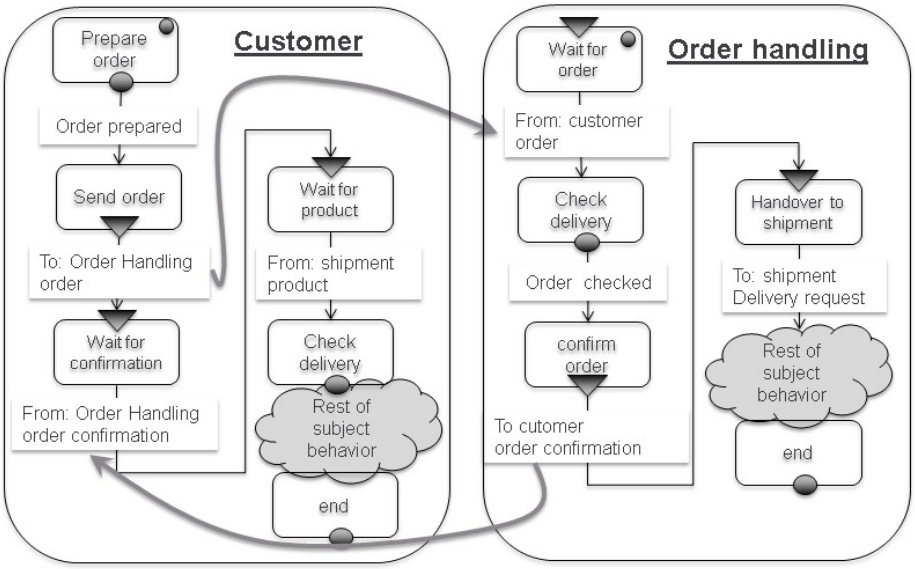


Figure 1.2: The Behavior of Subjects

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".

Review AF: Bild und Text passen nicht Change order !!!!!!!

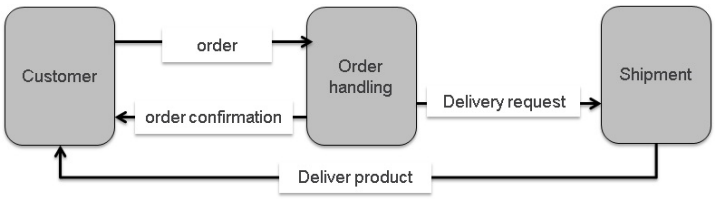


Figure 1.3: The Communication Structure with Change Message

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

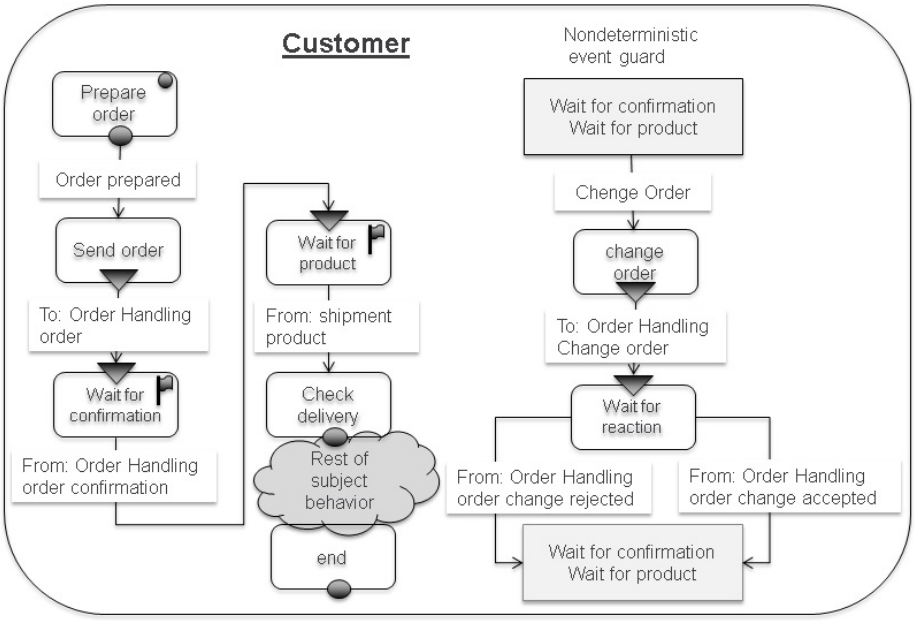


Figure 1.4: Customer is allowed to Change Orders

"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 **??** CS:Wrong reference link 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]).

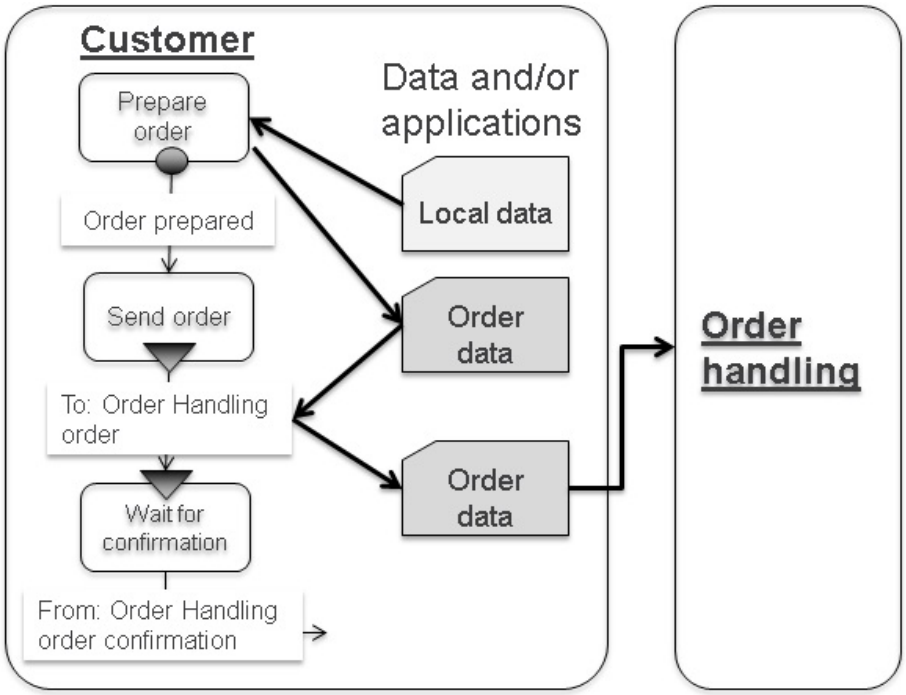


Figure 1.5: Subjects and Objects

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.

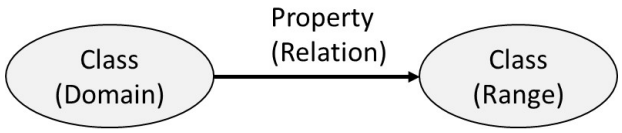


Figure 1.6: RDF graphic

RDF is based on the idea of making statements about resources (in particular web resources) in expressions of the form subjectâĂŞpredicateâĂŞobject, 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 BÃűrger, E., StÃd'rk 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.

Review CS: In this chapter, concepts are presented, that are not all part of the diagrammatic represen

2.1 Informal Description

2.1.1 Subject

As already detailed previously, 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.

CS: would it not be more convenient to move the conceptual understanding to section 1.1 and then just refer to this part, instead of rephrasing the same concept several times?

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 is 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 (CS: this term needs explanation or other wording, e.g., "other context"). 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 informationâĂŤso-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).

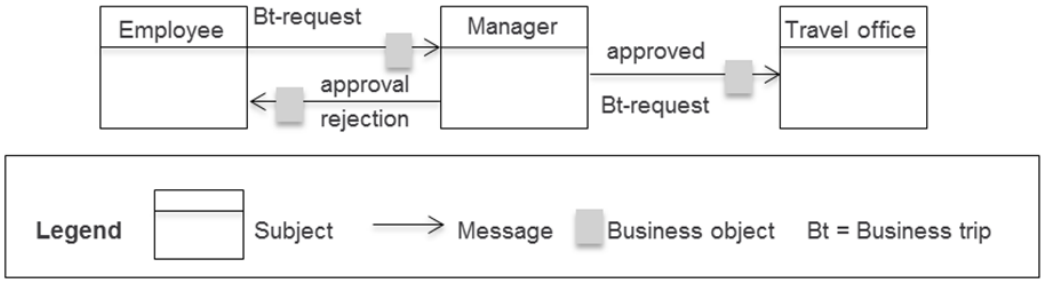


Figure 2.1: Subject interaction diagram for the process 'business trip application'

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 synchronous 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.

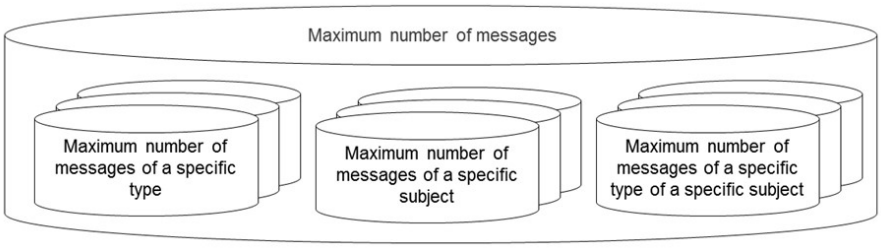


Figure 2.2: Configuration of Input Pool Parameters

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 poolâĂŹs 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

ÃIJbergang von informeller zur formalen Beschreibung sehr abrupt und unverstaÃd'ndlich

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.

Nummerierung erklÃd'ren, Wie kommen Nummern zustande

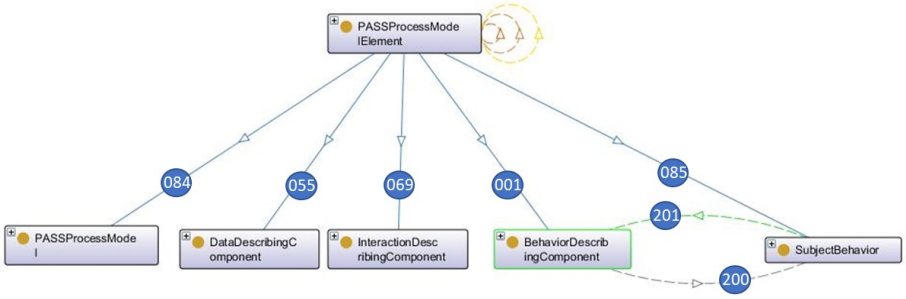


Figure 2.3: Elements of PASS Process Models

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, namely 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.

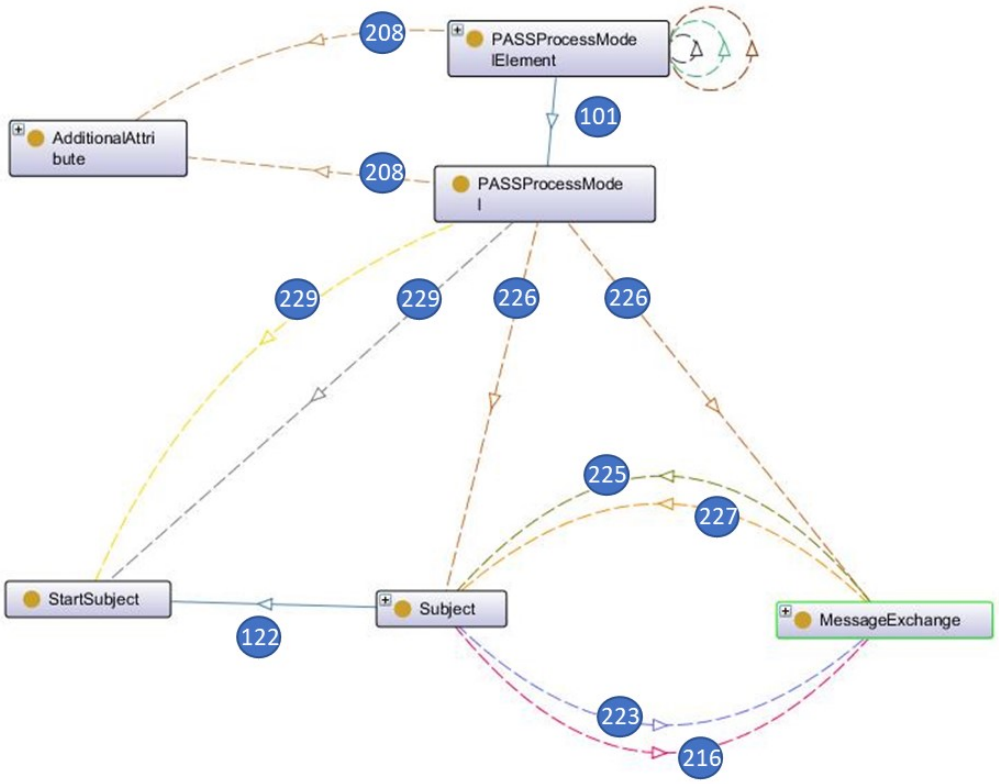


Figure 2.4: PASS Process Modell

`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` 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 `DtadescribingComponent` (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

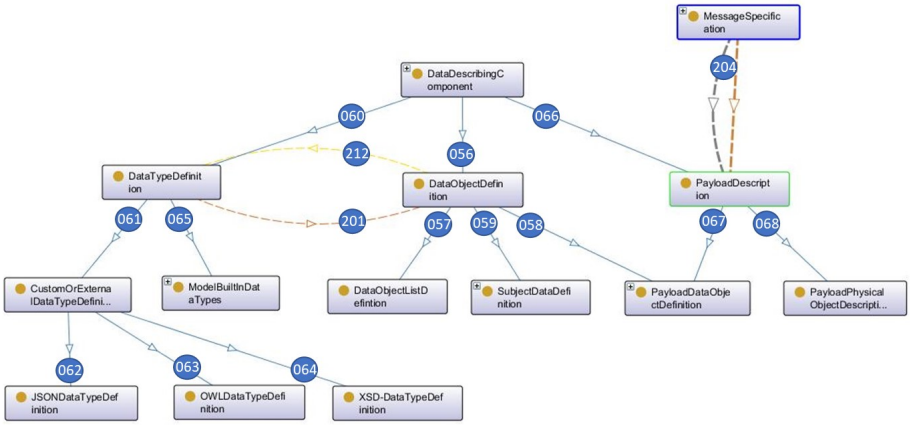


Figure 2.5: Data Description Component

objects are also a subclass of the class `DataObjectDefinition` (Subclass relation 058 in figure 2.5).

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

2.2.3 Interaction Describing Component

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

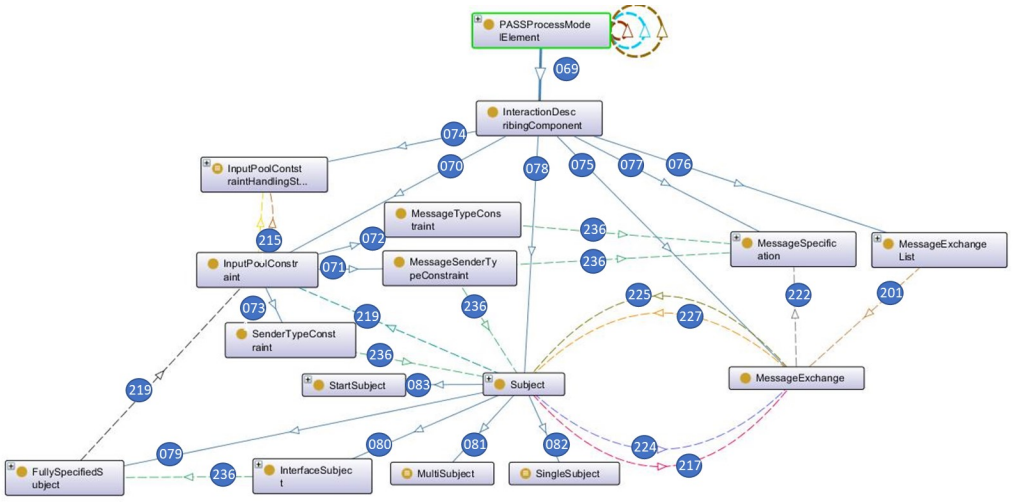


Figure 2.6: Subject Interaction Diagram

The central classes are `Subject` and `MessageExchange`. Between these classes are defined the properties `hasIncomingTransition` (in figure 2.6 number 217) and `hasOutgoingTransition` (in figure 2.6 number 224). These properties define 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

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.

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,

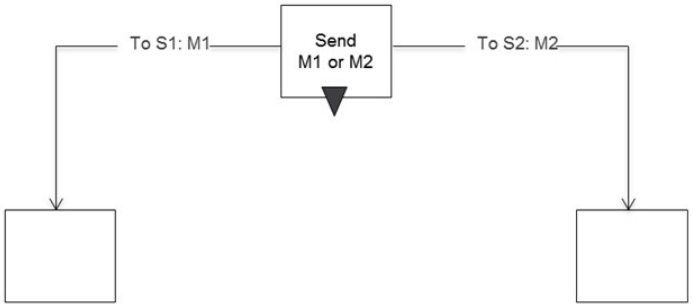


Figure 3.1: Example of alternative sending

the process immediately follows the timeout path when the alternative message delivery fails.

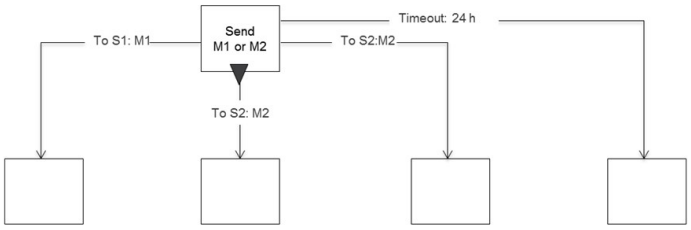


Figure 3.2: Send using time monitoring

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.

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

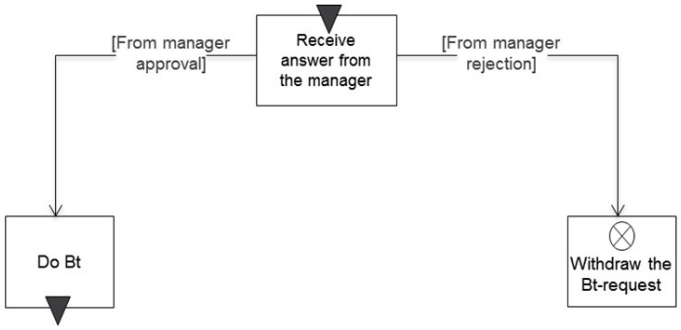


Figure 3.3: Example of alternative receiving

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.

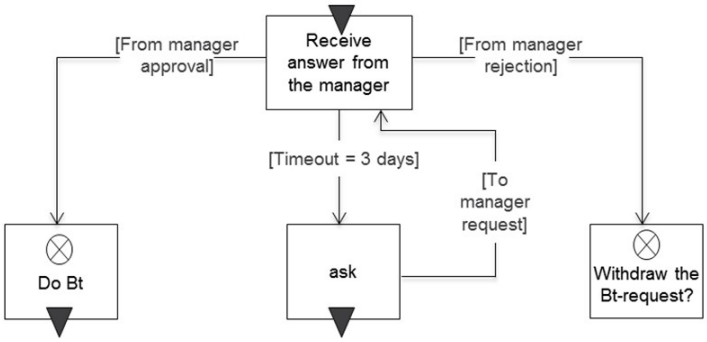


Figure 3.4: Time monitoring for message reception

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.

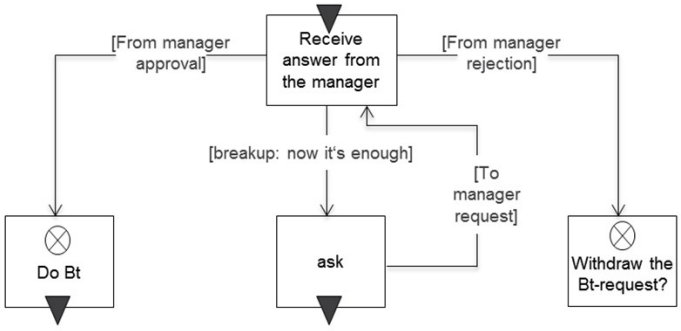


Figure 3.5: Message reception with manual interrupt

3.1.3 Standard Subject Behavior

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

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

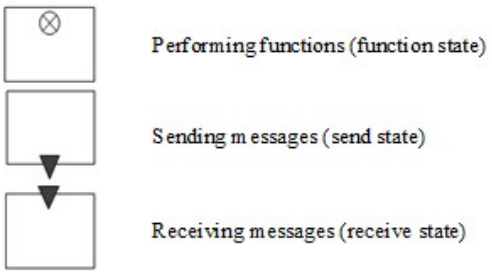


Figure 3.6: State types and coresponding symbols

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.

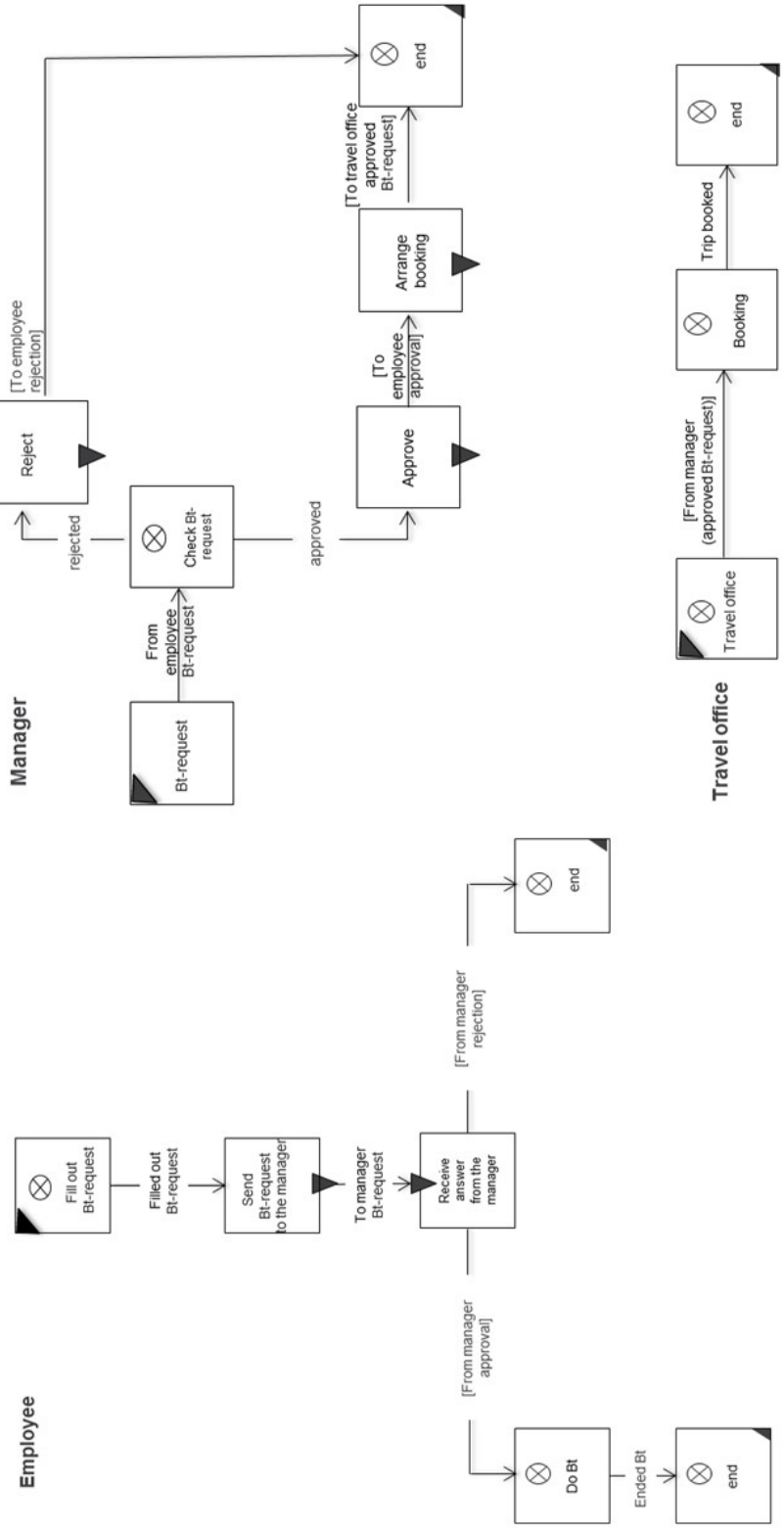


Figure 3.7: Subject behavior diagram for the subjects 'employee', 'manager', and 'travel office'

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.

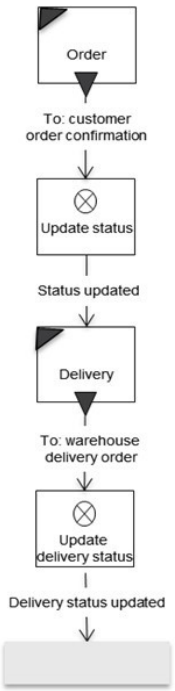


Figure 3.8: Behavior macro class 'request for approval'

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'.

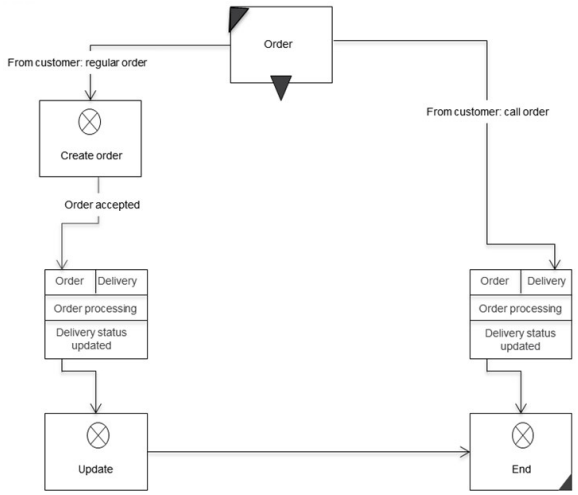


Figure 3.9: Subject behavior for order processing with macro integration

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 orderâĂŤan 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.

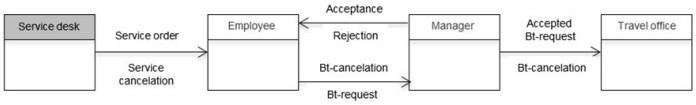


Figure 3.10: Communication structure diagram (CSD) of the business trip application

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

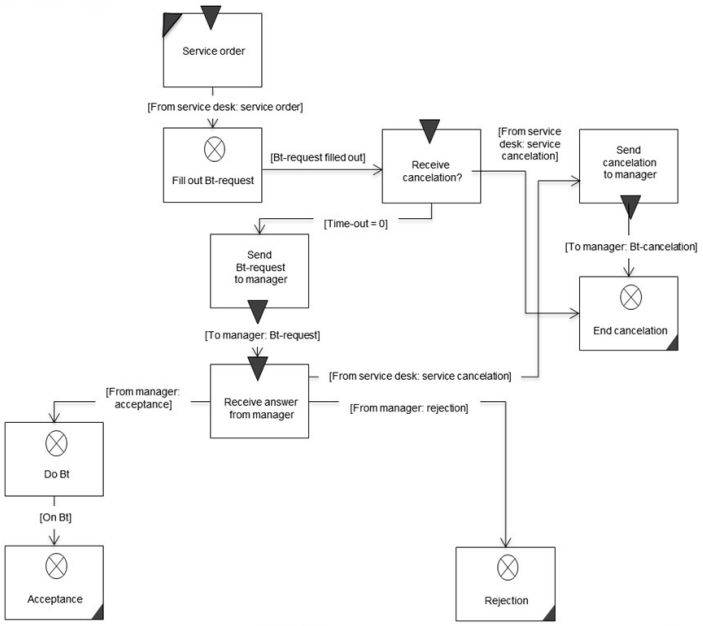


Figure 3.11: Handling the cancellation message using existing constructs

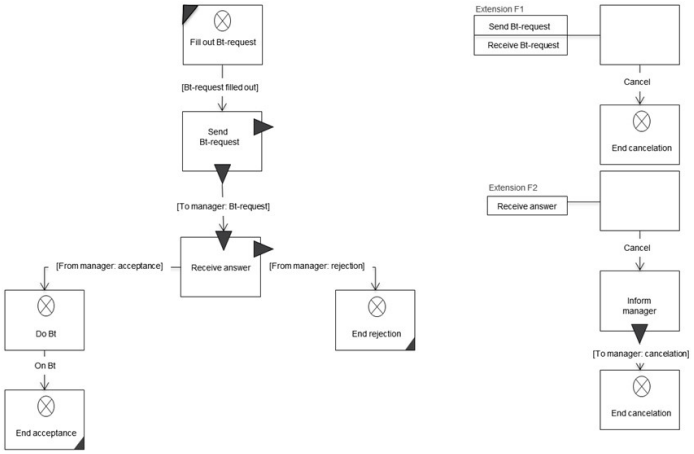


Figure 3.12: Behavior of subject 'employee' with exception handling

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 cancellation message is received, and the business trip application is only partially completed. Such consequences are considered acceptable, due to the urgency of cancellation messages. In less urgent cases, the modeler would like to extend the behavior of subjects in a similar way, however, without causing inconsistencies. This can be achieved by using a notation analogous to exception handling. Instead of denoting the corresponding diagram with 'exception', it is labeled with 'extension'.

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

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

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

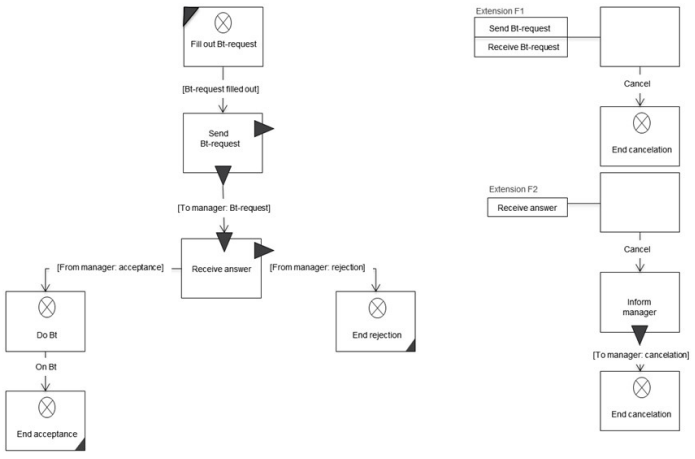


Figure 3.13: Subject behavior of employee with behavior extensions

Alternative Actions (Freedom of Choice)

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

Certain sequences of actions can be executed overlapping (CS: interleaving?). 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, and in case of doing so, it must be completed.
- Beginning is open/end is open: Alternative may be processed but does not have to be completed.

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

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

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

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

The individual actions in the alternative paths of an alternative clause may be arbitrarily executed in parallel and overlapping (CS: interleving?), or in other words: A step can be executed in an alternative sequence, and then be followed

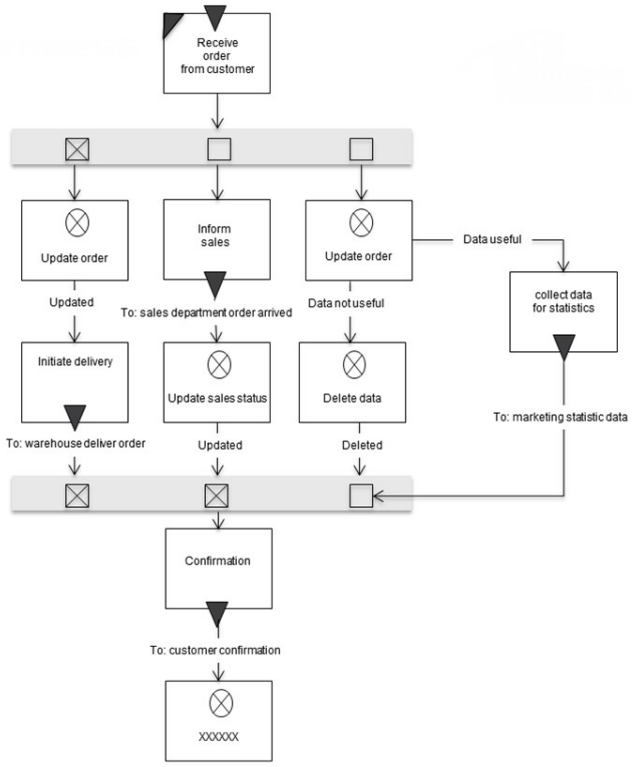


Figure 3.14: Example of Process Alternatives

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 alternativesâĂŤsituations 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).

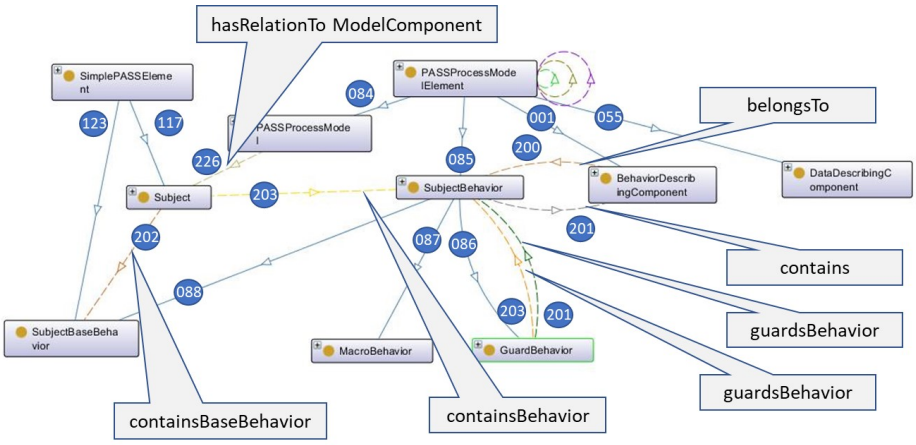


Figure 3.15: Structure of Subject Behavior Specification

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 and 024 in 3.16). The standard states `DoSTates`, `SendState` and `ReceiveState` are subclasses of the class `StandardPASSState` (class relations 114, 115 and 116 in 3.16). The subclass relations 104 and 020 allow that there exists a start state (class `InitialStatOfBehavior` in 3.16) and none or several end states (see subclass relation 020 in figure3.16) The fact that there must be at least one start state and none or several end states is defined by so called axioms which are not shown in figure 3.16.

States can be starting and/or endpoints of transitions (see properties 228 and 230 in figure 3.16). This means a state may have outgoing and/or incoming transitions (see properties 224 and 217 in figure 3.16). Each transition is controlled by a transition condition which must be true before a behavior follows a transition from the source state to the target state.

States

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

More complex states are choice segments (subclass relation 014). A choice segment contains choice segment paths (subclass 015 and property 200). Each choice segment path can be of one of four types. If a segment path is started

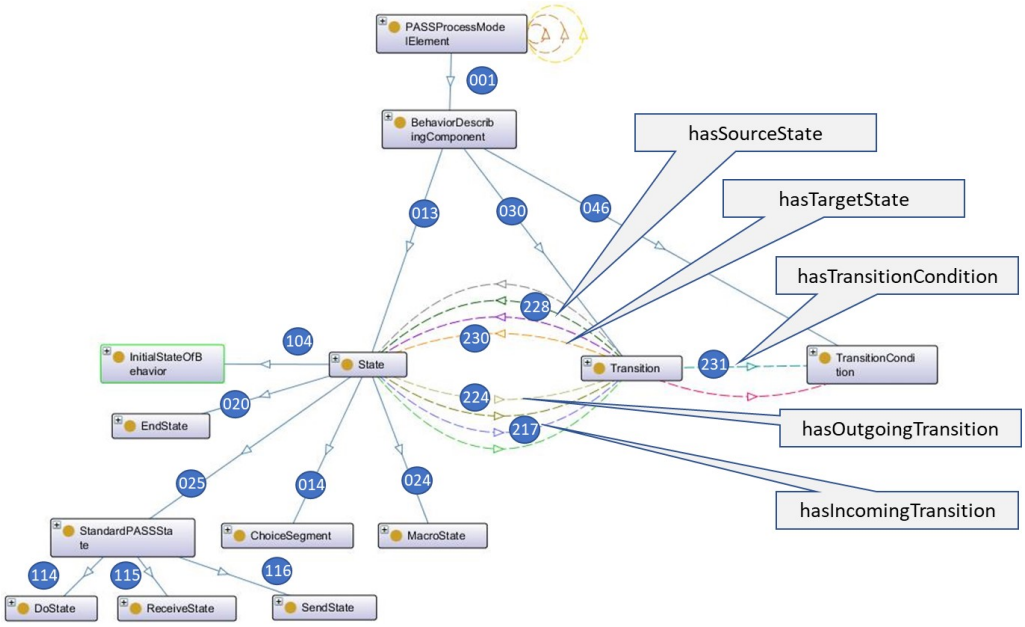


Figure 3.16: Subject Behavior describingComponent

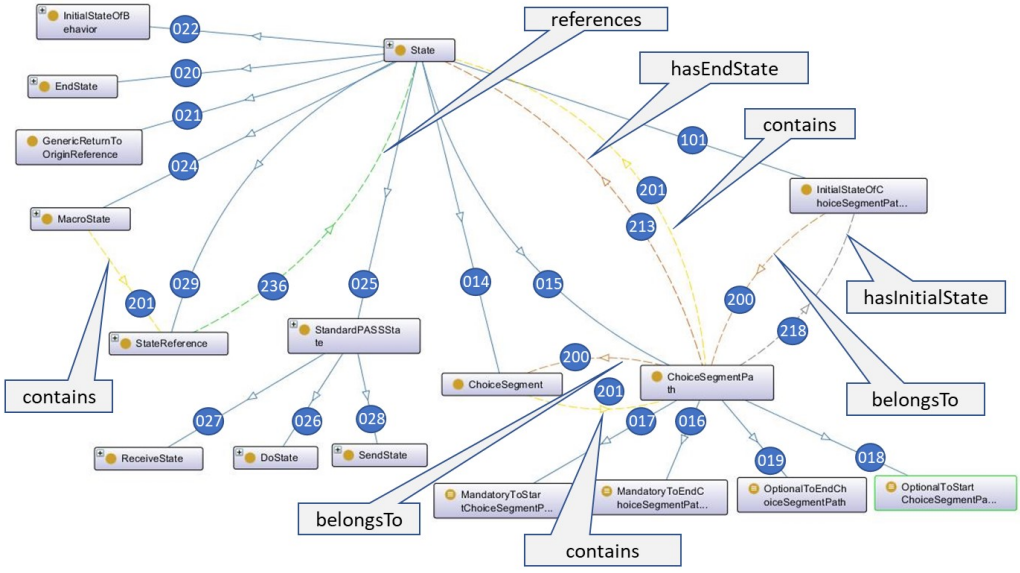


Figure 3.17: Details of States

then 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 order to execute a transition.

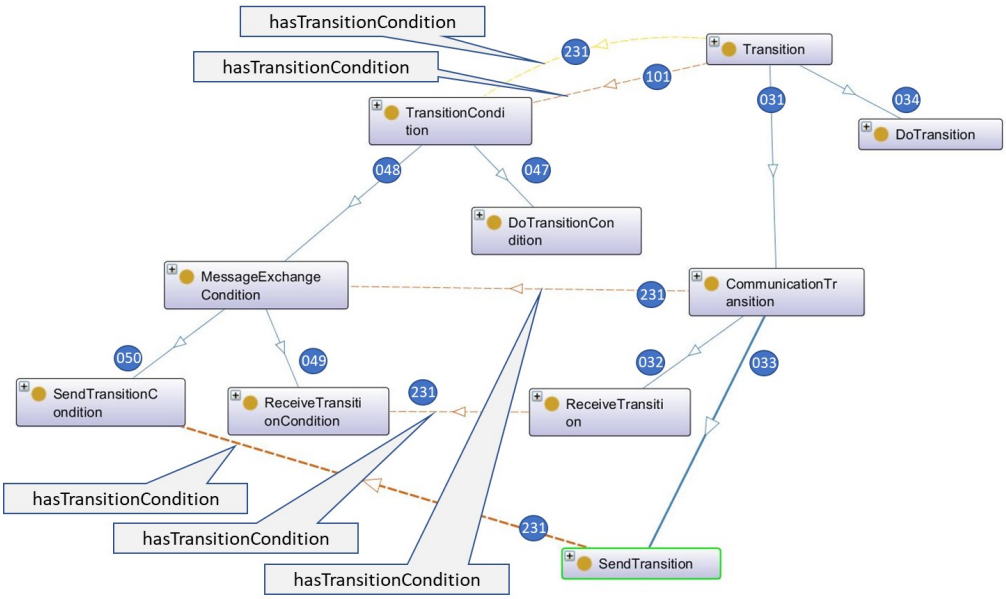


Figure 3.18: Details of transitions

3.3 ASM Definition of Subject Execution

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

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

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

3.3.1 Architecture

The reference implementation is composed by three main components. The specification of the execution semantics is defined with Abstract State Machines (ASM) and written in the CoreASM dialect. The specification is interpreted by the CoreASM Engine, which is part of the open-source CoreASM Framework and written for the Java Virtual Machine. The third component is a console application that is connected to the CoreASM Engine and enables an interactive abstract process evaluation. A detailed description of the architecture is given in C.2.

3.3.2 Foundation

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

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

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

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

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

Listing 1: Channel definitions

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

Analogous to programming languages Data Objects are called *Variables*. Their scope is either bound to a Subject or a Macro Instance and can only be accessed

by the Subject. Variables are identified by their name and have an explicit data type and a value.

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

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

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

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

Listing 2: variable

3.3.3 Interaction Definitions

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

Messages

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

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

Inputpool

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

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

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

The function `derived availableMessages(receiverChannel, senderSubjectID, msgType, ipCorrelationID)` returns the Messages from the

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

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

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

Listing 3: inputPool

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

3.3.4 Subject Behavior

The Subject Behavior consists at least of one *Main Macro* and might have an arbitrary number of minor Macros, called *Additional Macros*.

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

Listing 4: SubjectBehavior

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

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

Listing 5: MacroBehavior

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

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

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

In the beginning the `Behavior` rule initializes the state with the `StartState` rule, which will set `initializedState` to `true`. If the function should not be

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

Listing 6: Behavior

aborted the `Perform` rule calls the state behavior of the underlying function until it is `completed`. In the next phase the selected transition will be initialized by the `StartSelectedTransition` rule and the transition behavior will be performed with the `PerformTransition` rule until it is completed as well. As last step the `Proceed` rule removes the current state and adds the selected transition's target state.

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

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

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

Listing 7: wantInput

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

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

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

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

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

Listing 8: SelectTransition

3.3.5 Internal Action

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

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

  EnableAllTransitions(MI, currentStateNumber)
}
```

Listing 9: StartInternalAction

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

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

Listing 10: PerformInternalAction

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

3.3.6 Send Function

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

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

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

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

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

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

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

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

Listing 11: receivedMessages

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

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

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

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

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

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

Listing 12: StartSend

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

Listing 13: PerformSend

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

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

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

Listing 14: SetMessageContent

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

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

Listing 15: DoReservations

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

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

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

After all reservations could be placed the TryCompletePerformSend rule has to ensure that no receiver terminated *non-proper* in the meantime. In that case

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

Listing 16: DoReservation

the Send Function blocks and can only be left by a Timeout or Cancel transition. Otherwise the state function is completed and the behavior can continue with the transition function.

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

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

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

    SetCompleted(MI, s)
  }
```

Listing 17: TryCompletePerformSend

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

      EnsureRunning(r)
    }

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

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

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

Listing 18: PerformTransitionSend

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

Listing 19: ReplaceReservation

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

    SetAbortionCompleted(MI, s)
  }
```

Listing 20: AbortSend

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

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

3.3.7 Receive Function

The Receive Function retrieves Messages from the input pool. The state function updates the enabled outgoing transitions according to the available Messages in the corresponding queue. When an outgoing transition is selected the Messages are removed from the queue by the transition function.

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

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

The `CheckIP` rule loads the parameters from the communication transition attributes to determine the corresponding queue and the available Messages in it. The `messageSubjectCountMin` argument defines the minimal required number of different sender Agents of a MultiSubject. If the optional parameter `messageWithCorrelationVar` is not set, the CorrelationID 0 is used.

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

Listing 21: CancelReservation

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

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

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

3.3.8 Modal Split and Modal Join Functions

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

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

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

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

3.3.9 CallMacro Function

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

The `callMacroChildInstance` function stores the Macro Instance ID of the created Macro Instance. The `macroTerminationResult` function is written,

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

Listing 22: PerformReceive

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

Listing 23: CheckIP

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

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

Listing 24: PerformTransitionReceive

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

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

Listing 25: ModalSplit

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

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

Listing 26: ModalJoin

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

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

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

Listing 27: macroTerminationResult

either with the boolean `true` or a string to indicate which outgoing transition of the MacroState should be selected, when the Macro Instance terminates.

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

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

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

The second phase evaluates the created Macro Instance.

When the Macro Instance has terminated its result is used to select the outgoing transition of the CallMacro state and the `callMacroChildInstance` func-

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

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

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

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

Listing 28: CallMacro

tion is reset for the next iteration. Otherwise the `MacroBehavior` rule is called for the created Macro Instance ID.

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

3.3.10 End Function

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

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

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

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

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

Listing 29: InitializeMacroArguments

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

      FinalizeInteraction()

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

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

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

Listing 30: PerformEnd

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

«««< HEAD =======
»»»> pr/16

CHAPTER 4

Implementation of Subject-Oriented Models

«««< HEAD =======

»»»> pr/16 Subject oriented models address the internal aspects and structures of a system. They are essentially models of the internal structure of a system and cover organizational and technical aspects. When implementing the models, it is now necessary to establish the relationship between the process model and the available resources. Figure 4.1 shows the individual steps from a process model to the executable process instance.

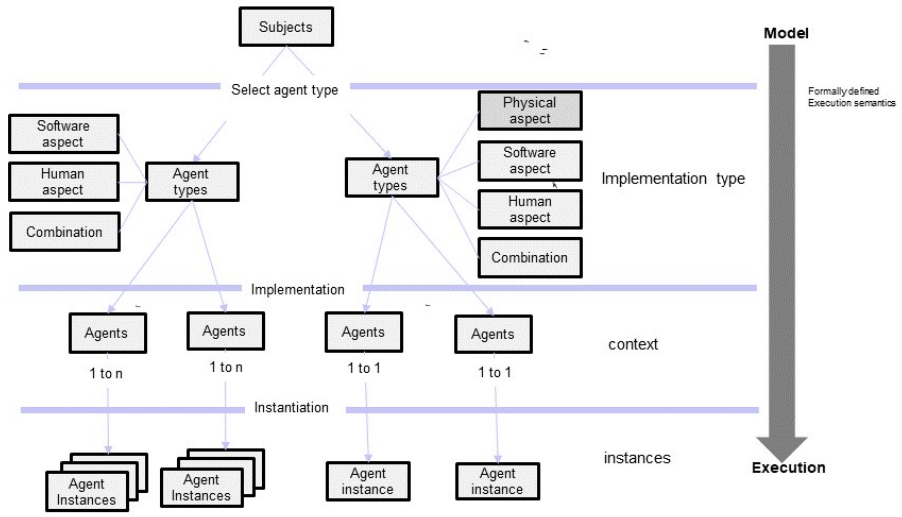


Figure 4.1: Implementation steps

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

bility and saves it. However, activities can also be carried out purely manually, for example when a warehouse worker receives a picking order on paper, executes it, marks it as executed on the order form and returns it to the warehouse manager.
When creating a system model, it is often not yet known which types of actors execute which actions. Therefore, it can be useful to abstract from said CS: said?? the? model when starting to describe processes by introducing abstract actors. A modeling language should allow the use of such abstractions. This means that when defining the process logic, no assertion should have to be made about what type of actor is realized. In S-BPM, the subjects represent abstract actors.
In the description of the control logic of a process, the individual activities are also described independently of their implementation. For example, for the action "create a picking order" it is not specified whether a human actor fills in a paper form or a screen mask, or whether a software system generates this form automatically. Thus, with activities the means by which something happens is not described, but rather only what happens.
The means are of course related to the implementation type of the actor. As soon as it has been defined which types of actors are assigned to the individual actions, the manner of realization of an activity has also been defined. In addition, the logical or physical object on which an action is executed also needs to be determined. Logical objects are data structures whose data is manipulated by activities. Paper forms represent a mixture between logical and physical objects, while a workpiece on which the "deburring" action takes place is a purely physical object. Therefore, there is a close relationship between the type of task holder, the actions and the associated objects actors manipulate or use when performing actions.
A system model can be used in different areas. The process logic is applied unchanged in the respective areas. However, it may be necessary to implement the individual actors and actions differently. Thus, in one environment certain actions could be performed by humans and in another the same actions could be performed by software systems. In the following, we refer to such different environments of use for a system model as context. Hence, for a process model, varying contexts can exist, in which there are different realization types for actors and actions.
In Subject Oriented Modeling, actors are not assigned to individual activities, but rather the actor type is assigned to an entire subject. This assignment is not part of the process logic, but in the most simple way it is done instead for each process in a separate two-column table. The left column contains the subject name and the right column the implementation type. If there are several contexts for a model, a separate assignment table is created for each of them. «««< HEAD The assignment of the implementation type forms the transition from the system logic and its implementation. Subsequently, it has to be defined which persons, software systems and physical systems represent the actors and how the individual actions are concretely realized. These aspects are described in detail in the following subsections. ======= The assignment of the implementation type forms the transition between the system logic and its implementation. Subsequently, it has to be defined which persons, software systems and physical systems represent the actors and how the individual actions are concretely realized. These aspects are described in detail in the following subsections. »»»> pr/16

4.1 People and organizations

In this section, we show how the organizational view of a company can be formally modeled and how elements of it can be referenced with a modeling language. Expressions of this language are stored in the subjects (the abstract actors) and resolved at runtime into concrete actors of the organization. These are then assigned the activities for processing. The presented approach is able to map any resources of a company (people, software, machines etc.) and any relationships between them. In this way, the organizational structure of a company can be mapped very precisely.

It should also be briefly pointed out that the original idea of the approach goes back to Schaller's work [Sch98]. Enhancements were made in the following articles [LSR14b, LRS14, LSR14d, LSR14c, LSR14a, LSR13, LRS11].

4.1.1 What is an organization?

Insights. Let's start with a real world scenario in the context of an insurance company. A claims department usually has a manager, a number of clerks and a lawyer. Generally the lawyer is the deputy of the department head, cf. figure 4.2.

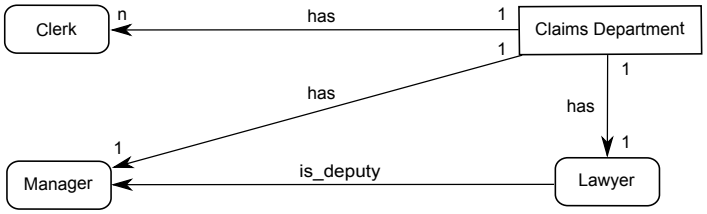


Figure 4.2: Claims department in general

We examined two concrete departments: one responsible for "Car Damages" the other responsible for "House Damages". Compared to the general structure and policies we observed some differences (cf. figure 4.3). At "Car Damages" there was an additional secretary position. In absence of the manager, organizational tasks were assigned to the secretary position. There was a change in the deputyship between the department head and the lawyer as well. Byron[1], the lawyer, had been working in the department for only three weeks and therefore was not very experienced. The clerk Winter has been working in the department for over ten years. Based on that constellation the department head Smith decided that Winter should be his general deputy. Hinton was as well a deputy for Smith but only depending on some constraint information like the cash value of a claim for instance (constrained deputy relation in figure 4.3).

Looking at this two departments we also found an interesting mutual deputyship between the lawyers of the two departments (cf. figure 4.3). This observation gets important when thinking about dividing the organization system into types or classes on the one hand and instances on the other. Please note, that the relationships defined until now are specified on different levels of abstraction (positions and actors).

[1] We are using fantasy names.

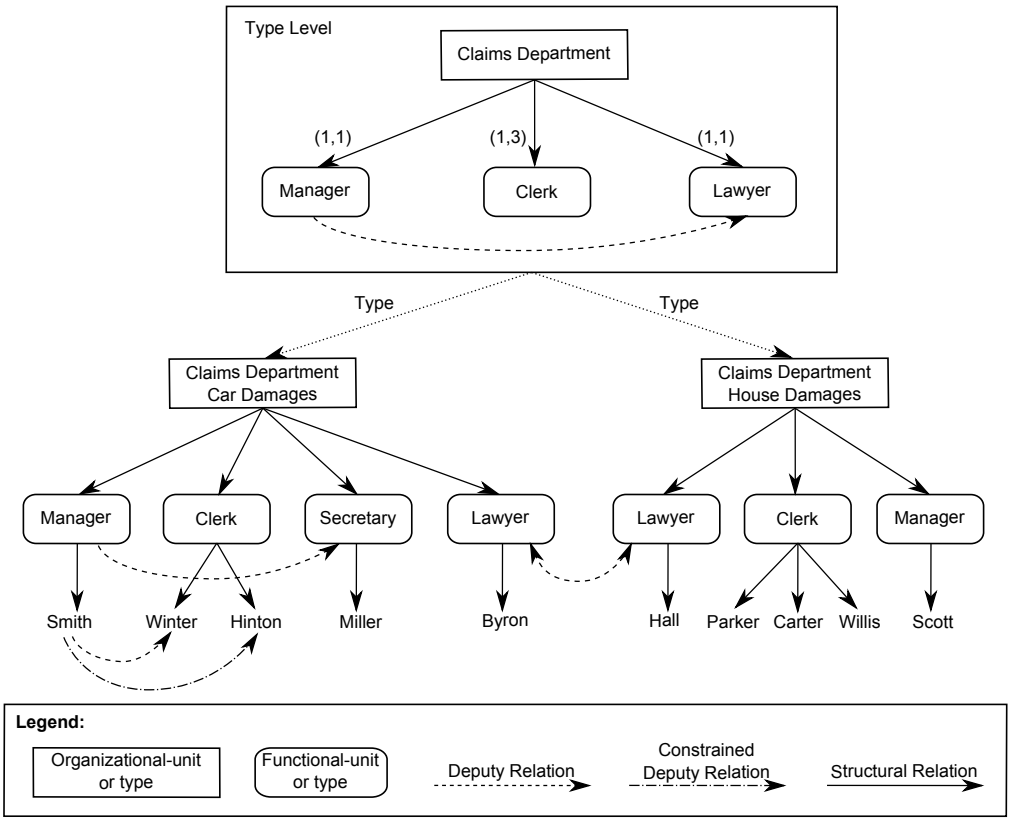


Figure 4.3: Type and instance level of the example, adapted from [LSR13, fig. 3]

Lessons Learned. This section describes additional observations concerning real world organization policies[2].

Knowledge Hierarchy. As we have seen there are different levels of organizational knowledge. On the top level general structural assertions like "a department consists of one to three clerks" are dominant. We call this level the *type* or *template* level. Knowledge on this level is based on experience and is changed seldom as time goes by. Looking at real world departments – we will call them *instances* – things become more concrete and specialized. There are concrete positions and the relationships between them. Finally, actors are assigned to the concrete positions. The organizational structures on this level are changing more frequently according to the demands of the daily business.

Relationships. An organization structure is formed by elements and relationships between them. It is important to realize the existence of several relationship types like "is_part_of", "is_deputy", "is_supervisor", "reports_to" and so on.

Positions are abstractions of persons (actors) having a defined skill set fulfilling specific tasks. These abstractions help defining a more stable model of the organization that is independent from employee turnover. Relationships can be defined between abstract positions or on the concrete actor level.

[2] A complete overview can be found in [Sch98].

Relationships are rarely of a general nature. As discussed in our example, relationships depend on specific constraint information like the cash value of a car claim. Even the "is_deputy"-relationship can depend on projects or products if you think in the terms of a matrix organization. They can also be only valid for a fixed time period.

Multidimensional organizations. Business organizations are multidimensional. Even in organizations that – at first glance – are structured hierarchically, there are structures belonging to the so called secondary ("shadow") organization comprising committees, commissions, boards and so on. The positions and functions of the secondary organization are assigned to the employees. This leads to a multidimensional organization in every case.

4.1.2 Linking the process and the organizational model

Based on the organizational model, a server component is proposed that implements a special architecture and a unique algorithm for the resolution of expressions denoting abstract actors in the sense of subjects. This server is part of a greater system, the IT landscape of the company. ERP, process automation or database management systems can be other components of the landscape. Each of these systems uses mechanisms to map actors to tasks of processes or to permissions on data objects. Instead of maintaining such assignments for each system individually by total enumeration, we propose using an organizational server. This organizational server contains the organizational model. A formal language is used to formulate expressions that define the assignments. Clients send these expressions as requests to the organizational server and retrieve sets of real actors as reply (cf. figure 4.4). The server offers a versatile interface consisting of only one function *dispatch* that returns a subset of the actors maintained within the organization model.

A major benefit of this approach is that the server forms the result set based on the current organizational model. This means that if actors change functions or relations, these changes have an immediate impact on the client systems. The language expressions remain unaltered. Before the organizational change, "Manager(*)" yields (according to fig. 4.3) the actors Smith and Scott. If Scott leaves the company and is replaced by Willis, the model is changed. Now, the same expression evaluates to Smith and the new manager Willis.

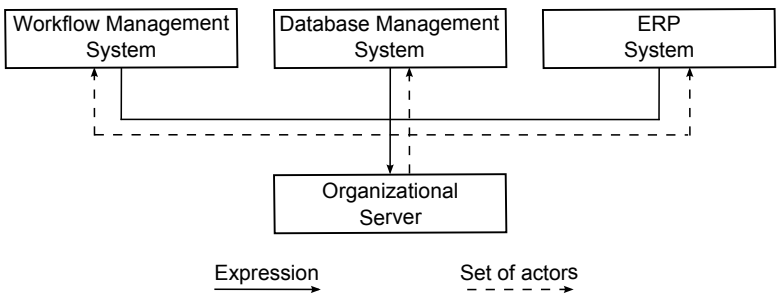


Figure 4.4: Architecture

An informal overview on the functionality of the proposed server based on the introduced scenario could be assumed as the following. In a process management system a subject is defined by the expression "Manager(Claims De-

partment Car Damages)". At runtime the expression is passed to the organizational server. By traversing the organizational graph in figure 4.3, the algorithm moves to the department "Claims Department Car Damages" looking for a position "Manager". After that, the algorithm determines all the actors assigned to that position, finding manager Smith. If Smith is on the job, his identification is handed back to the workflow management system and the search ends. In case that Smith is not available (e.g. through vacation or sickness) the algorithm searches for deputy relations between Smith and other actors. Obviously, there are two relations. If Winter **and** Hinton appear in the search result depends on the constraint on the relation to Hinton and whether they are on the job. In case of an empty set, the algorithm moves to the functional-unit manager looking for a deputy relation and finds the functional-unit secretary assigned to Miller. If Miller is on the job, her identification will be returned to the workflow management system. If not, the algorithm has the alternative of determining a valid deputy on the type level. Let us assume that the department is linked to the department type as depicted in figure 4.3. Within this type, the algorithm finds the lawyer as a deputy. It moves back to the instance "Claims Department Car Damages" and checks if there is a functional-unit with this name and an actor assigned to that functional-unit that is available. If Byron is on the job, his identification is returned. Otherwise, the lawyer of the "Claims Department Car Damages" has a two-way deputy relation with the lawyer of the "Claims Department House Damages". If this functional-unit has an actor assigned to itself and the actor is available, the algorithm will hand back his identification (here Hall, the lawyer of the "Claims Department House Damages"). Otherwise the returned set is empty. In this case, the workflow management system has to postpone the execution of the task.

4.2 Formal Specification

The formalization of the organizational model is described using relations and integrity constraints. In the following we present a simplified model of Schaller's approach.

Within our meta-model an organization is a tuple O = ($name$, $\mathcal{DOM}$, $\mathcal{ORG}$, $\mathfrak{R}$, $\mathcal{REL}$) where $name$ denotes the modeled organization. The remaining symbols have the following semantics:

4.2.1 Domains $\mathcal{DOM}$

$\mathcal{DOM} = \{\mathcal{BEZ}, T, ID, \mathcal{RN}, \mathcal{ATT}, \mathcal{W}, \mathcal{P}\}$ is a set of domains consisting of the subsets:

- $\mathcal{BEZ}$ an organization specific set of terms describing the building blocks of the organization, like "'claims department"', department "'head"' and so on,
- T denotes a set of time values, like "May 19th 2010 08:00:00".
- ID a set of abstract identifiers.
- $\mathcal{RN}$ denotes a set of relationship names, "'deputy"' or "'reports_to"' for instance.

- $\mathcal{ATT}$ a set of attributes used to detail the elements of our model. Attributes are mapped to model elements using the function $val : \mathcal{ATT} \rightarrow \mathcal{W}$ that assigns a value $w \in \mathcal{W}$ to each $a \in \mathcal{ATT}$.

- $\mathcal{P}$ denotes a set of predicates like "'(ActualYear - HiringYear) > 10"'.

4.2.2 Organization Elements $\mathcal{ORG}$

The set $\mathcal{ORG} = \mathcal{OE} \cup \mathcal{F} \cup OE \cup F \cup A$ comprises all the building blocks of an organization on the type as well as on the instance level. The elements of $\mathcal{ORG}$ represent the nodes of the resulting organization graph.

- $\mathcal{OE}$ denotes the set of organizational-unit types, like departments or working groups.

- $\mathcal{F}$ is the set of functional-unit types, like the "manager", "lawyers" and so on.

- OE represents the set of organizational-units, like departments, committees, teams and so on. As already explained, organizational-units can have a relation to a type. The total function $type_{OE} : OE \rightarrow \mathcal{OE} \cup NULL$ returns the specific type for every organizational-unit.

- F is the set of functional-units, like positions or roles. There also exists a type function that is almost defined in the same manner as described above. The type of a functional-unit is returned by the function $type_F F : F \rightarrow \mathcal{F} \cup NULL$.

  $\Gamma_s^E \subset OE \times (OE \cup F)$ denotes the is_part_of-relation between organizational- and functional-units.Γ_s^E on the one hand describes the mapping of functional-units to organizational-units. On the other hand the hierarchy between orga-nizational-units can be modeled. When focusing on the organizational-units the relation $\Gamma_s^{E'} = \Gamma_s^E \rhd OE$ has to be irreflexive and cycle-free. $\Gamma_s^{E''} = \Gamma_s^E \rhd F$ has to be surjective.

- R^F denotes a set of user-defined relations. All members $r \in R^F$ have the structure $r \subset (F \times F)$ and are irreflexive.

- The set A denotes the actors: employees (users) and the computer systems. We explicitly model these computer systems because they can carry out tasks and therefore need permissions. $\Gamma_s^{FA} \subset F \times A$ is a relation and describes the assignments of employees to positions. As seen before, there is also a user-defined set of relationships R^A. All relations $r \in R^A$ have the structure $r \subset (A \times (A \cup F))$. Further on, for all $r \in R^A$ the condition $\forall r \in R^A : [(x, y) \in r \rightarrow x \neq y]$ holds, meaning that every relation $r \in R^A$ is cycle-free.

- Additionally every element of $\mathcal{ORG}$ is described as tuple $(id, name)$, with $id \in ID$ and $name \in \mathcal{BEZ}$.

4.2.3 Set of Relations $\mathfrak{R}$

$\mathfrak{R}$ denotes a set of relation sets and is defined as $\mathfrak{R} = \mathfrak{R}^{\Upsilon} \cup \mathfrak{R}^{A}$, with:

- $\mathfrak{R}^{\Upsilon}$ denotes the set of relations defined between the types of organizational- and functional-units. $\mathfrak{R}^{\Upsilon}$ is defined as $\mathfrak{R}^{\Upsilon} = \Gamma_s^{\Upsilon} \cup \mathfrak{R}_b^{\Upsilon}$, with:
  - $\Gamma_s^{\Upsilon} \subset \mathcal{OE} \times (\mathcal{OE} \cup \mathcal{F})$ is the "is_part_of"-relation on the type level. Concerning the structure between the elements of $\mathcal{OE}$, there are some restrictions. Let's say $\Gamma_s^{\Upsilon'} = \Gamma_s^{\Upsilon} \rhd \mathcal{OE}$[3]. An organizational-unit type can not be his own successor. $\Gamma_s^{\Upsilon'}$ therefore has to be irreflexive and cycle-free. Let's have a look at the functional-unit types. Obviously the relationship between $\mathcal{OE}$ and $\mathcal{F}$ can be described as $\Gamma_s^{\Upsilon''} = \Gamma_s^{\Upsilon} \rhd \mathcal{F}$. Since organizational-unit types combine functional-unit types, $\Gamma_s^{\Upsilon''}$ has to be total. On the other side, every $f \in \mathcal{F}$ has to be linked to an organizational-unit type $o \in \mathcal{OE}$. $\Gamma_s^{\Upsilon''}$ therefore has to be surjective.
  - As explained above, there is the need for a flexible integration of new relation-types into the model. Therefore we define a set of relation-types $\mathfrak{R}_b^{\Upsilon}$. Every relationship $\Gamma_b^{\Upsilon} \in R^{\Upsilon}$ has the structure $\Gamma_b^{\Upsilon} \subset (\mathcal{F} \times \mathcal{F}) \times \mathcal{P}$ and can further be constraint using predicates to restrict the set of valid functional-unit types and therefore the set of valid users[4]. Please note that Γ_b^{Υ} is used as variable. The relations between the functional-unit types, that can expressed using the term $dom(\Gamma_b^{\Upsilon})$, are irreflexive. Defining a deputyship between one node and itself is not very meaningful for instance. Concerning the predicates we additionally postulate that each $\Gamma_b^{\Upsilon} \in \mathfrak{R}_b^{\Upsilon}$ has to be a function, assigning each ordered pair $(f, f) \in \mathcal{F} \times \mathcal{F}$ a unique predicate $p \in \mathcal{P}$.
- $\mathfrak{R}^{A}$ defines several relations between organizational- and functional-unit types as well as actors. We declare $\mathfrak{R}^{A}$ as $\mathfrak{R}^{A} = \mathfrak{R}^{E} \cup \mathfrak{R}^{FA}$, with:
  - $\mathfrak{R}^{E} = \Gamma_s^{E} \cup \mathfrak{R}_b^{E}$ the set of relations between organizational- and functional-units, with:
    - $\Gamma_s^{E} \subset OE \times (OE \cup F)$ denotes the "is_part_of"-relation between organiza-tional- and functional-units. On the one hand the relation describes the functional-units belonging to an organizational-unit, on the other hand the organization structure between the units themselves. Let $\Gamma_s^{E'} = \Gamma_s^{E} \rhd OE$ denote the structure between the organizational-units. According to our description $\Gamma_s^{E'}$ has to be irreflexive and cycle-free. In the same manner and similar to our definition of $\Gamma_s^{\Upsilon''}$,$\Gamma_s^{E''} = \Gamma_s^{E} \rhd F$ has to be total and surjective.
    - $\mathfrak{R}_b^{E}$ is a set of user-defined relations. Every single relation Γ_b^{E} within $\mathfrak{R}_b^{E}$ has the structure $\Gamma_b^{E} \subset F \times F$ and is irreflexive.
  - $\mathfrak{R}^{FA}$ denotes a set of relations between functional-units and actors. $\mathfrak{R}^{FA}$ is defined as $\mathfrak{R}^{FA} = \Gamma_s^{FA} \cup \mathfrak{R}_b^{FA}$, with:

[3] The operator $\rhd$ is defined as $((\Gamma \subset A \times B) \rhd (C \subset B)) := \{(x, y) \in \Gamma \mid y \in C\}$.

[4] Please take a look at the relation Γ_s^{FA} and its according constraints

- $\Gamma_s^{FA} \subset F \times A$ describes the function assignments of the actors. We demand that every actor is named to at least one function.
- $\mathfrak{R}_b^{FA}$ a set of user-defined, irreflexive relations Γ_b^{FA} having the structure $\forall \Gamma_b^{FA} \in \mathfrak{R}_b^{FA} : \Gamma_b^{FA} \subset A \times (A \cup F)$. For every $\Gamma_b^{FA} \in \mathfrak{R}_b^{FA}$ the condition $\left[(x,y) \in \Gamma_b^{FA} \rightarrow x \neq y\right]$ holds.

All elements of our model can be detailed using time constraints and attributes.

4.2.4 Additional relations $\mathcal{REL}$

$\mathcal{REL}$ consists of several relations and is defined as $\mathcal{REL} = \{\Gamma_{Time}, \Gamma_{\mathcal{ATT}}, \Gamma_{Card}, \Gamma_{val}, \Gamma_{Name}\}$, with:

- $\Gamma_{Time} \subset \left(\mathcal{ORG} \cup \mathfrak{R}\right) \times \left(T \times T\right)$ describes the duration of validity of every single organizational element in our model. Γ_{Time} therefore has to be a total relation.

  The two functions $start$ and $stop$ denote the birth and death of an organizational element. We define these functions as:

  $$\begin{aligned} start : \left(\mathcal{ORG} \cup \mathfrak{R}\right) \rightarrow T, &\quad \text{with the semantic} \\ &\quad start(x \in (\mathcal{ORG} \cup \mathfrak{R})) = dom(ran(x \lhd \Gamma_{Time})) \\ stop : \left(\mathcal{ORG} \cup \mathfrak{R}\right) \rightarrow T, &\quad \text{with the semantic} \\ &\quad stop(x \in (\mathcal{ORG} \cup \mathfrak{R})) = ran(ran(x \lhd \Gamma_{Time})) \end{aligned}$$

  It is obvious that the following constraint should hold: $\forall x \in \left(\mathcal{ORG} \cup \mathfrak{R}\right) : start(x) \leq stop(x)$. In order to define an existence ad infinitum we introduce the symbol "$*$" concerning the value of the $stop$ function.

- $\Gamma_{\mathcal{ATT}} \subset \left(\mathcal{ORG} \cup \mathfrak{R}\right) \times \mathcal{ATT}$ assigns attributes to our organizational elements. $\Gamma_{\mathcal{ATT}}$ is a surjective relation. Thus every attribute can only be assigned to one organizational element.

- $\Gamma_{Card} \subset \Gamma_s^{\Upsilon} \times (I\!N_o \times I\!N_o)$ assigns cardinalities to our "is_part_of"-relation between organizational- and functional-unit types. Γ_{Card} is a total and unique relation.

  As abbreviations we define the functions min and max, with

  $$\begin{aligned} min : \Gamma_s^{\Upsilon} \rightarrow I\!N_o, &\quad \text{with the semantic} \\ &\quad min(r \in \Gamma_s^{\Upsilon}) = dom(ran(r \lhd \Gamma_{Card})) \\ max : \Gamma_s^{\Upsilon} \rightarrow I\!N_o, &\quad \text{with the semantic} \\ &\quad max(r \in \Gamma_s^{\Upsilon}) = ran(ran(r \lhd \Gamma_{Card})) \end{aligned}$$

  Additionally we demand $\forall r \in \Gamma_s^{\Upsilon} : min(r) \leq max(r)$.

- Via $\Gamma_{val} \subset \mathcal{P} \times (F^{\Upsilon} \cup A)$ each predicate, its typed functional-unit and actors is assigned. The predicate *true* holds for all typed functions and actors and we define $\forall x \in F^{\Upsilon} \cup A : \quad (true, x) \in \Gamma_{val}$.

- $\Gamma_{Name} \subset \left(\mathfrak{R}_b^{\Upsilon} \cup \mathfrak{R}_b^{E} \cup \mathfrak{R}_b^{FA}\right) \times \mathcal{RN}$ assigns names to our user-defined relations. None of these relations should be nameless. Γ_{Name} therefore has to be total and unique. As already mentioned, $\Gamma_s^{\Upsilon}, \Gamma_s^{E}$ and Γ_s^{FA} denote "is_part_of"-relations. A specific naming of these relations is therefore unnecessary.

4.2.5 Policy Resolution

As discussed in section 4.1.2 our organizational server offers an interface with a very small footprint, consisting only of the function "'dispatch"'. The function returns a subset of the actors fulfilling the language expression in conjunction with conditions. These conditions are formulated via the following parameters expected by the function.

- a set of organizational-unit names $oe_{bez} \in \mathcal{BEZ}$,
- a set of tuples consisting of attributes and corresponding values $attr_{oe} \subset (\mathcal{BEZ} \times \mathcal{W})$ belonging to defined organizational-units,
- a set of functional-unit names $f_{bez} \in \mathcal{BEZ}$,
- a set of tuples consisting of attributes and corresponding values $attr_f \subset (\mathcal{BEZ} \times \mathcal{W})$ belonging to functional-units,
- a set of actor names $a_{bez} \in \mathcal{BEZ}$,
- a set of tuples consisting of attributes and related values $attr_a \subset (\mathcal{BEZ} \times \mathcal{W})$, belonging to actors,
- the name of a relation $rel \in \mathcal{RN}$ and
- a set of tuples consisting of attributes and corresponding values $attr_{rel} \subset (\mathcal{BEZ} \times \mathcal{W})$, belonging to that relation.

Algorithm 1 (dispatch)

```
(1)  funct dispatch(oe_bez ⊂ BEZ, attr_oe ⊂ (BEZ × W),
(2)    f_bez ⊂ BEZ, attr_f ⊂ (BEZ × W),
(3)    a_bez ⊂ BEZ, attr_a ⊂ (BEZ × W),
(4)    rel ∈ RN, attr_rel ⊂ (BEZ × W)) ⊂ A
(5)    begin
(6)      var ⟨oe ⊂ OE; f ⊂ F; a ⊂ A⟩;
(7)      / * First we have to determine the existing organizational elements * /
(8)      oe := (OE ▷ oe_bez);
(9)      f := (F ▷ f_bez);
(10)     a := (A ▷ a_bez);
(11)     if (oe ≠ {} ∨ attr_oe ≠ {})
(12)       then return   GetATbyOE(oe, attr_oe, f, attr_f, rel, attr_rel)
(13)     fi
(14)     if (f ≠ {} ∨ attr_f ≠ {})
(15)       then return   GetATbyF(f, attr_f, attr_a, rel, attr_rel)
(16)     fi
(17)     if (a ≠ {} ∨ attr_a ≠ {})
(18)       then return   GetAT(a, attr_a, rel, attr_rel)
(19)     fi
(20)     return{};
(21)   end
```

The execution of `dispatch({Claims Department Car Damages},{},{Clerk}, {},{},{damage sum < $100},{},{})` is equivalent to search all clerks in the organizational-unit *car damages* that are authorized to sign claims with a damage

sum lower than 100 dollars. The execution will lead to a call of the `GetATbyOE`-function. Based on the values of oe, f and a `GetATbyOE` determines the organization elements fulfilling the remaining conditions specified in $attr_{oe}$, $attr_f$ and so on.

Algorithm 2 (GetATbyOE)

```
(1)  funct GetATbyOE(oe ⊂ OE, attr_oe ⊂ (BEZ × W), f ⊂ F,
(2)    attr_f ⊂ (BEZ × W), rel ∈ RN, attr_rel ⊂ (BEZ × W),
(3)    attr_a ⊂ (BEZ × W)) ⊂ A
(4)    begin
(5)      var ⟨f' ⊂ F; oe_successor, oe'_successor ⊂ OE;
(6)      o_p ⊂ OE × OE⟩;
(7)      / * o_p denotes the vector corresponding to oe * /;
(8)      o_p := {(x, y)|x ∈ oe ∧ y ∈ OE};
(9)      oe_successor := dom( ( ( ( (Γ_s^E')^* )^T ) ∘ o_p );
(10)     if (oe_successor = {})
(11)       then oe_successor := OE;
(12)     fi
(13)     oe'_successor := GetOrgElements(oe_successor, attr_oe);
(14)     if (f = {})
(15)       then
(16)            / * determine all functional-units of the organizational-units * /
(17)            f' := ran({oe'_successor} ◁ Γ_s^E) ∩ F;
(18)            return   GetATbyF(f', attr_f, attr_a, rel, attr_rel);
(19)        else
(20)            / * which of the preselected functional-units belong to * /
(21)            / * the organizational-units determined in oe'_successor? * /
(22)            f' := ran({oe'_successor} ◁ Γ_s^E ▷ {f}) ∩ F;
(23)            return   GetATbyF(f', attr_f, attr_a, rel, attr_rel);
(24)     fi
(25)     return   {};
(26)   end
```

Line 8 describes the declaration of a relational vector. The calculation of all successors in line 9 is obtained by multiplying the transpose of the irreflexive closure of $\Gamma_s^{E'}$ with our vector o_p. After that we select the appropriate organizational-units.

In the case of the functional-unit set f passed to our function is empty, the algorithm selects all functional-units of the calculated transitive-reflexive closure and calls the function `GetATbyF` (line 14). If f != {} the relevant functional-units are selected in dependency on the specified organizational-units. After that the function `GetATbyF` is called (line 19).

Algorithm 3 (GetATbyF)

```
(1)  funct GetATbyF(f ⊂ F, attr_f ⊂ (BEZ × W), attr_a ⊂ (BEZ × W),
(2)    rel ∈ RN, attr_rel ⊂ (BEZ × W)) ⊂ A
(3)    begin
(4)      var ⟨f', f'' ⊂ F; a ⊂ A⟩;
(5)      if (attr_f = {})
(6)        then attr_f := ATT;
```

```
(7)     fi
(8)     f' := f;
(9)     if (f' = {})
(10)      then f' := F;
(11)    fi
(12)    f'' := GetOrgElements(f', attr_f);
(13)    a := ran(f'' ◁ Γ_s^FA);
(14)    return  GetAT(a, attr_a, rel, attr_rel);
(15)  end
```

Function `GetATbyF` determines the set of functional-units fulfilling the attributes passed over as parameters first. After that corresponding actors are selected and handed over to `GetAT`.

Algorithm 4 describes the core idea of our system. The passed parameters are being processed from the actor level up to level of organizational- and functional-unit types. Individual rules therefore have a higher priority than policies specified on the more abstract type level.

Algorithm 4 (GetAT)

```
(1)  funct GetAT(a ⊂ A, attr_a ⊂ (BEZ × W), rel ∈ RN, attr_rel ⊂ (BEZ × W)) ⊂ A
(2)    begin
(3)      if (attr_a = rel = attr_rel = {})
(4)        then return  a;
(5)      fi
(6)      / * exit for recursions * /
(7)      if (attr_a ≠ {} ∧ (rel = attr_rel = {}))
(8)        then
(9)              if a = {} then a := A fi
(10)             return  GetOrgElements(a, attr_a);
(11)     fi
(12)     if rel ≠ {}
(13)       then
(14)             if a = {} then a := A fi
(15)             var ⟨Γ_x, Γ_x' ∈ R_b^FA; y ⊂ (F ∪ A)⟩;
(16)             / * is there a user-defined relation fulfilling the attributes? * /
(17)             Γ_x' := dom(Γ_Name ▷ rel) ∩ R_b^FA;
(18)             Γ_x := GetOrgElements(Γ_x', attr_rel);
(19)             if Γ_x ≠ {}
(20)               then
(21)                     var ⟨z ⊂ A ∪ F; w, y ⊂ A⟩;
(22)                     z := ran(GetOrgElements({a} ◁ Γ_x, attr_rel));
(23)                     / * select the actors according to z
(24)                     w := z ∩ A;
(25)                     / * select the actors according to Γ_s^FA * /
(26)                     y := ran({{z} ∩ F} ◁ Γ_s^FA);
(27)                     return  w ∪ y;
(28)               else
(29)                     var ⟨Γ_x^F, Γ_x^F' ⊂ R_b^E⟩;
(30)                     Γ_x^F' := dom(Γ_Name ▷ rel) ∩ R_b^E;
(31)                     Γ_x^F := GetOrgElements(Γ_x^F', attr_rel);
(32)                     if Γ_x^F ≠ {}
(33)                       then
```

```
(34)                    var ⟨a' ⊂ A⟩;
(35)                    /* select all actors connected to the functional-unit */
(36)                    a' := ran({ran(Γ_x^F)} ◁ Γ_s^FA);
(37)                    return GetAT(a', attr_a, {}, {});
(38)                else /* lookup in the type level */
(39)                    var ⟨Γ_x^F, Γ_x^F' ⊂ ℜ_b^ϒ⟩;
(40)                    Γ_x^F' := dom(Γ_Name ▷ rel) ∩ ℜ_b^ϒ;
(41)                    Γ_x^F := GetOrgElements(Γ_x^F', attr_rel);
(42)                    if Γ_x^F ≠ {}
(43)                      then
(44)                          var ⟨f', f'' ⊂ F; a', a'', a''' ⊂ A⟩;
(45)                          /* 1. address all true relationships */
(46)                          f' := dom(type_F^ϒ ▷ ran(dom(Γ_x^F ▷ {wahr})));
(47)                          a' := ran(f' ◁ Γ_s^FA);
(48)                          /* 2. address false relationships */
(49)                          /* 2.1 address typed functional-units */
(50)                          f'' := dom((type_F^ϒ ▷ ran(dom(Γ_x^F ⩥ wahr}))))
(51)                          ∩
(52)                          ran(ran(Γ_x^F ⩥ {wahr}) ◁ Γ_val));
(53)                          a'' := f'' ◁ Γ_s^FA;
(54)                          /* 2.2 address direct relationships */
(55)                          a''' :=
(56)                          (dom(type_F^ϒ ▷ ran(dom(Γ_x^F ⩥ {wahr})) ◁ Γ_s^FA)
(57)                          ∩ran(ran(Γ_x^F ⩥ {wahr}) ◁ Γ_val));
(58)                          return GetAT(a' ∪ a'' ∪ a''', attr_a, {}, {});
(59)                      else
(60)                          /* no corresponding relationship found */
(62)                          return {};
(63)                    fi
(64)                fi
(65)            fi
(66)        fi
(67)    end
```

Line 3 describes the trivial case. If the only parameter is a set of actors the return value will be the same set.

In case that attributes are handed over (that have to be fulfilled by the respective actors) and no user-defined relation was defined (line 7), the algorithm determines all actors with a fulfilling attribute set. If a is empty, all actors (line 9) are used within the search (line 10).

The core concept of the algorithm starts with line 12. The following user-defined relations have to be resolved:

- The relation between actors and functional-units ($\mathfrak{R}_b^{FA}$),
- The relation between functional-units ($\mathfrak{R}_b^{E}$) and
- The relation between functional-unit types ($\mathfrak{R}_b^{\Upsilon}$)

If no corresponding relation on the actor level can be found (line 12), the algorithm checks for a connection between two functional-units ($\mathfrak{R}_b^{E}$) in order to retrieve the attached actors. If no match is possible, the algorithm searches for a

relation on the more abstract type level (line 40). If a relation can be found these policies are used for retrieving matching actors on the instance level. Lines 45 to 58 show the semantics of the predicates $p \in \mathcal{P}$ of the relation $\Gamma_b^{\Upsilon} \in \mathfrak{R}_b^{\Upsilon}$. All not predicate constrained relations are evaluated (line 45), first. After that the algorithm examines the constrained relations (line 48). The resulting actor set is used for a recursive call of algorithm 4.

Algorithm `GetOrgElements` selects those elements fulfilling a defined attribute set $attr$ from the set of organizational-units and relations.

Algorithm 5 (GetOrgElements)

```
(1)  funct GetOrgElements(k ⊂ ORG ∪ REL, attr ⊂ (BEZ × W)) ⊂ ORG ∪ REL
(2)    begin
(3)      var ⟨x ⊂ ORG ∪ REL⟩;
(4)      x := {};
(5)      foreach y ∈ k do
(6)        if CheckAttributes(y, attr)
(7)          then
(8)                x := x ∪ y;
(9)        fi
(10)     od
(11)     return x;
(12)   end
```

The following algorithm checks if an attribute of set $attr$ is mapped to an organizational element or relation $k \in \mathcal{ORG} \cup \mathcal{REL}$.

Algorithm 6 (CheckAttributes)

```
(1)  funct CheckAttributes(k ∈ ORG ∪ REL, attr ⊂ (BEZ × W)) ⊂ IB
(2)    begin
(3)      var ⟨attr_k ⊂ ATT⟩;
(4)      if (attr = {})
(5)        then return true;
(6)        else
(7)              attr_k := ran(k ◁ Γ_ATT);
(8)              if (dom(attr) ⊂ ran(attr_k))
(9)                then
(10)                     foreach y ∈ attr do
(11)                       if (ran(y) ≠ val(attr_k ▷ dom(y)))
(12)                         then return false;
(13)                       fi
(14)                     od
(15)                     return true;
(16)               else return false;
(17)             fi
(18)     fi
(19)   end
```

If the required attributes and attribute values are mapped to an organizational element or relation, function `CheckAttributes` returns $true \in \mathbb{B}$, or otherwise $false \in \mathbb{B}$. $\mathbb{B} = \{true, false\}$ denotes the set of boolean values.

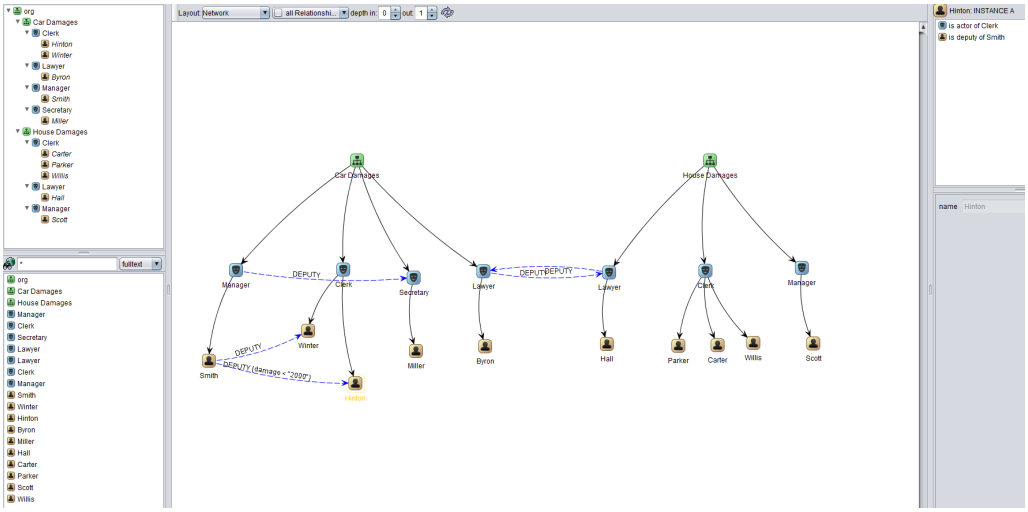


Figure 4.5: Screenshot: Implementation of the $\mathcal{C} - \mathcal{ORG}$ GUI

4.3 Implementation

The formalism discussed in this contribution is implemented in a prototype. Figure 4.5 depicts part of this implementation – the graphical user interface (GUI). It contains a *model* editor, a *search* area, a *tree-navigation* as well as an *attribute pane* and a *relation list* for a selected organizational element.

The *model editor* provides a graph-based view on the organizational structure. Organizational elements are represented as nodes and their relations as edges. It provides means to navigate the model by centering on selected nodes. As the central component of the user interface, it is discussed below in more detail.

The *search area* can be used to retrieve a list of organizational elements. It has two modes of operation:

1. It provides a simple text index search for attribute values, e.g. entering "Wi*" will yield Winter and Willis.
2. It can also be used to evaluate language expressions based on the approach described in sections **??** and 4.2.5. An expression is entered and the result set for the current state of the organizational model is shown.

The *tree-navigation* projects the concrete organizational structure on a tree. Consequently, entities are duplicated in the projection if they can be reached on different paths.

The *attribute pane* in the bottom right section shows the attributes of the currently selected node or relation. It allows a quick modification, e.g. the assignment of a predicate to a relation.

The *relation list* lists all relations of the currently selected node, independent from the relation-types hidden in the model editor. This allows access to

connected nodes and significantly reduces the time required to alter existing relations.

For quick access, elements can be dragged from any of the outer GUI sections and dropped into the model editor. If the elements have existing relations to the nodes already shown in the model editor, these relations will be shown as well. Otherwise, the elements are represented as unconnected nodes.

Figure 4.6 provides an enlarged view of the model editor[5]. Users perform most modifications of the organizational model via this component. In addition to navigating the model, they can create, modify and delete organizational elements and their interconnections.

It contains the model with the desired[6] relations. The editor also shows concrete constraints (predicates) on relations, e.g. the deputy relation with *damage* $<$ *"2000"* between *Smith* and *Hinton*.

In addition to the user interface, the implementation provides a service that can accept language expressions from client systems. This interface is based on the Representational State Transfer (REST) paradigm.

4.4 PHYSICAL INFRASTRUCTURE

4.5 IT-SYSTEMS AND SOFTWARE

[5]It shows the example model (cf. fig. 4.3). The type level is hidden.

[6]The relation-types to be shown can be selected.

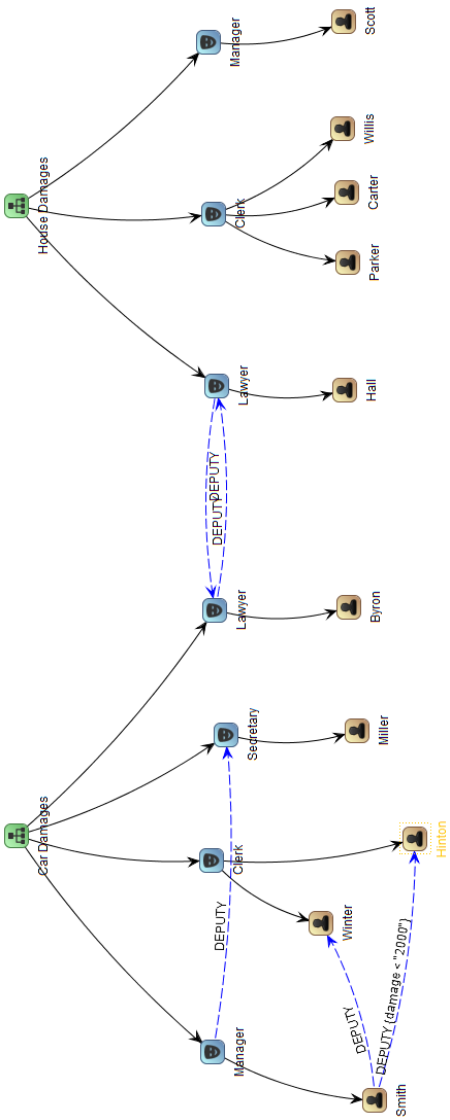


Figure 4.6: Model Region of the Implementation

CHAPTER 5

Aspects for further standardisation activities

CS: Hochkommas und - im pdf falsch gedruckt.

In this chapterr various aspects of the subject oriented modelling and programming concept are outlined. These aspects have already been published on different conferences. The following sections are based on these publications. CS: They contain original text parts and thus, the conclusions need to be aligned to the standardization effor, as tried for Fog Computing. The concepts described in theses sections will be part of future standardisation activities. The following sections are based on following publications:

- Subjects and Shared Input Pools:
  Hierarchies in Communication Oriented Business Process Models:
- Business Activity Monitoring for S-BPM: [**?**]
- Subject Oriented Project Management:
- Subject-oriented Fog Computing: [**?**]
- Activity based Costing [**?**]

5.1 Subjects and Shared Input Pools

Shared input pools have the same structure like subject-specific ones, and thus, the same properties like the standard input pool. The only difference is that different subjects can deposit in or remove messages from a shared input pool. Subjects that want to send a message via a shared input pool do not use a subject name as addressee of a message, but the name of a shared input pool instead. In a distributed system several shared input pools for different purposes can be used. Figure 7 shows the slightly changed structure of the traffic management system when operating it with a shared input pool CS: hier fehlt das originÃd're Beispiel als Bezugspunkt. The subject "Car detection" represents the shared input pool.

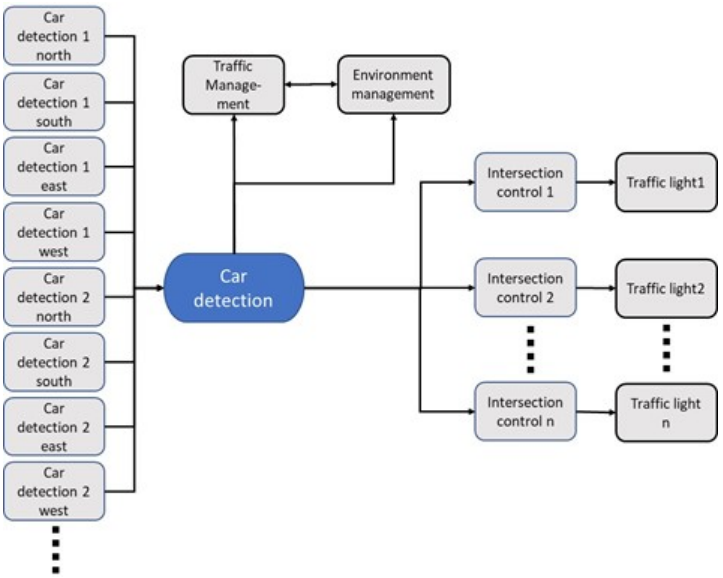


Figure 5.1: Traffic Management System with Shared Input Pool

Shared input pools make a distributed system more flexible when additional participants or nodes are added. For instance, a third intersection control could be added to the traffic management system without much effort. In this case, only the additional detectors and the components for controlling the intersection have to be complemented and linked to the shared input pool. The extension would have no impact on the behavior of the other subjects and their behavior in that system. There is one additional attribute for shared input pool: It defines whether a message will be removed from the input pool once a message has been picked up by a receiving subject. This mechanism is required, since several subject may need to process a particular message. In addition, it allows keeping historical information in the input pool, in particular for analyzing the content of an input pool independently of the behavior of interacting subjects. The messages of an input pool can be analyzed with respect to certain patterns of its messages. In order to perform such an analysis, Complex Event Processing (CEP) concepts can be applied. Complex Event Processing can be encapsulated in a subject. A subject of this kind scans the messages of a shared input pool and checks whether patterns of interest can be found. Once such a pattern is identified, a message including the discovered pattern can be sent to other participants, and initiate further activities. Figure 8 shows the traffic management example enriched with subjects processing complex events. In the example, the subject "CEP pollution analyzer" can analyze the time between cars passing the intersection in a certain time period. It can identify the events "low traffic" or "high traffic" and send it to the subject "Environment management". In case of tunnels, the subject "Environment management" might react to this information in a different way compared to open air settings.

5.1.1 Implementing Shared Input Pools

As mentioned earlier, shared data repositories represent a single point of failure of a distributed system. A malfunction of a shared data storage component or device may have a significant impact on the functionality of the whole distributed system. If a subject or a communication line is disturbed, only a small

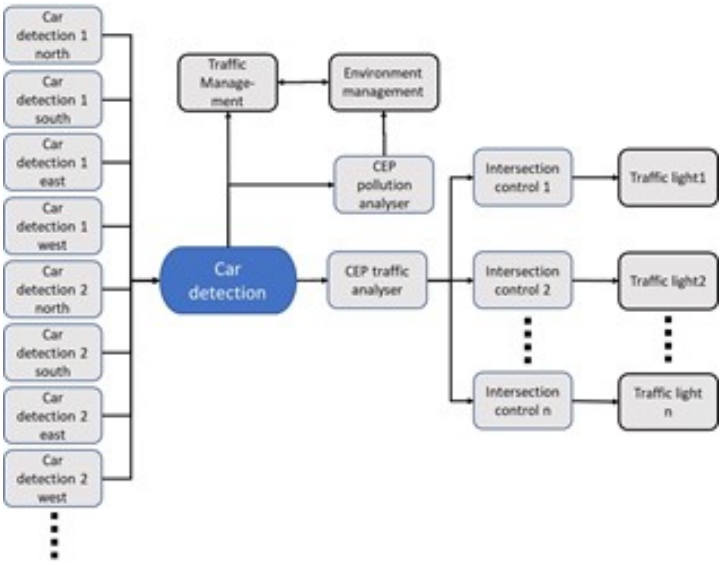


Figure 5.2: Shared Input Pools and Complex Event Processing

part of a system may be concerned but if a shared data store is down this has an impact on all subjects accessing this input pool.
In addition to this operational problem it must be decided in the course of organizational implementation which organization is held responsible for running and maintaining the system hosting the shared data. Such issues become prominent, if a distributed system is connecting several independent organizations, e.g., different companies in a supply chain. Distributed systems run by independent organizations may also have to deal with several changes dynamically, affecting the data quality and system stability. Even companies can be replaced by other organizations. If only functional subjects are concerned, such a change can be managed without affecting the operation of the entire system: The execution of a subject is just assigned to the new actor. The problem is more serious if a company leaving a distributed system is responsible for running the system with shared data, as other participants of the shared system are affected. Then, a new company still part of the distributed system must take over the responsibility for the shared data. The migration of these data from one company to another can become very cumbersome from the business point of view and from a technological perspective, too. One way to solve these problems is implementing shared input pools with blockchain technology. A blockchain is an open, distributed ledger that can record transactions efficiently in a verifiable and permanent way. Blockchains allow to achieve the integrity of a collection of data in a distributed peer-to-peer system, whereas the number of the peers is unknown and an unknown number of them are not reliable and trustworthy [**?**]. Today, blockchains are mainly used for managing the ownership of money, goods, real estates, etc. Each participant in a distributed system may have a copy of a blockchain. Changes in a blockchain follow a mechanism which manage changes in a consistent way and the change protocol guarantees that any participant will have again a consistent copy after a change. A change of a blockchain means that a new data record is added, and nothing can be removed from a block chain. Adding a new block to a block chain requires some effort from parties involved in a blockchain. This effort is rewarded by adding crypto money to

the party when having accomplished the task successfully. These rewards serve as an incentive for the creators of blocks.
Although heavily questioned with respect to effort and gains by practitioners [**?**] blockchain technology provides concepts ensuring the trustworthiness of system components. The latter becomes crucial when operating sensitive distributed systems, such as public transportation and healthcare, in particular when event-based data fusion is needed, where nodes of various type (sensor systems, vendor-specific monitoring systems. user devices, household items, etc.) exchange notifications of events and decision-relevant data with each other. In such settings, not only notification mechanisms needs to be streamlined in case of heterogeneity of nodes, but also data source trust is important for further processing and system behavior [**?**].
In order to ensure dependable sharing of data, these basic properties of blockchains need to be adapted to the requirements of a shared input pool. Hence, a blockchain-oriented implementation of a shared input pool must meet several requirements:

1. Subjects can subscribe for the access to a shared input pool.
2. Subjects subscribed for an input pool may deposit or read events from that input pool.
3. Events can be marked as removed from a shared input pool.
4. Subjects may analyze the content of a blockchain, e.g., when processing complex events.
5. There must be a mechanism that a block chain can be deleted, once all involved parties agree on that.

Traditionally data received from "things" are not very complex. These data are mainly values as measured by sensors, or binary signals. This may lead to a paradox situation: If such simple data are to be stored in a blockchain, the fee to be paid for adding blocks containing simple data is larger than the value being transferred. One way to solve the resulting incentive problem is to use permissioned block chains instead of open block chains: Blockchains for dedicated distributed application are not open blockchains like the ones implementing the management of digital currencies.
For the implementation of shared input pools, we suggest managed or permissioned blockchains. For instance, Hyperledger Fabric [**?**] is an open source implementation of a permissioned blockchain. Unlike to a public permissionless network, the participants are known to each other, rather than staying anonymous and interacting untrusted. It means, while the participants may not fully trust one another, e.g., in case of being competitors in the same industry sector, a network can be operated under a governance model that is built on the extent of trust existing between participants, such as a legal agreement or framework for handling disputes. When building a business process with known participants, such type of a blockchain implementation would be sufficient. Consensus algorithms for permissioned blockchains are faster and do need much less energy than permissionless blockchain networks.
In [**?**] it is reported that hyperledger fabric is the fastest available permissioned blockchain. The transaction throughput could even be increased from 3,000 to 20,000 transactions per second [**?**].

When using Hyperledger to create blockchain networks of that kind, a hyperledger blockchain network provides a technical infrastructure offering ledger and smart contract (chaincode) services to applications. Primarily, smart contracts are used to generate transactions which are subsequently distributed to each peer node in the network where they are immutably recorded on their copy of the ledger. The users of applications can be users of client applications or blockchain network administrators.
Subject add messages to the shared input pool and other subjects want to read these messages. If a shared input pool is implemented as a blockchain it is necessary that the chain code (smart contract in Ethereum) realizing the functions of the shared input pool must interact with the world outside the block chain. In hyper ledger fabric (including Ethereum), this problem is solved by so called oracles. We suggest using the blockchain patterns Oracle and Reverse Oracle as described in [**?**]. For flexibility reasons we prefer off chain oracles - see figure 5.3.

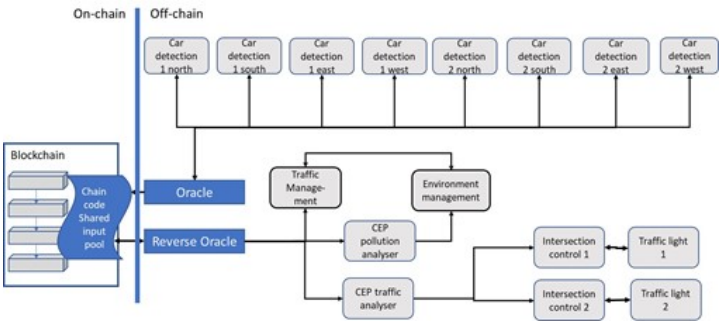


Figure 5.3: Utilizing block chain patterns Oracle and Reverse Oracle

5.1.2 Conclusion

The more the Internet of Things (IoT) propagates into domain-specific applications, the more stakeholders get involved with respect to business and user requirements. They expect omnipresent use and adaptation on demand. Ensuring robust and semantically correct operation in dynamically networked IoT environments requires tools and development methods to handle complex patterns of interactions due to the different components and capabilities of actors.
These patterns refer to the (reactive) flow of control and correct exchange of data. We have proposed an integrated approach based on subject-oriented process models. These role-specific representations allow behavior abstractions on various levels of granularity and can be enriched with a mechanism for handling complex events and sharing data. The data handling mechanism is bound to exchanging messages and a blackboard-like structure. Its behavior can be implemented through blockchain technologies, in case a single point of failure in system operation should be inhibited. The latter is of crucial importance, once the data exchange between IoT-system elements should be trustworthy and traceable.
The presented approach should facilitate transparent development and stakeholder understanding of (complex) IoT systems in dynamic settings, due to the implementation-independent representation on a mainly diagrammatic level based on a minimalistic notation, stemming from subject-oriented modeling. Abstractions and decomposition into IoT system components encapsulate behavior. The

overall behavior of an IoT system is determined by a set of interactions that integrates the control flow with data exchange patterns from a semantic process perspective. Application design can be understood as top-down approach with the functionality specific to the IoT application residing on an edge operating system. Platform services implement all functional requirements, and are backed by communication and information processing technologies. Cross-functional issues, such as secure operation, business-relevant standardization, and critical event handling can be explicated on an implementation-independent level due to the semantic process representation scheme. The resulting models are executable and thus, can be adapted dynamically.

5.1.3 Future Work

Due to the novel conceptual integration addressed, several aspects and topics need to be addressed by future research: spacing

- From an application perspective, the results need to be aligned with novel industry 4.0 concepts (cf. [29]), since there not only existing standards are framed by business processes, but also distributed operation of production-relevant processes and real-time sharing of data.
- From an implementation perspective, our approach requires a (prototypical) realization of an appropriate block chain mechanism for managing shared input pools meeting all requirements in section 4.
- From an industry perspective, performance evaluations might lead to reconsider our conceptual findings, e.g., how to manage a shared input pool of a distributed system in real time.
- Definition of structural semantics in OWL
- Definition of execution semantics in ASM

5.2 Hierarchies in Communication Oriented Business Process Models

PASS offers powerful possibilities for structuring complex process systems. The ways to do that are demonstrated with an example. As an example we will consider a process for realizing a car break down service. This service consists of several connected processes. There is the main process for handling the car accident and supporting e.g. processes for organising towing and repair shop services. Insurance companies may be involved for covering damages, the customer gets an invoice, uses money transfer services or banks for paying the invoice. These processes are executed by various organisations like help desk service companies, towing service companies, car repair workshops banks etc.. In most business process projects not all parts of processes are described in detail. Only a certain part is considered, e.g. only the help desk process has to be considered in detail. In order to do so we have to consider the whole environment in which a considered process is embedded. We have to know which relations exists to these other processes. It is necessary to know which inputs are rquired by neighbour processes and which results they deliver. A help desk process which organizes the towing services has to know how the towing service is requested and which further interactions are required. For instance it must be

agreed whether the towing service informs the client about the arrival time of the towing truck or the help desk does it.

5.2.1 Process Architecture

Rectangles represent processes. Each process has a name. Processes consists of other processes and/or subjects. The lines between the rectangles represent the communication channels between processes. Each communication channel has a nameand can contain other communication chan-nels and/or messages.

Figure 5.4 shows the highest process level of the car break down service. In the "car use" process the event "car break down" happens. In order to organize support an interaction is initiated with process "car break down service" . Between these processes messages are exchanged which are elements of the communication channel "Car break down handling".

Figure 5.4: High level structure of car break down service

Figure 5.5 shows the next process structure level of the process "car break down ser-vice". In this level the process "Car break down service" is spltid in 10 processes. The processes "Bank", "Insurance service", "Car repair workshop", "Incident Management","Mobility Manage-ment" and "Towing Management" have a communication channel to the prcess "Car usage". This means the communication channel "Car break down handling" is split into five communication channels. Each of them covers the communication with the relared process, e.g. the communi-cation channel "Accident notification Car break down" is the communication channel between the processes "Car usage" and "Incident Management".

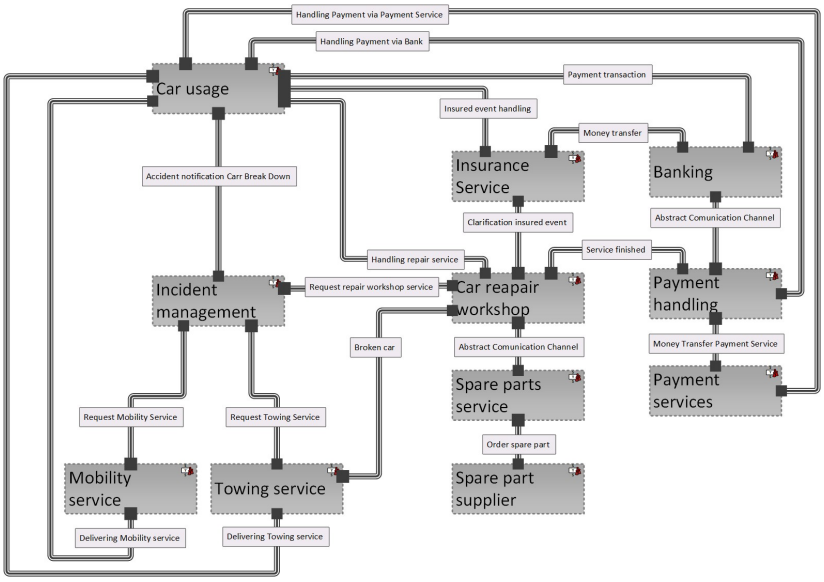


Figure 5.5: Structure of the Emmergency Call Handling Process

Inside a process there can be also processes. This means that levels of processes can be built. Figure 5.6 shows the next deeper level of our process hierarchy. The process "Car repair workshop" is structured in six processes. According to this separation the communication sets are also splitted e.g. the communication set "Handling repair service" is splitted into three parts, one part is han-dled by the process "Service scheduling" the other by the process "Car droping" and the third one by the process "Customer Satisfaction".

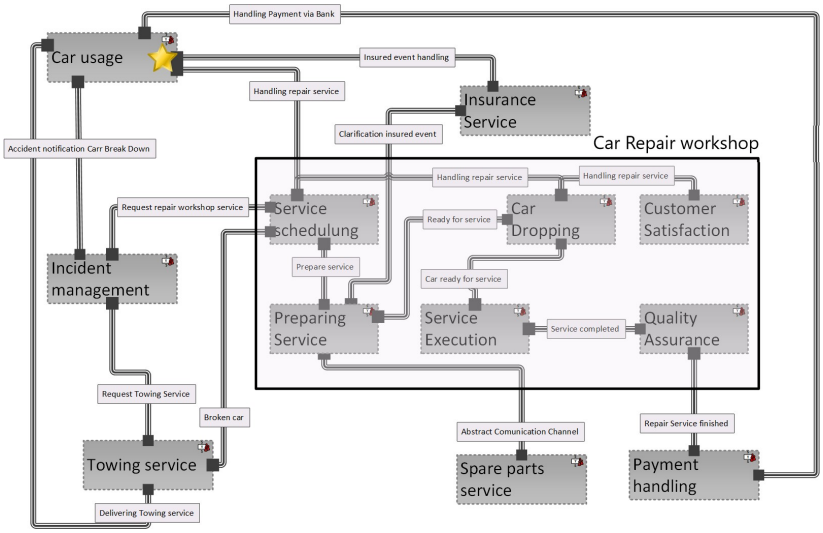


Figure 5.6: Details of the "Car repair workshop" Process

As already mentioned, processes cannot communicate directly with each other. The active entities of a process, the subjects communicate with each other. This means messages from one process are sent to an other process are reveived by a subject inside of that process. Messages belonging to a channel are assigned to a sending or receiving subject at the lowest level of a process architecture. This lowest level of a process description is the subject interaction diagram (SID) which shows the involved subjects of a process and the messages they exchange. In the following we consider the process incident management in more detail. This process does not contain other processes like the process "Car Repair Shop". The process "Incident management" contains a Subject Interaction Diagram. Some of the subjects of a process communicate with subjects in other processes. These subjects are called border subjects because they are at the border of a process to other prcesses. Figure 5.7 shows the process "Incident management" with its border subjects. There is a border subject "Help agent" which communicates with the processes "Towing service", "Mobility ser-vice"and "Car repair workshop", precisely it communicates with a subject in one of these processes. Another border subject of the process "Incident management" which is called "Help desk"communicates with the process "Car usage".

The border subjects of the process "Incident management" must have a coresponding border sub-ject at the neighbour processes. The border subjects "Call agent" communicates with the border subject "Help requestor" of process "Car usage" and the border subject "Help agent" communi-cates with border subjects of the processes "Car repair workshop", "Towing service and "Mobility service".

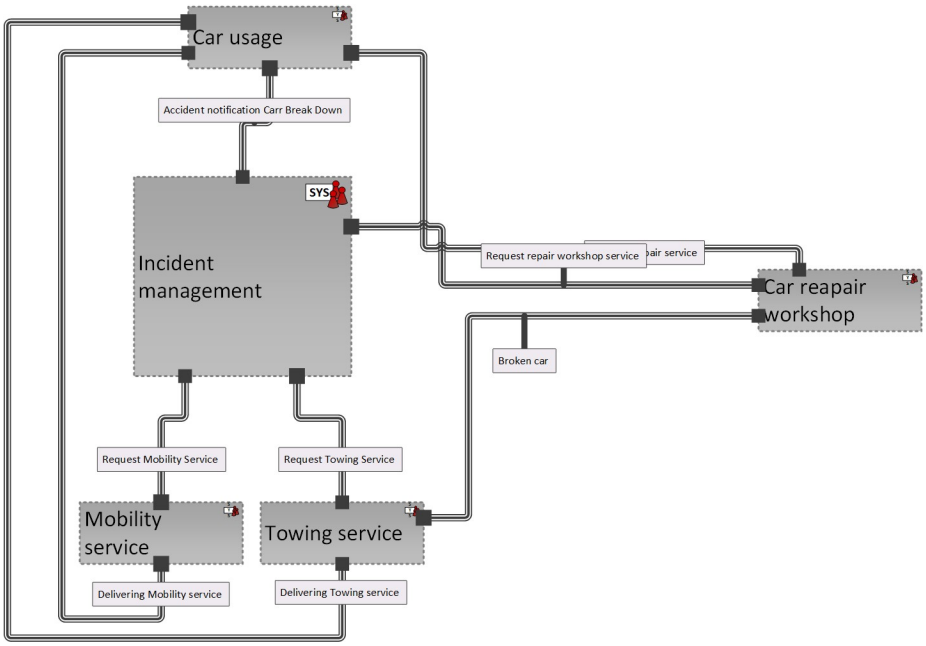


Figure 5.7: Neighbors of the "Incident Manaement Process"

The process "Incident management" with all the border subjects is shown in figure 5.8.

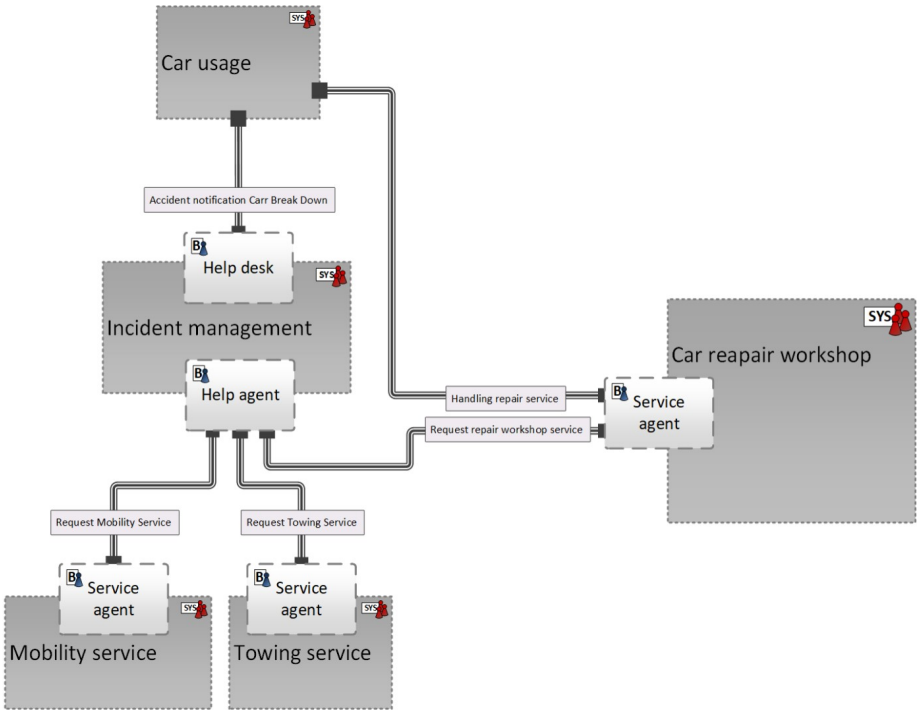


Figure 5.8: Border subjects of the "Incident Management" Process

The border subjects of the processes "Mobility service", "Towing service" and "Car repair work-shop" have the same name âĂIJService agentâĂİ but these are different subjets because they belong to different processes. Because the process

"Car repair workshop" consists of several layers the corre-sponding border subject can be in a process which is part of process "Car repair workshop" in a lower level.
From the perspective of the subjects inside of the process "Incident managent" are the border subjects of the processes "Mobility service", "Towing service" and "Car repair workshop" interfaces to these processes, therefore they are called interface subjects in the Subject Interaction Diegram of a process. Figure 5.9shows the Subject Interaction Diagram of the process incident management.

5.2.2 Behavioral Interface

Processes to which a considered process has communication relationships are called process neighbours or for short neighbours. Now we want to consider the details of the communication relationships between two neighbours. The interface between two processes is defined by the related border subjects and the allowed sequences in which the messages are exchanged between them in a communication channel. As already described above each message is defined by a name and the data which are transported the so-called payload. A border subject observes the behavior of the border subject of the neighbour process and vice versa. Figure 5.10 shows the border subject "Help desk" of the processes "Incident Management" which communicates with the border subject of process "Car usage".

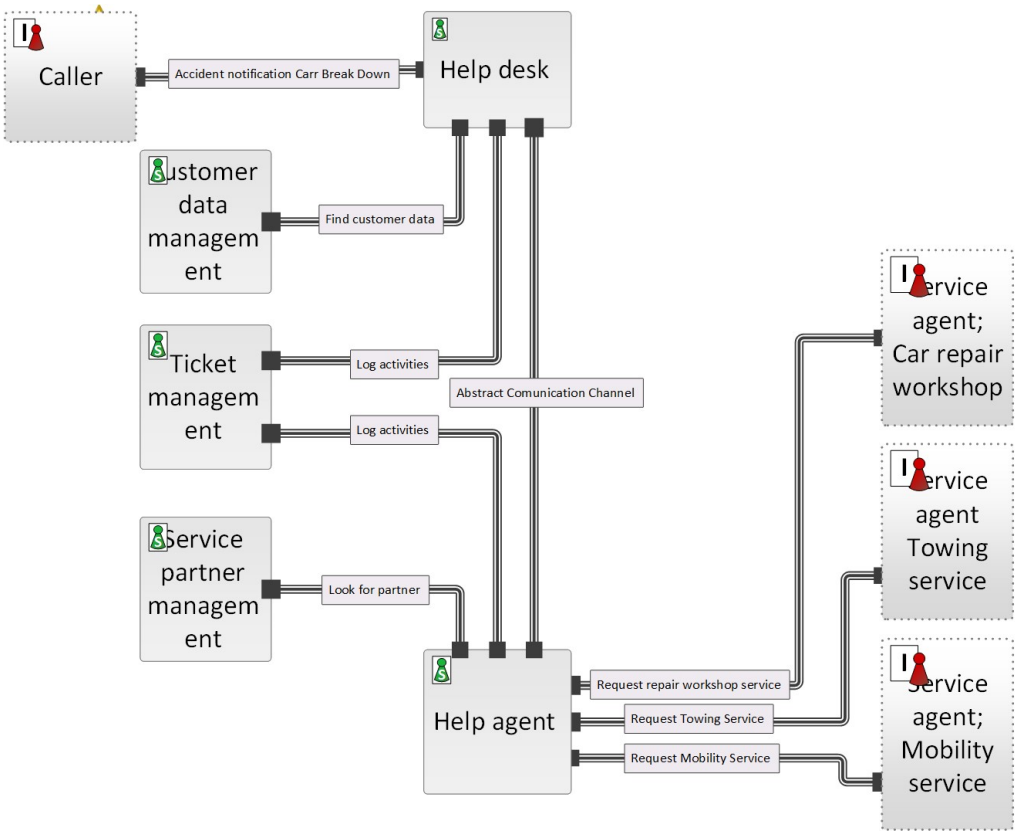


Figure 5.9: Subject Interaction Diagram of the Process "Incident Management"

Because we consider the process "Incident management" the border subject

"Caller" of the process "Car usage" becomes an interface subject in the SID (details about interface subjects can be found in [?]) of the process âĂIJIncident ManagementâĂİ. Figure 5.10 shows the detailed Subject Interaction Diagram around the subject help desk.

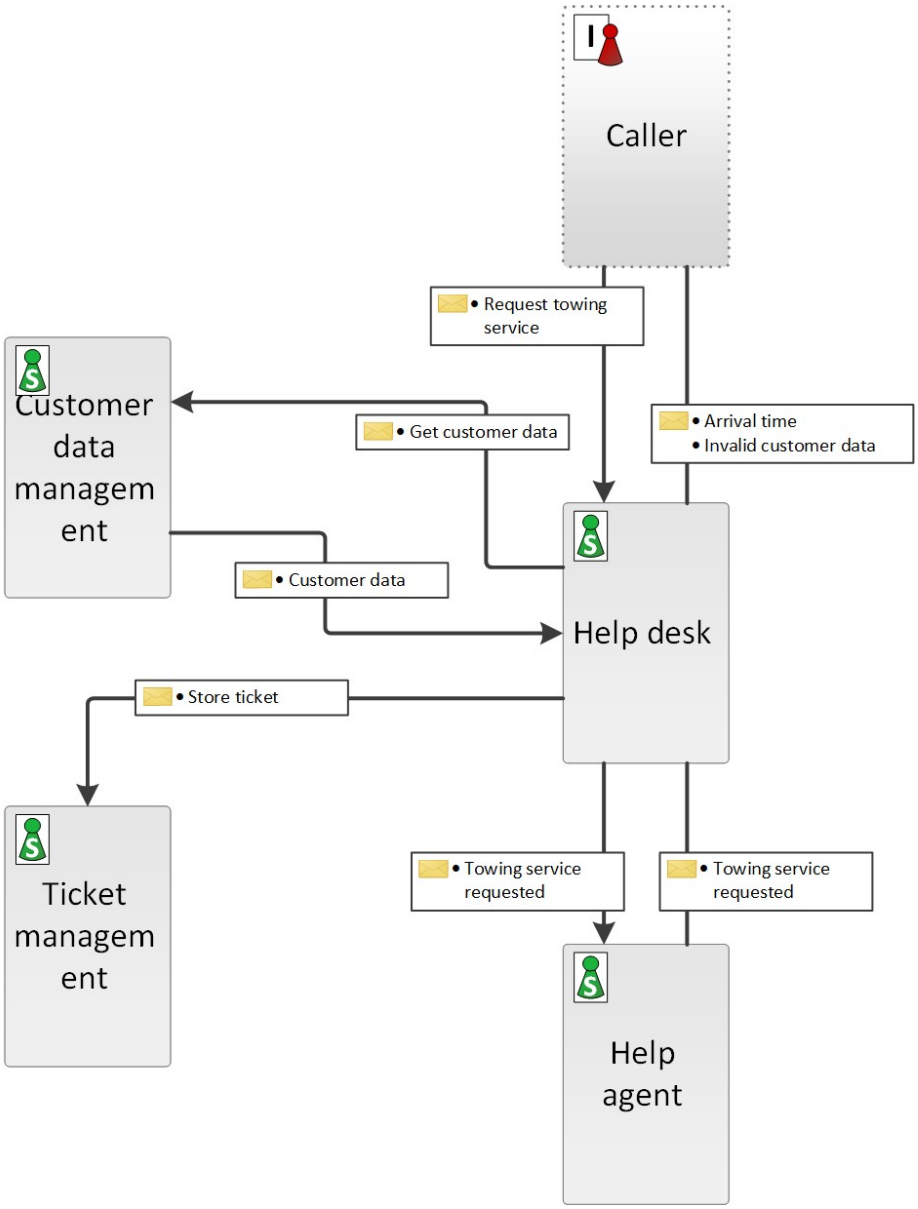


Figure 5.10: Subject Interaction around the subject âĂIJHelp deskâĂİ

Instead of the channels the messages required for a towing service request are shown. A message "Request towing service" comes from the interface subject. This message is accepted by the subject "help desk". The subject help desk checks the customer data received with this message by sending a corresponding the message "Get customer data" to the subject "Customer data management". This subject send the complete customer data back to the subject "Help desk" via the message "Customer data". The subject "Help desk" checks the customer data. If the data are invalid a message "Invalid customer data" is sent to

the subject "Caller" and the process is finished. If the customer data are valid with that data the subject "Help desk" creates a trouble ticket which is sent to the subject "Ticket management". After that the message "Towing service requested" is sent to the help agent which organizes the towing service. The part of the communication structure of the subject "Help agent" in order to organize the towing service is not shown in figure 5.11. We only see that subject "Help agent" sends the message "Towing service data" to the subject "Help desk". This message contains all the data about the service e.g. name of the towing company and arrival time. The subject "Help desk" forwards that data to the interface subject "Caller". This behavior is shown in figure 5.10.

The behavior described in the figure above contains the communication with all neighbor subjects of subject "Help desk" including the communication with the interface subject "Caller". From the perspective of this subject the communication of the subject "Help desk" with its other neighbor subjects is not relevant. For the subject "Caller" only the commumication sequence between itself and the subject "Help desk" is relevant. These allowed communication sequences are called the behavioral interface.
The behavioral interface between two subjects can be derived from the complete behavior of a subject by deleting the interactions with all the other subjects . Figure 5.12 shows how the communication sequence relevant for the communication be-tween the subject "Help Desk" and "Caller" is derived from the complete behavior of subject "Help desk".

A behavoral interface is always relative to a communication partner. In figure 5.12 the behavioral interface is relative to the interface subject "Caller". The behavioral interface to the subject "Ticket Management" is different because only the communication activities with this subject are considered.This behavioral interface would be very simple. It consists of only one send activity, sending the message "Store ticket".
The behavioral interface relative to a partner subject can be automatically derived from the complete behavior of a subject (see [**?**]).

5.2.3 Future Work

Due to the novel conceptual integration addressed, several aspects and topics need to be addressed by future research: spacing

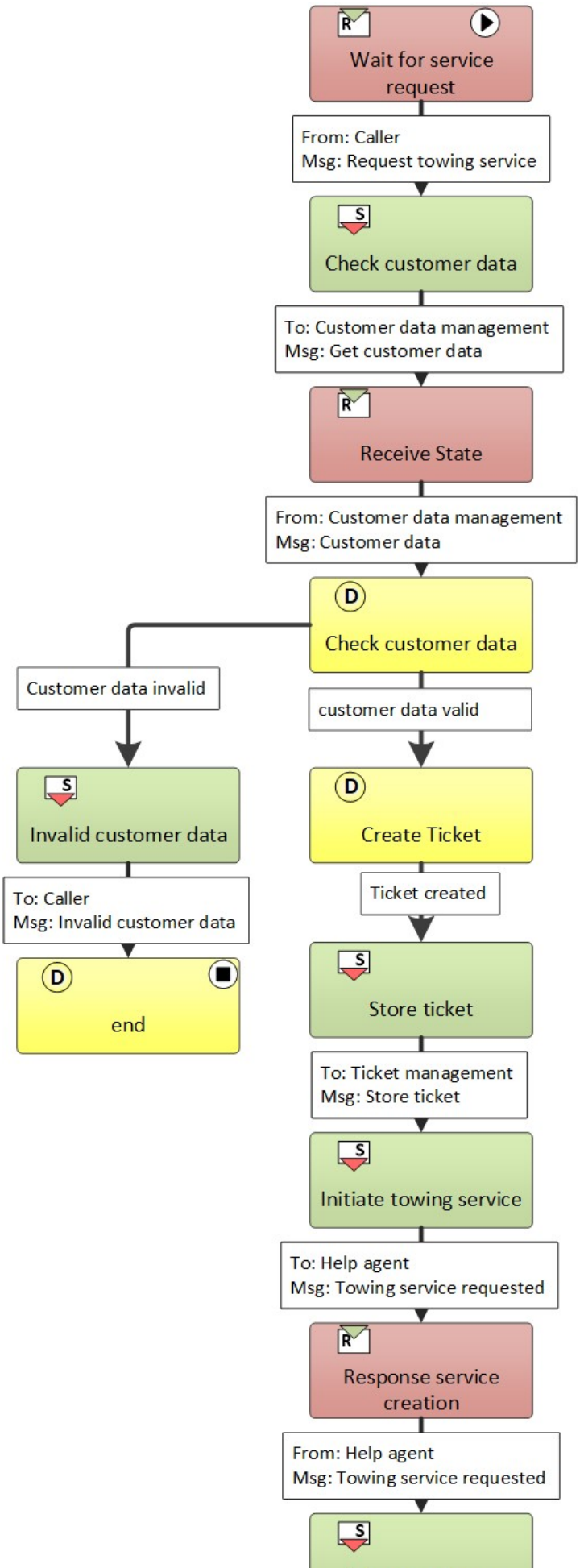
- Clarify terminology e.g. using the term interface subject, system interface, implementation

- Definition of structural semantics in OWL

- Definition of execution semantics in ASM. The semantic of the behaiour interface and its relation to the behavior of the related subject has to be described.

5.3 Business Activity Monitoring for S-BPM

Monitoring of Business Process looks at running instances. For those it measures metrics, aggregates them to Process Performance Indicators (PPIs) as a business process-related form of Key Performance Indicators (KPIs), reveals deviations


Wait for service request
From: Caller
Msg: Request towing service
Check customer data
To: Customer data management
Msg: Get customer data
Receive State
From: Customer data management
Msg: Customer data
Check customer data
Customer data invalid
customer data valid
Invalid customer data
Create Ticket
To: Caller
Msg: Invalid customer data
Ticket created
end
Store ticket
To: Ticket management
Msg: Store ticket
Initiate towing service
To: Help agent
Msg: Towing service requested
Response service creation
From: Help agent
Msg: Towing service requested

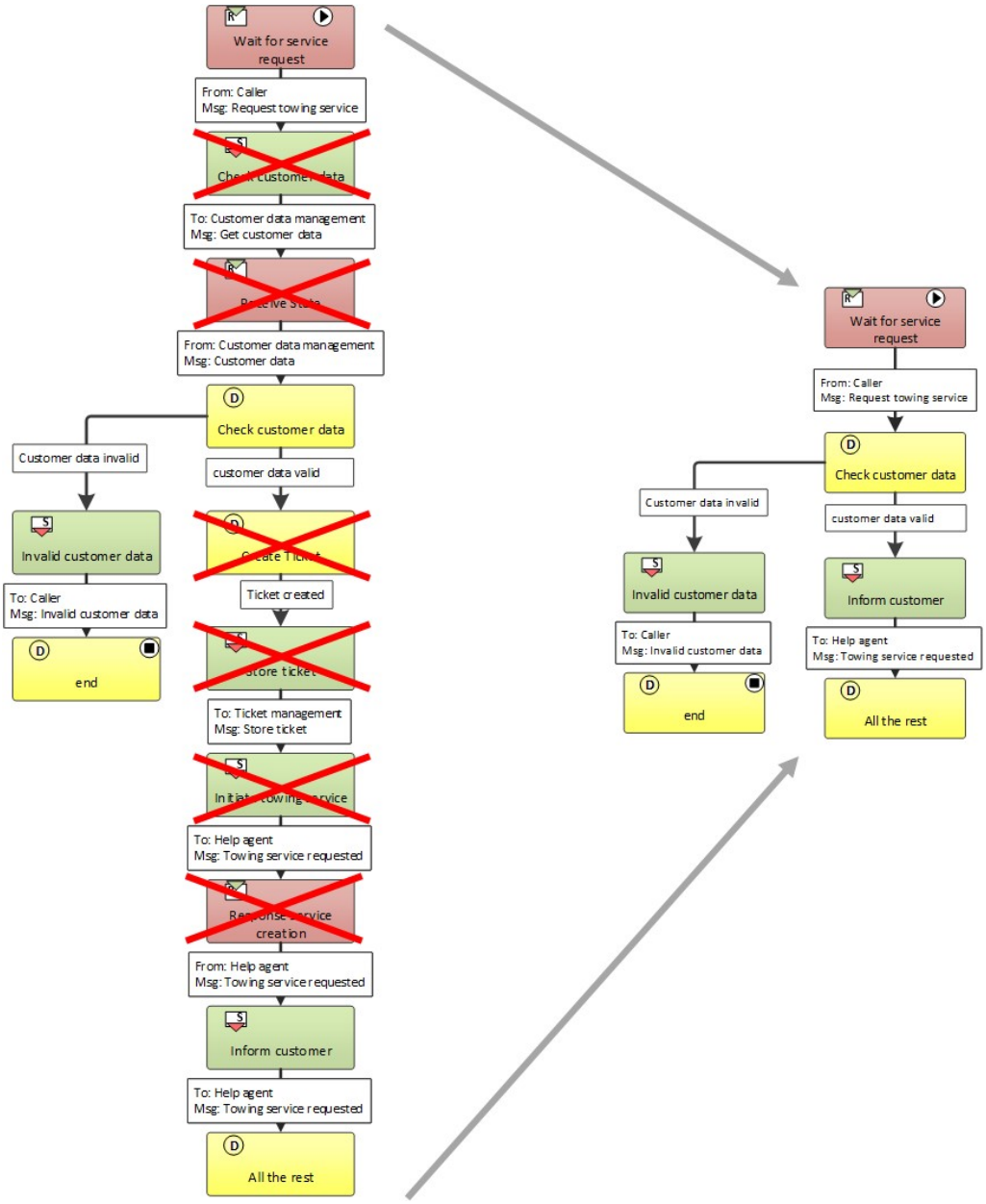


Figure 5.12: Deriving the Behavioral Interface from the Subject Behavior

(as-is vs. to-be) and report and presents results to people in charge or interested in the value of the PPI. Thus monitoring lays ground for the performance analysis in the key dimensions quality, time and costs of processes and helps identifying weaknesses and opportunities for improvement [**?**]. By feeding back information for completed and running instances to analysis monitoring fosters organizational learning, forms an important part of the Business Process Management (BPM) lifecycle [**?**] and thus helps implementing the operational level in the closed-loop approach to enterprise performance management [**?**] (see figure 5.13).

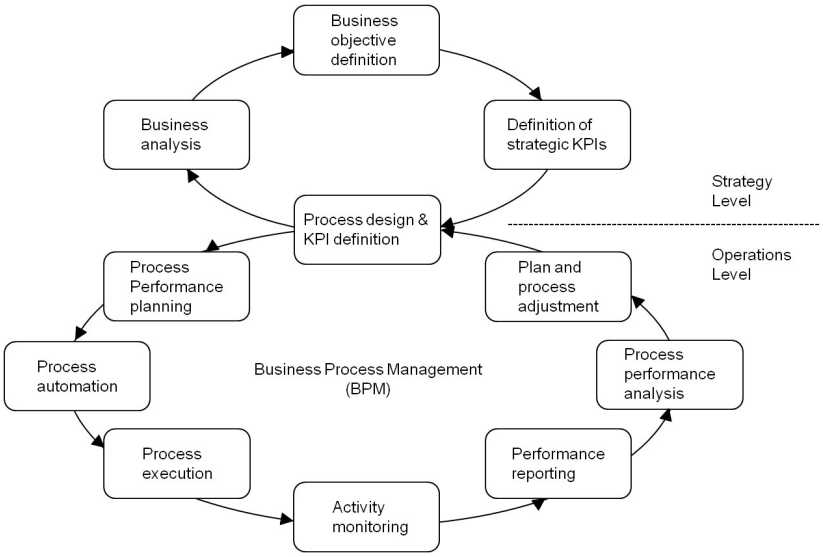


Figure 5.13: Closed-loop Approach to Performance Management [**?**]

5.3.1 Architecture

A Business Activity Monitoring (BAM) environment supported by Complex Event Processing consists of several elements necessary at build time and at runtime (see figure 5.14) and [**?**], [**?**] , [**?**]). At build time a modeling environment should provide tools for designing processes (e.g. Metasonic Build) and defining process performance indicators (PPIs), BAM events, rules, thresholds etc. as well as parameters for their visualization in report and on dashboards. At runtime there are (1) event producers like a process engine (e.g. Metasonic Flow) or an ERP system (e.g. SAP) which feed events into an event cloud or stream (chronologically ordered). (2) Event Processing Agents (EPA) form the event processing logic. They process events based on metrics, event patterns, rules and other parameters specified at design time. Their basic logical functions include filtering and transforming events and detecting patterns among them. Global state elements allow them accessing data from outside the application (e.g. from an ERP system). EPAs put the results of their processing (also to be understood as events) out to Event Consumers (3) like dashboards or process engines. Input and Output Adapters (IA, OA) transform event data between different formats of system elements as necessary. All system elements involved form an Event Processing Network (EPN), in which events are exchanged by communication mechanisms.

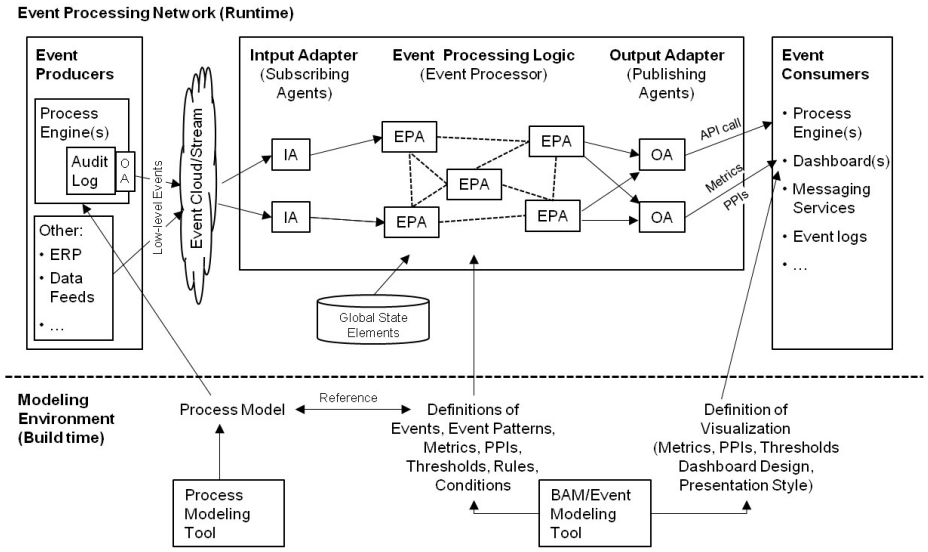


Figure 5.14: Integrated BAM/CEP Architecture [?]

5.3.2 Modeling BAM Parameters at Build Time

As mentioned in the last section it is necessary not only to model the processes, but also numerous pieces of information relevant for a sound process monitoring in the sense of Business Activity Monitoring (BAM model). These can be derived from answers to questions like what, when, how and how often should be measured by whom [?]. The information should also include how single metrics are to be aggregated in order to determine Process Performance Indicators (PPIs). For systematically collecting and documenting the necessary information fact sheets or templates for metrics and performance indicators have been developed [?], [?]. Figure 5.1 shows an extract of a sample fact sheet defined for the average processing time of activities (see also [?], [?]).

Replacing the content column by more formal ontology-based linguistic patterns as suggested by Del-Rio-Ortega et al. (see table 5.2) could help relating PPIs to elements of the process model, performing automated analysis [?] and implementing the measurement at runtime.

Friedenstab et al. argue that such linguistic patterns do not fit to the usually graphical modeling of processes which makes integration difficult [?]. The authors discuss some more approaches to BAM modeling. With regard to the limitations revealed, they present a BAM-related extension of the graphical Business Process Model Notation (BPMN) [?]. Using an abstract language syntax based on the Unified Modeling Language (UML) they started defining meta models for language constructs needed for BAM as there are Duration, Frequency, Composed Basic Measure, Aggregated Measure, Filter, Target Definition, Actions, Measure-based Expressions and Dashboard. Figure 5.15 depicts the example for the duration of elements on different levels of detail, where the grey colored parts indicate references to the BPMN specification.

In a second step Friedenstab et al. developed a concrete syntax allowing for modeling the abstract language elements with graphical symbols and text labels. Parts of it are visible in figure 5.16. The example shows the BAM model for determining the cycle times of a purchase order process modeled in BPMN (lower part). The upmost part for example expresses the fact that the overall

Attribute	Content
	Characteristics
Identifier	Average activity time
Description	Average time of a process activity within a certain period
To-be value/unit	tbd specifically (min.)
Tolerance range/unit	tbd specifically (%)
Escalation Rules/ Actions	In case of violation alert the process owner and start escalation process (tbd specifically)
Addressees	Process Owner, Middle Mangement, Accountants (tbd specifically) Responsibility Process Owner (tbd specifically)
	Measuring and Computing
Measuring Object	All instances of the process 'Purchase Order'
(Single) Metrics	Start time and end time of all activities of the process
Measuring Method	Read time stamps for beginning and end of activities written by Metasonic Flow
Measuring Frequency	For every single instance as it occurs
Algorithms	For computing period: Sum of processing time of all activities divided by number of instances
Data Sources (general)	Tables in the database of Metasonic Suite: RT_-PROCDESC, RT_PROCINST, REC_PARADESC, REC_RECTRANS, UM_USER
Data Sources (specific)	Activity processing time (for one instance): **SELECT** TIMESTAMP1 (**SELECT** STARTTIME **FROM** RT_PROCINST **WHERE** RT_PROCDESC = *process (purchase order)* **AND** ID = *instance (9)* **FROM** REC_RECTRANS **WHERE** RT_STDESC = *state (fill_in_form)* **AND** RT_PROCINST = *instance (9)* Completed instances: see separate fact sheet .
Computing Period (time, no. of inst.)	Daily
	Presentation
Presentation Style	As-is value and to-be value in combination with a spark line showing the historical development, deviation from to-be value in %
Presentation Frequency	Weekly and in case of escalation
Archiving	Stored in additional database table, linked with RT_PROCDESC

Table 5.1: Fact Sheet for a PPI (extract)

Attribute	Linguistic Pattern	Example
PPI-<ID>	<PPI descriptive name>	PPI-001 Average time of RFC analysis
Process	<process ID the PPI is related to>	Request for change (RFC)
Goals	<strategic or operational goals the PPI is related to>	BG-002: Improve customer satisfaction BG-014: Reduce RFC time to response
Definition	The PPI is defined as {<DurationMeasure> <CountMeasure> <ConditionMeasure> <DataMeasure> <DerivedMeasure> <AggregatedMeasure>} [expressed in <unit of measure>] & The PPI is defin	
Target	The PPI value must { be {greater lower} than [or equal to] <bound> be between <lower bound> and <upper bound> [inclusive] fulfil the following constraint: <target constraint>} } & The PPI value must b	
Scope	The process instances considered for this PPI are { the last <n> ones those in the analysis period <AP-x> } The process instances considered for this PPI are the last 100 ones Source	<source from which the PPI measure can be obtained>
Responsible	{ <role> <department> <organization> <person> } & Planning and quality manager	
Informed	{ <role> <department> <organization> <person> } & CIO	
Comments	<additional comments about the PPI>	Most RFCs are created after 12:00

Table 5.2: PPI Template based on Linguistic Patterns [**?**]

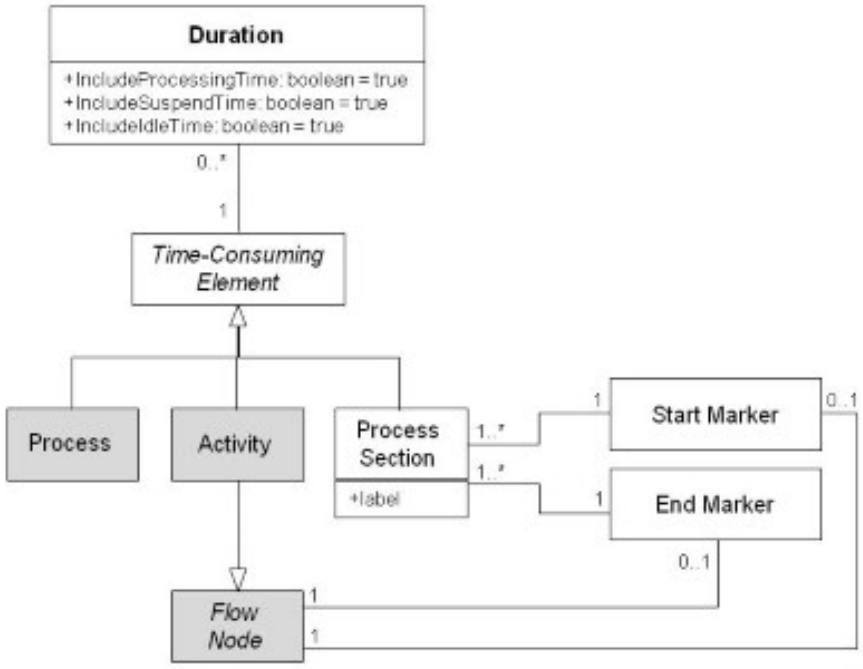


Figure 5.15: Meta Model for Duration (related to BPMN) [**?**]

cycle time (Duration) for the last 50 instances (Filter) has to be determined and displayed on the dashboard (Dashboard). Monitoring the average of the overall cycle time for completed instances controls the modeled business logic of the process. If it is above 48 hours goods are delivered with an express shipping if the average cycle time is more than. Otherwise standard shipping is carried out. A deviation also leads to an alert sent to the process owner, while in any case the average is to be presented on the dashboard. The latter is also valid for the third time-related metric in the example, the partial cycle-time for the company-internal part of the process, which is set into relation with the overall cycle time.

The concept presented by Friedenstab et al. is thoroughly thought-out and clearly and precisely elaborated. The idea now is to adapt it to Subject-oriented Business Process Management and relate the abstract syntax to the S-BPM meta model instead of BPMN. Due to S-BPM being a more precise and comprehensive notation than BPMN [**?**] the mapping should be possible without problems. Table 5.3 compares the BPMN specification elements used by [**?**] with the ones appropriate in S-BPM [**?**].

BAM Language Syntax Construct	**Used BPMN Specification Element**	**Suitable S-BPM Specification Element**
Duration (Time-Consuming Element)	Process, Activity, Flow Nodes	Process, Subject Behaviour States (Function, Send, Receive, Start, End)
Frequency (Countable Element)	Process, Activity, Data Objects, Data States	Process, Subject Behaviour States (Function, Send, Receive), Business Objects and their States
Actions	Process	Process
Measure-based Expressions	Expression, Sequence Flow	Incoming Message

Table 5.3: BPMN and S-BPM Specifications used in BAM Constructs

The remaining constructs as well as the extensions do not depend on the process modeling language and thus are not included in the table. On the other hand S-BPM, following its paradigm of regarding subjects, predicates and objects as equally important parts of a process, offers the subject as an additional specification element to add . In figure 5.17 we modified the picture of figure 5.15 by replacing the BPMN by S-BPM elements and adding the subject. This allows modeling the determination of the overall time a subject (respectively the allocated resource(s)) spends on working on a process instance. This is of interest for cost-related analysis.

In order to show how the BAM language syntax constructs can be related to subject-oriented models we designed the purchase order process in S-BPM. Due to missing information in the BPM model some assumptions were necessary like who performs the process steps (subjects). We then added the BAM modeling symbols to create a monitoring model similar to that in figure 5.17. The result is depicted in the following graph. In the lower part it includes the subject interaction diagram (SID) of the process. The SID shows the subjects involved and how they coordinate themselves in the course of action by exchanging messages. In the monitoring model in the upper part a difference can be seen. The partial cycle time for the company-internal activities can be modeled by just relating the clock symbol to the subject "Sales". In the example this subject represents all

Model for Cycle Times.jpg

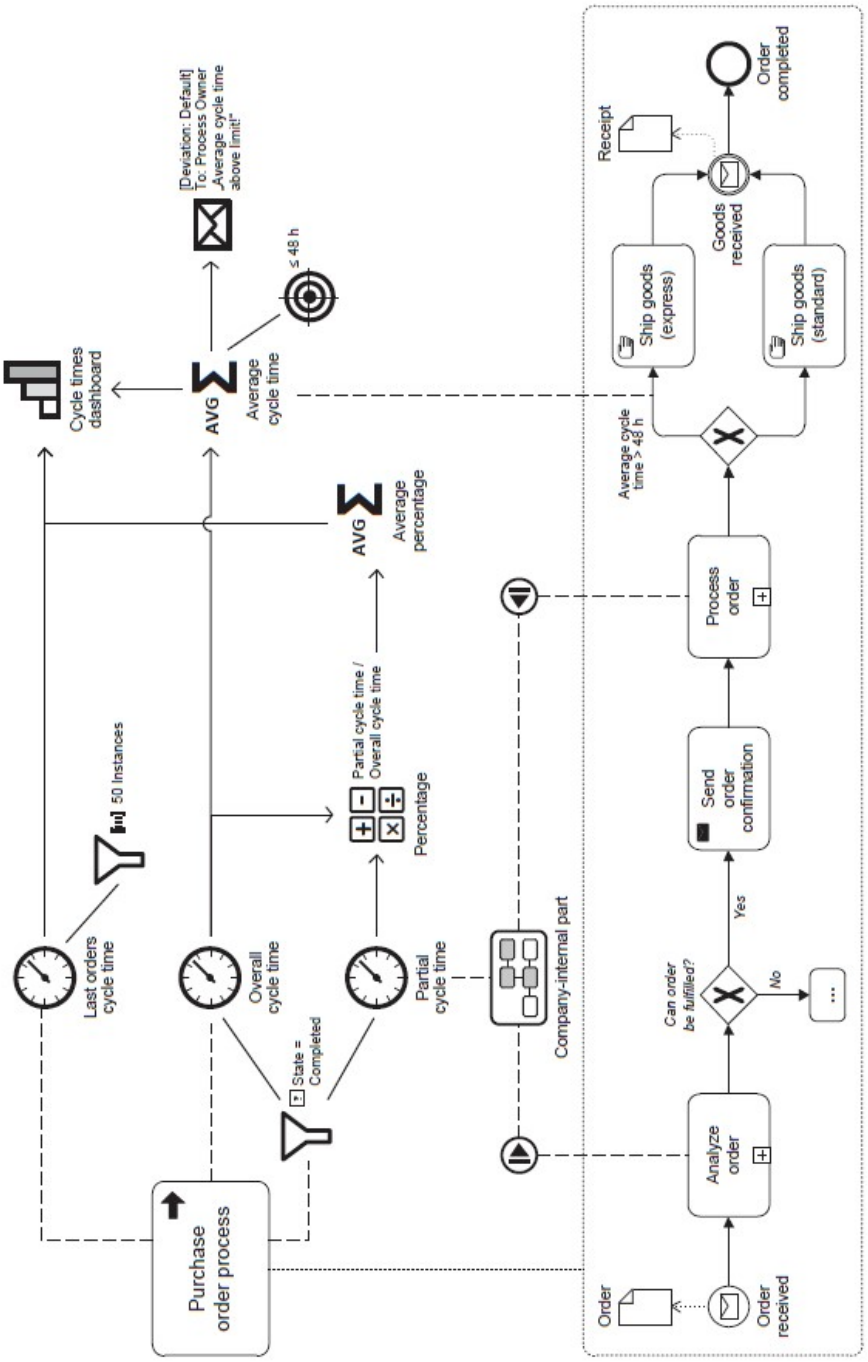


Figure 5.16: BAM Model for Cycle Times of a Purchase Order Process based on BPMN [?]

to-SBPM.jpg

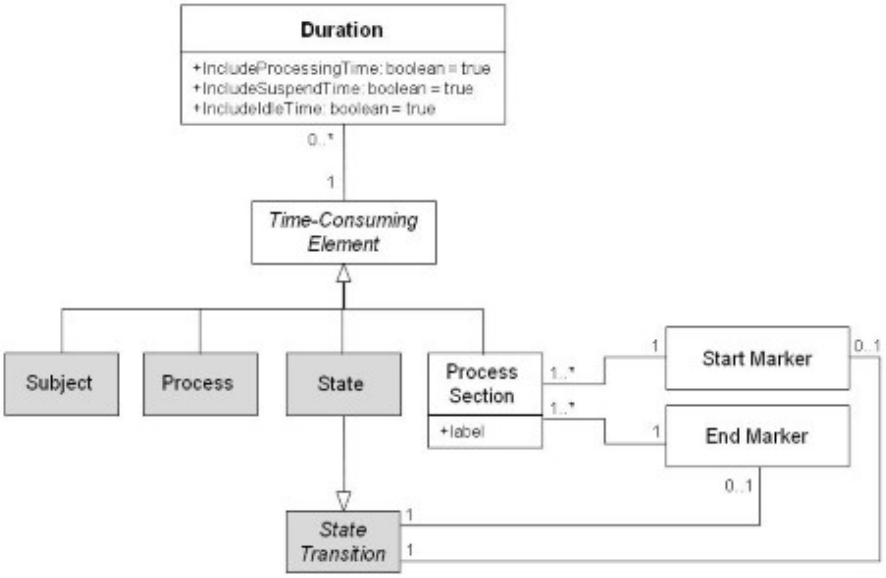


Figure 5.17: Meta Model for Duration (related to S-BPM)

steps carried out within the organization. In the same way we can determine the cycle time for the other subjects.

Given a special information demand a more granular modeling of BAM parameters is possible on the subject behavior level. Figure 5.19 for example details the behavior of "Sales" including all receive, send and functional states walked through by the subject. The symbols indicate that the average cycle time between order reception and confirming the order to the customer should be measured. In the same way cycle times between states in behaviours of different subjects can be modelled.

Back on the level of subject interaction diagram we could also model to determine the overall time for receiving (waiting), sending and doing, both by process and by subject. Modeling on the two diagram levels reduces complexity.

5.3.3 Conclusion and future Work

This contribution systematized Business Process Monitoring and shed some light on the current state of monitoring in the context of S-BPM. Starting there we emphasized Business Activity Monitoring and took a closer look to the modelling of BAM parameters. We showed that the approach for BPMN presented by Friedenstab et al. can be adapted to S-BPM with little effort and that S-BPM shows additional potential to further develop the concept.

5.3.4 Future Work

Due to the novel conceptual integration addressed, several aspects and topics need to be addressed by future research: spacing

- Extension of the structural semantics in OWL with possibilities to add Process Performance Indicators

Cycle-Times-of-a-Purchase-Order-Process-based-on-S-BPM.png

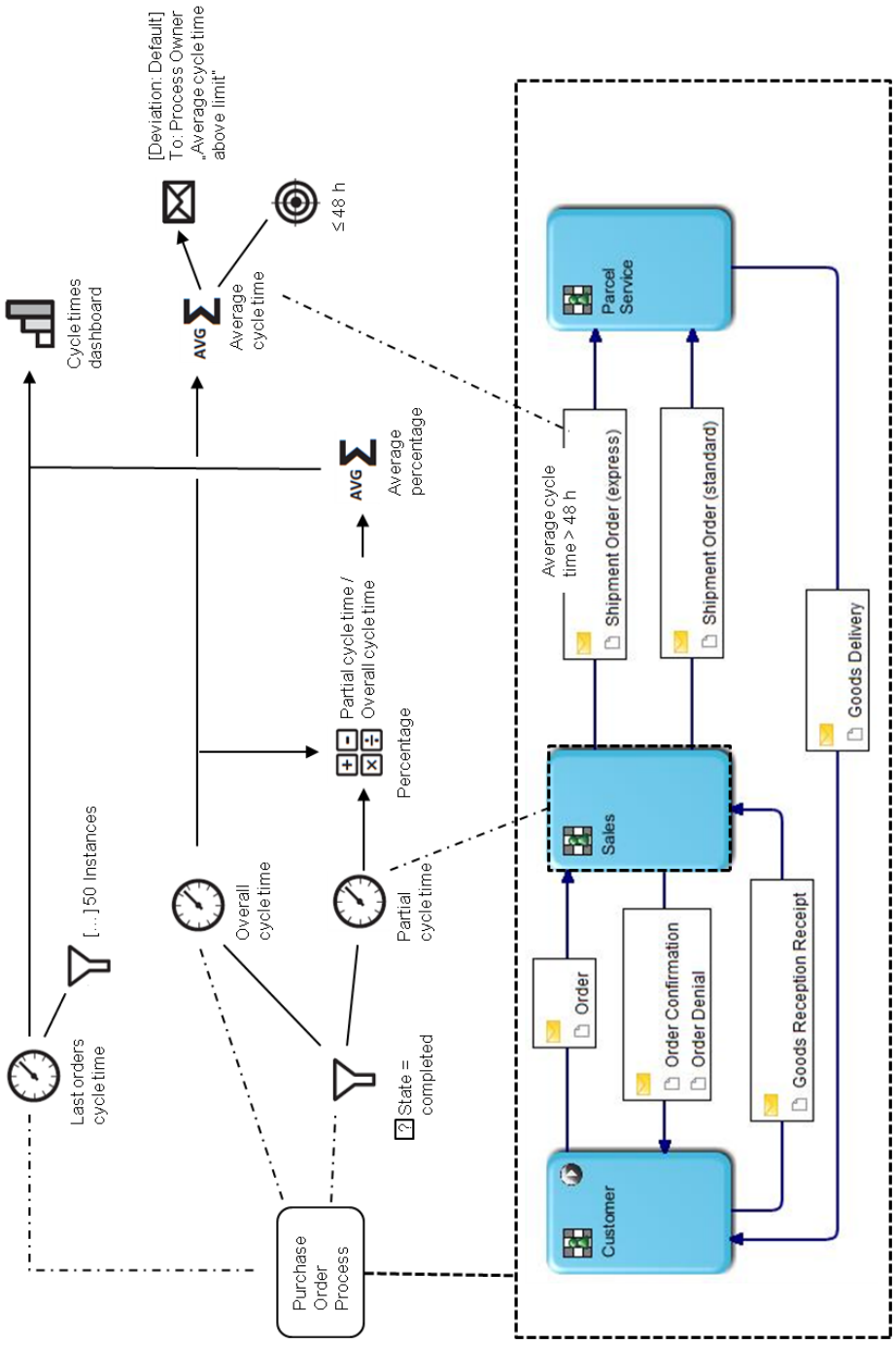


Figure 5.18: BAM Model for Cycle Times of a Purchase Order Process based on S-BPM

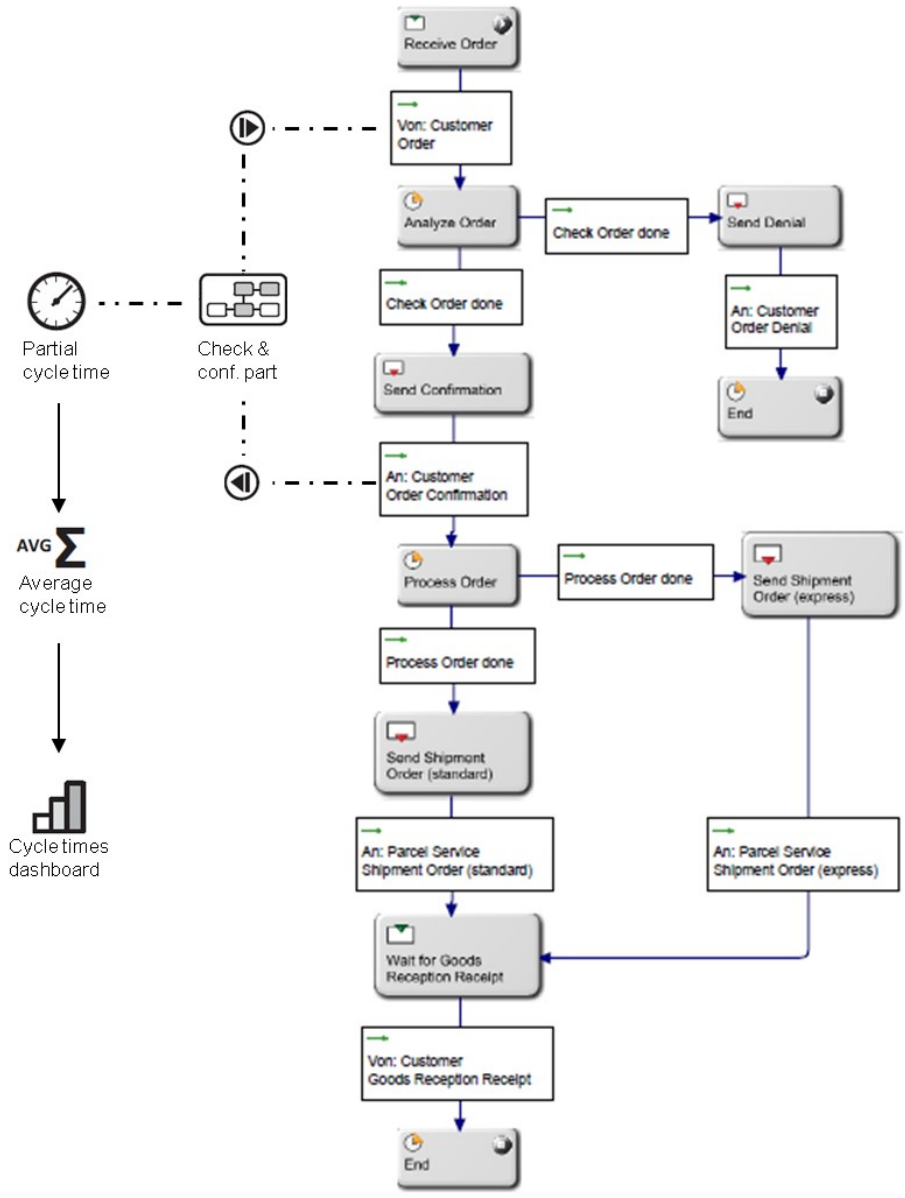


Figure 5.19: BAM Model for Cycle Time of a Process Section based on S-BPM

- ASM definition of execution semantics for throwing events if process performance indicator bounderies are violated.

- ASM definition of execution semantics for handling violation events

5.4 Subject Oriented Project Management

Subject orientation is focused on networks of indeppendent systems, which coordinate their cooperation by exchanging messages. The involved system may belong to different organisations. In our global economy enterprises cooperate around the globe in order to create services or manufacture products for customers which are also distributed all over the world. The challenge of the cooperating partners as a federation of independent systems (virtual enterprise, VE) is to establish smooth cross-enterprise communication to reach the common objectives [**?**]. Information and communication technologies (ICT) are essential to create a federation of independent software systems suitable to execute business processes across the involved companies.
Figure 5.20 shows an example of an order-to-cash scenario where federated applications support a cross-company business process. A dog food store sells its products via internet. It commissions a transportation service provider to deliver the ordered products to the customer, who confirms the reception of the goods. The store deducts the money from the customer's bank account. The process steps are facilitated by several independent software applications and message exchanges (order, order confirmation, delivery notification etc.) enabled by respective communication systems.

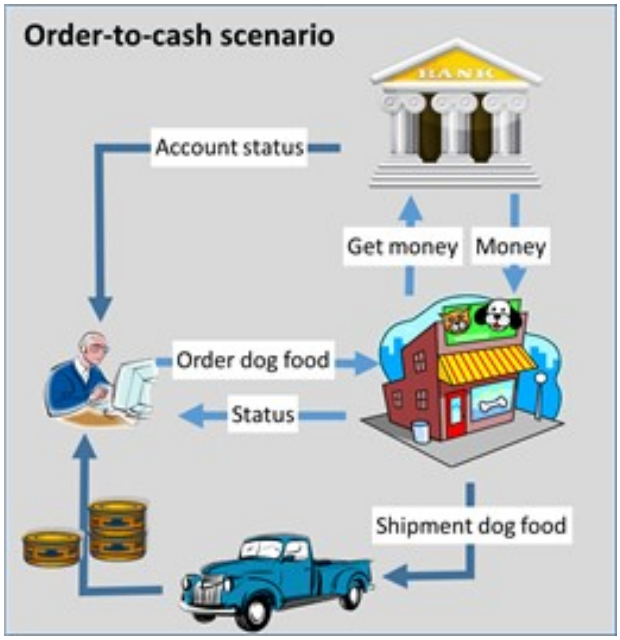


Figure 5.20: Order-to-cash scenario in a federation of enterprises and applications (simplified)

Developing such a mutually adjusted solution by a federation of independent enterprises requires a project management approach different from traditional software development projects taking a process perspective (cf. [**?**]).

Therefore our focus is on how to implement loosely coupled systems for exchanging information between independent partners, rather than tightly coupled solutions for sharing information or other resources. The section is structured as follows. First software development methodology and its elements are reviewed with respect to developing federated systems. This leads to our proposal of a software development approach for federated systems based on subject orientation.

5.4.1 Background

Recommendations for creating federated systems

When independent enterprises develop a federated system a lot of managerial and technological aspects have to be considered, particularly with respect to managing collaborative business processes. This is reflected in the following recommendations (cf. [**?**] , [**?**]):

1. Start the foundation of a federation and identify members.
2. Identify and describe the business services that organizations can provide or they need from partners in service level agreements.
3. Harmonize the enactment of collaboration by coordinating the participating organizations according to defined business processes and identify the systems required for the federation.
4. Integrate the identified and implemented services/systems into the intended application.
5. Maximize the autonomy of organizations when collaborating, thereby ensuring organizations to benefit most from their own business objectives.
6. Represent the partnerships between collaborating organizations when collaborating, and update changes in partnership.
7. Guarantee the business privacy of organizations in the course of collaboration.
8. Allow partners and other third parties to monitor, measure, and oversee the execution of business processes.

Federation of enterprise information systems

[**?**] define virtual enterprises and federations of enterprise information systems as follows: *"The Enterprise partners' Virtual Enterprise (EP VE) is the federation of partners in the community that come together to achieve the goal of a federated distributed system environment, sharing their resources, and collaborating to achieve a common goal: the Federated System VE (FS VE). The partners in the federation retain autonomy over their resources, deciding which resources (personnel, resource dollars, equipment, etc.) are sharable for achieving this goal. The results of this VE are then useable by the partners in furthering their individual systems. The FS VE is seen to be a virtual system of distributed processing components (hardware and software), which are physically implemented and managed by the partners. It is a federation of the partners' systems, where each system retains its autonomy over all processing system components*

and sharable data/information. Retaining autonomy means defining which data or information and software/hardware assets will participate in the federation and be accessible and usable by other systems in the federation."
The definition shows that the focus is on sharable resources. This means when setting up a federation the VE members need to clarify ownership of the shared resources as well as access rights and the rights to change those. Such an approach often implies tight coupling of the involved enterprises and the related resources. Entities leaving a federation then cause difficulties with respect to separating involved systems (changing access rights) and sorting out ownership of information. Alternatively, information can be exchanged between the partners by messages, implying only a loose coupling of the involved systems. In this case the partners only need to agree upon structure and meaning of the data, e.g., using XML schemes, and upon the implementation of the message exchange, e.g., by web services.

Software development methodology

"A software development methodology is a collection of procedures, techniques, tools and documentation aids which help developers to implement software systems" [**?**]. It may include modeling concepts, tools for model-driven architecture, integrated development environments (IDEs) etc. The so-called magic triangle (see figure 5.21) summarizes the various aspects of a software development methodology [**?**].

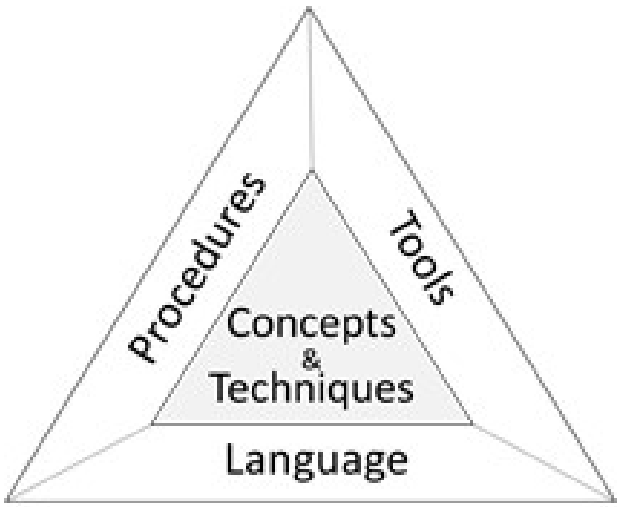


Figure 5.21: Magic triangle of software development methodologies

Concepts and Techniques are used to create models of the software to be implemented, and are thus significantly influencing which languages, procedures and tools are utilized. The applied concept implies the artifacts to be produced, of which the executable software system is the most important one. The Language is used to create the artifacts and tools. Procedures describe the sequence in which the activities for creating the various artifacts are executed. While languages and tools can be replaced without impacting concepts and procedures, the latter are decisively determining the shape of a software development envi-

ronment.

Modeling concepts

Developing a federated system like the dog food store requires modeling cross-company business processes and the entities performing activities in these processes.

Business process modeling. There are various approaches for specifying business process models. IT implementations of those models are called process-controlled applications [7] or workflows. The modeling approaches can be distinguished in three classes: (i) Control flow-based specifications put the focus on the activities. (ii) Object-based models mainly describe business objects and the sequence of operations to manipulate them. (iii) Communication-based models focus on the active entities in a process which exchange messages in order to coordinate their work. By their nature the latter are promising candidates for modeling federations of systems. Business Process Model and Notation (BPMN), the currently most widely discussed modeling language, contains elements for the description of control flows and communication in business processes. In the following we discuss its communication-oriented features. To model communication BPMN provides so-called pools, each representing a process that can exchange messages with processes in other pools. Conversation diagrams are the means to describe this mechanism: However, they do not allow specifying the sequence in which messages are exchanged. Although the sequence can be captured by collaboration diagrams, the semantics of sending and receiving messages is not precisely defined. For instance, it remains unclear whether messages are exchanged synchronously or asynchronously. Additionally a certain message from a pool can only be received in a single activity state, but not in other states. Choreography diagrams in BPMN also define the allowed message sequence between pools. [8] describe a choreography-based tool for specifying global processes. The problem is that choreography specifications cannot contain data. As a consequence a modeler can only describe message sequences being covered by regular expressions, which is the lowest level in the Chomsky hierarchy. This fact makes it impossible to model a behavior like the following: Pool S sends n messages of a type X to pool R. After that S sends a message Y to R. Subsequently S expects m messages of type A from pool R, which received the n messages of type X. The reason for that is that the messages cannot be counted, because data are not allowed in BPMN choreographies. Given these properties of BPMN this notation has significant draw backs for modeling communication, hindering the precise development of federations of systems.

Multi-agent systems modeling. The term agent has multiple meanings. We follow the definition given in [9]: An agent is an entity that performs a specific activity in an environment of which it is aware and that can respond to changes. A multi-agent system (MAS) is a system where several, perhaps all, of the connected entities are agents. The most important property of agents is their controlled autonomy: They independently execute their role-specific behavior, and in multi-agent systems they communicate with each other. These properties are alike those of federated systems which therefore can be considered as multi-agent systems. This means that software development methodologies for agent-

oriented software (for an overview see [8]) can help developing federations of applications.

Procedures

Software Life Cycles (SLC) build a framework for software development procedures. All software development projects follow a series of phases. While software life cycles can be defined in many different ways, each of them comprises the following generic activities: spacing

- Project conception or initiation
- Planning
- Execution with specification and implementation activities
- Termination

In the traditional waterfall approach these activities are performed in the sequence shown above. Other life cycle concepts propose overlapping the development steps, suggest alternatives like the V model or agile development procedures like Extreme Programming and Scrum. [**?**], [**?**] and [**?**] give an overview of the various approaches.

Work break down structure (WBS)

The work break-down structure describes the artifacts to be created in a project in a hierarchical way. A work break-down structure element may be a product, data, service, or any activity results contained in the software life cycle or any combination thereof. A WBS also provides the necessary framework for detailed cost estimating and control along with guidance for schedule development and control. The top level of the WBS should identify the major phases and milestones of the project in a summative fashion. Consequently, the phases used in the top level depend on the software development methodology applied in a project. The first level can either represent the phases used in the software life cycle or the major artifacts of the system to be developed. In case the top level is SLC-oriented it might be built by requirement specification, software architecture, programming, test etc. In the case of an evolutionary life cycle there will be topics like Release 1, Release 2 etc., followed by headlines like requirement specification on the second level. Another alternative is to use top level headlines corresponding to artifacts created by modeling activities, such as 'create communication structure' or 'describe subject behavior'.
The WBS is created during the planning phase of a project life cycle. During this phase the project manager works with the project team to make sure that the client's needs are addressed and the project is planned completely and approved by the client prior to any sort of production beginning on the project.

Organisational break down structure and software architecture

An organizational breakdown structure (OBS) complements the WBS and resource breakdown structure of a project. Project organizations can be broken down in much the same way as the work or product. The OBS is created to reflect the strategy for managing the various aspects of the project and shows the

hierarchical breakdown of the management structure. Hence, the work break down structure has a significant impact on the organizational structure of the project team. The same holds for the phases of the software life cycle and the system architecture influencing the work break down structure. ConwayâĂŹs law states âĂIJorganizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizationsâĂİ [**?**]. A variation of ConwayâĂŹs law can be found in [12]. "If the parts of an organization (e.g., teams, departments, or subdivisions) do not closely reflect the essential parts of the product, or if the relationship between organizations do not reflect the relationships between product parts, then the project will be in trouble... Therefore: Make sure the organization is compatible with the product architectureâĂİ [**?**]. As we look at developing federations of systems with a federation of independent project teams, the system architecture needs to be aligned with the multiple project team structure.

5.4.2 Software Development Methodology For Federated Systems

The software development methodology for federated systems proposed here is based on Subject-oriented Business Process Management (S-BPM)

Development as a multiple-team structure

We now assume that the dog food order-to-cash scenario does not yet exist. The store wants to extend its services for the customers by offering online shopping and home delivery. In order to reach this business objective it takes the initiative to found a federation of enterprises which combine their services and develop a corresponding federation of systems. Each federated enterprise establishes a project team, working on their parts of the solution independent from each other. This leads to a multiple-team project on the federation level [**?**]. As the teams belong to different, independent companies they all have their own development culture and methodology. Since there is no single line management who can assign an overall project manager, the federation members need to agree on a project leader and the competencies related to this role. As the initiator of a federation has the most interest in the development of the federated solution it might be helpful that this company, in our case the store, recruits the leader.

His or her major task is to ensure smooth communication between the independent teams, respectively their managers. The project teams needs to coordinate how the systems they are developing communicate with each other. Their major communication paths are predefined by the communication structure of the system federation. This strategy leads to a high socio-technical-congruence. Figure **??** (CS: no missing) shows the team and communication structure of the dog food order-to-cash federation.

Beside that top-level communication implied by the problem structure, each team can use services offered by other enterprises. Figure 6 reveals that the shipment company uses the service of carriers and forwarding agents, in order to implement the transportation service offered to the dog food shop. This communication relation is of no interest for other federation members and thus should not be visible to the top level teams. It belongs to the internal issues of the shipment project team.

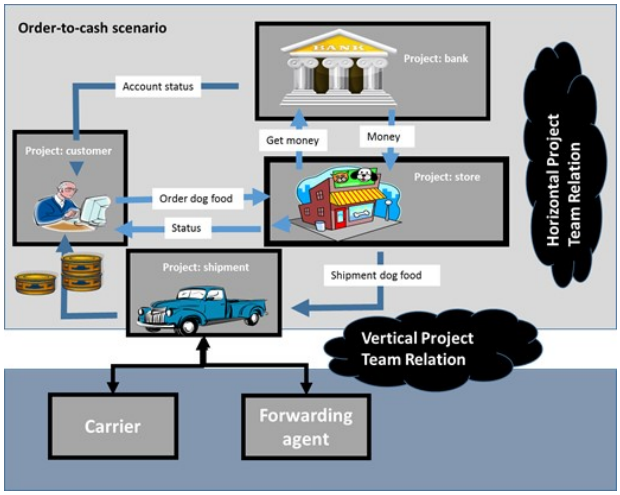


Figure 5.22: Multiple-team project and its communication structure

Development process for federated systems

The artifacts to be created according to subject orientation need to be developed by a federation of teams related to the subject interaction structure.

Specification of the communication structure. The communication between the various members of the federation needs to be specified in more detail. This is done by assigning a subject to each member of the federation and defining the messages exchanged between the subjects. Together with the data transported by the messages a communication model of the system federation is defined. The advantage of the subject-oriented approach is that the system communication structure is directly in line with the communication structure of the corresponding developing teams. The result of that step is the subject interaction diagram (SID).

Specification of the subject behaviour. After defining the communication structure the behavior of each subject is specified. The modelers describe the allowed sequence of messages exchanged on top level and the internal functions of the individual systems. These internal functions represent the services executed by the corresponding federation partner either directly or supported by other service providers. They also encapsulate the communication with those subcontractors as it is of no interest on the top level of the federation.
The behavior of a subject is mainly defined by the corresponding project team, however, in close coordination with the teams responsible for the partner subjects. The teams only need to make sure a message sent to a partner has a receive state in the corresponding subject behavior and vice versa. This pairwise coupling means, e.g., that the behavior description of the shipment company has to contain a state for receiving the âĂIJTransfer orderâĂİ message, transmitted by the related send state in the behavior diagram of the dog food store subject. In order to correctly model these interactions the responsible project teams need also to agree on the interaction sequence of the subjects. However, their internal task behavior (i.e. sequence of functions for task accomplishment) might not become visible to others, as is specified decentralized and might not be shared at all.

Implementation of the input pool. The input pool is the abstract concept for defining the semantics of message exchange. Partners exchanging messages need to agree on how they implement the input pool semantics. Sending requires the sending subject to execute a function to deposit a message in the input pool of the receiver. For each subject doing so an implementation agreement is necessary. Since an input pool is owned by exactly one subject, the functionality for accessing it is local and does not need to be coordinated with the partners. In most cases input pools are implemented as web services.

Implementation of subject behaviour. Each team has to implement the behavior of its subject. This means they have to ensure that depositing and removing messages (including business objects) in or from the input pool are executed and internal functions are invoked in the specified sequence. Workflow engines are appropriate tools for implementing that functionality.

Implementation of internal functions. The internal functions realize the kernel of the service contributed by a partner to a federated application. Messages are the means to cause the invocation of an internal function, and they transport its result to a partner subject. Internal functions can be based on existing systems, e.g., an SAP client. They also can be implemented using another federated solution, or being developed from scratch. The way an internal function is realized is a local decision taken by the corresponding project team.

Operation of a federated system. Beside the development and deployment the non-functional aspects of a federated system need to be agreed upon by the contributing partners. For this purpose they negotiate service level agreements (SLA) defining response time, down time, reaction time in error cases etc. The SLA also includes business aspects like costs and regulations for exceptional situations like a member leaving the federation and bringing in another one.

Federated work break down structure

The various activities described so far can be organized in a federated work break-down structure as shown in figure 5.23.

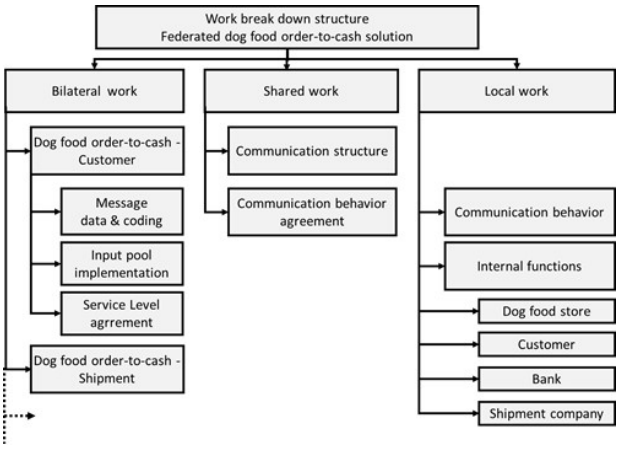


Figure 5.23: Work break down structure for the development of a federated system

The tasks can be divided into three types: **Joint work** concerns the top level of the federation and therefore is done collaboratively by all members of a federation. The major issue on this level is to agree on communication structure and behavior of the entire system, while the behavior of each subject can be described individually by the corresponding member of the federation.
Some work can be done bilateral. Communicating partners, e.g., agree on the coding of the business objects and the implementation of the input pool. They also define the service level agreements.
Local work comprises activities of the development teams which do need to be coordinated with teams of other federation members. A major example in this context is the set of internal functions of each subject, being a local matter, and developed following the particular culture and methodology of the respective team.

5.4.3 Conclusion

We have presented an approach for developing federated systems. The concept considers the characteristics of virtual enterprises combining the services of the partners to satisfy customer needs while keeping legal, organizational, technological and cultural independence. Our communication-oriented view follows the idea that the decentralized structure of federated systems needs to be reflected in the organizational structure of multiple project teams for developing such systems. Those teams belong to separate enterprises and are mutually independent with respect to methodology, technology etc. they use to develop their individual part of the federated system. The proposed approach establishes a layer above the enterprise-specific environments. It helps building coherence on the top level of the federated system solution, while the teams, system elements etc. on the individual level of each federation member keep the highest degree of independence.

5.4.4 Future Work

It has to be investigated whether the OWL definition and/or the execution semantics has to be adapted for a better project management support. Based on that results some guidelines for a subject oriented project management has to be developed and enhanced based on practical experiences.

5.5 Subject-Oriented Fog Computing

Many scenarios related to digitalization increasingly (i) require an easy-to-customize development environment, (ii) capture on-the-edge systems or devices under the control of users or responsible stakeholders. Typical examples are home support systems in healthcare, maker environments producing local goods, and intelligent transport control systems for smart regions. Developing such applications requires architectures that allow to network or compose systems in a modular, while effective and efficient way [**?**]. During the last years, with the advent of advanced equipment and technologies, such as production devices for the private consumer market, networked applications have become common. As a consequence of this trend, a significant issue also appears, namely the increases in the demand of both communication and execution capability. New

applications, such as home care support systems, all deal with complex interaction operations, which should be understood by users, and thus require a high level of abstraction [**?**],[**?**].
Such demands pose significant challenges to existing development paradigms, particularly in terms of edge computing and stakeholder-oriented communication capacities (cf. [**?**],[**?**]). Using behavior abstractions aligning stakeholder needs with communication and processing capabilities in this context is an appealing idea. For instance, in-situ care support devices can be utilized to handle the tasks of preparing the pharmacy order or they can be employed to collaborate with each other to transmitting maintenance messages and sharing resources [**?**]. Besides network technologies, mobile cloud computing is a typical enabler for this demand [**?**].
However, according to Syed et al. [**?**] purely cloud-based systems typically require low latency, support for heterogeneity, mobility, geographical distribution, location awareness, etc. Consequently, Fog Computing (FC) as a near-the-edge-computing paradigm has been defined as a collection of various small distributed clouds deployed closer to the systems or devices at the edge of a communication network (ibid.). Fog applications can be structured along several dimensions, either directly or indirectly referring to stakeholder interaction [**?**]: spacing

- Geo-distribution: wide (across region) and dense - high population of events, such as ramp accesses in traffic, sensor systems in production halls, clustering medical devices in home healthcare application development
- Low/predictable latency: tight within the scope of a certain location - intersection, production isle, treatment room
- Fog-cloud interplay: data at different time scales - sensors at intersection/traffic info at diverse collection points, supply chain monitoring/production control in process industry, monitoring body condition/treatment planning procedure in healthcare
- Multi-agencies orchestration: Agencies that run the system coordinate policy implementation at the same time, e.g., traffic authority runs light system while controlling law policies in real time; active elements for production control implement also governance regulations; home healthcare support is effective with respect to medical treatment and personal well-being.
- Consistency: adjusting demands and capabilities, such as getting the traffic landscape demands a degree of consistency between collection points, aligning engineering with production processes, or ensuring well-being while adapting medication to patient needs.

In this contribution, we present Subject-oriented Fog Computing (SFC), a choreographic approach and multi-layered infrastructure for Fog Computing. Separating modeling from organizational and technical implementation along a staged procedure it aims for supporting system architects, designers, and developers, who are interested in stakeholder interactions when building Fog Computing solutions. We propose a development and software architecture scheme without platform dependencies, open for various networked settings. It is based

on behavior abstractions termed subjects that integrate a socio-technical design perspective, and allows composing applications from a stakeholder perspective (cf. [6-8]). In the following section we review related research to developing fog applications according to stakeholder needs in various domains. Subsequently, we introduce SFC based on a System-of-Systems perspective, and provides an exemplary case from developing home healthcare support systems. Finally, we conclude summarizing SFC and indicating further standardization activities.

5.5.1 Fog Computing and Subjects

We introduce Fog actors by starting with the encoded System-of-System perspective, sketching the federated nature of choreographic ecosystems (subsection above). We then provide the basic modeling notation and exemplify Fog actors as subjects for a home healthcare scenario. Finally, the corresponding Fog runtime system is sketched in terms of its application along the organizational and technical development phases.

Federated Systems

When considering Fog Computing as an addition to cloud ecosystems we expand software architectures to include systems outside the software system which interact with the software system [**?**]. Each component of the ecosystem can be represented as a system using behavior models. Thereby, cloud ecosystems can serve as service providers for the nodes of the network (of applications). The Fog network enriches the cloud ecosystem, e.g., for specific purpose like home healthcare with domain-specific models.

Since these enrichments are compound systems, a System-of-Systems (SoS) perspective helps conceptualizing the construction and development of Fog applications [**?**]. SoS have as essential properties 'autonomy, coherence, permanence, and organization' (ibid, p.1) and are constituted 'by many components interacting in a network structure', with most often physically and functionally heterogeneous components. For instance, home healthcare applications comprise support systems for dementia, blood pressure measurement, and pharmacy shopping, and need to be adaptable on-the-fly in case of changing operational conditions (cf. [**?**]).

Since users tend to develop applications incrementally, their specifications are adapted to changes dynamically. Once these specifications in terms of SoS models become executable, users can interactively bootstrap their modifications. Behavior can be deployed, once being specified and validated. Utilizing subject-oriented modeling and execution capabilities (cf. [**?**]), systems or subjects are viewed as emerging from both the interaction between subjects and their specific behaviors encapsulated within the individual subjects. Like in reality, subjects as systems can operate in parallel and exchange messages asynchronously or synchronously.

Subject-oriented Representation

According to the SoS perspective, Fog applications operate as autonomous, concurrent behaviors of distributed Fog actors. A Fog actor or subject is a behavioral role assumed by some entity that is capable of performing actions. The entity can be a human, a piece of software, a machine (e.g., a robot), a device (e.g., a sensor), or a combination of these, such as intelligent sensor systems.
When subject-oriented concepts and development techniques are applied, SoS

subjects can execute local actions that do not involve interacting with other subjects (e.g., calculating a threshold value for medical intervention and storing a pharmacy address), and communicative actions that are concerned with exchanging messages between subjects, i.e. sending and receiving messages. Subjects are one of five core symbols used in specifying designs. Based on these symbols, two types of diagrams can be produced to conjointly represent a system: Subject Interaction Diagrams (SIDs) and Subject Behavior Diagrams (SBDs).
SIDs provide an integrated view of a Fog SoS, comprising the subjects involved and the messages they exchange. The SID of a home healthcare support process is shown in Figure fig:homeCare. The aim of such systems is not only to support patients when needing healthcare at home, but also to profit from networked services, in particular, getting drugs in time from pharmacy, receiving in-situ service when required, and intelligent networking of local devices, while being scheduled for managing everyday life and being reminded of individual caretaking activities (cf. [**?**]).
Home healthcare comprises several subjects involved in near-edge communication: A Personal Scheduler coordinating all activities wherever a patient is located (traditionally available on a mobile device), a Medication Handler taking care of providing the correct medication at any time and location, Blood Pressure Measurement sensing the medical condition of the patient, and Shopping Collector as container for all items to be provided for home health care. In the figure the messages to be exchanged between the subjects are represented along the links between the subjects (rectangles).
In-situ, and thus near-edge communication is required for delivering Blood Pressure Measurement data to the Personal Scheduler and the Medication Handler, as the patient handles the measurement device at home and needs to know, when to activate it and whether further measurements need to be taken. Another need for near-edge communication is given through the Shopping Collector: It receives requests from both, the Medication Handler when drugs are required from the pharmacy, physician, or hospital, and the Personal Scheduler, in case further shopping for the patient is required. As such, the Shopping Collector serves as an interface subject for shopping services to the homecare environment.

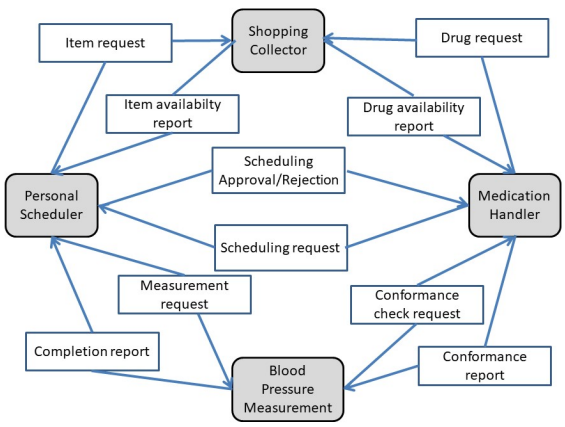


Figure 5.24: Example of home care support (SID)

As usual Subject Behavior Diagrams (SBDs) provide a local view of the process from the perspective of individual subjects.
Given these capabilities, SoS Fog designs are characterized by (i) simple communication protocols (using SIDs for a process overview) and thus, (ii) standardized behavior structures (enabled by send-receive pairs between SBDs), which (iii) scale in terms of complexity and scope.
Subject-oriented Fog Computing (SFC) allows meeting ad-hoc and domain-specific requirements. As validated behavior specifications can be executed without further model transformation, stakeholders can guide the implementation of specification, representing domain-specific task flows, and make ad-hoc changes by replacing individual subject behavior specifications during runtime. Due to the distributed nature and loose coupling of subject-oriented representations, the ultimate stage of scalability could be reached through dynamic and situation-sensitive formation of edge systems.
SFC structures SoS, e.g., when federating a blood pressure measurement device with a personal health scheduling systems, according to their communicating with each other. When these devices need to communicate directly with the cloud, e.g., as required in case of maintenance, or calling a specialist for medication, this link is encoded in the diagrams and executed during runtime after technical implementation. On the modeling layer the activity is a request sent to another subject, waiting until an answer is received, and processing the received answer.

Execution

Once a Subject Behavior Diagram, e.g., for the Blood Pressure Measurement subject is instantiated, it has to be decided (i) whether a human or a digital device (organizational implementation) and (ii) which actual device is assigned to the subject, acting as technical subject carrier (technical implementation) (cf. [**?**]). Typical subjects as edge devices are smart devices, which can have Internet connectivity, including smart phones, tablets, laptops, healthcare devices, etc. The subject-oriented runtime engine [**?**] is then a Fog Computing infrastructure providing low-latency virtualized services and is linked with the Cloud Computing infrastructure by the same subject interaction mechanism. As there can be a variety of edge devices, such a Fog Computing platform also needs to manage and control these devices (see also foglets described below).
Size, storage capacity, processing capabilities, and latency increase as we move closer to cloud computing. The subject-oriented Fog acts as an intermediate layer between the edge devices and the cloud. Edge devices request computing, storage and communication services from the Fog according to the subject-oriented communication scheme. The Fog provides local, low latency response to these requests and forwards relevant data for computationally intensive processing, long-term analytics and persistent storage over to the cloud. Figure[Fog Computing Architecture]Fog Computing Architecture provides a schematic visualization of this constellation, as it can be used for implementing the sample home healthcare support system.

With respect to the home-healthcare example, a typical infrastructure comprises local devices and their interconnected services, such as linking the Blood Pressure Measurement to the Personal Scheduler. These subjects can be either linked to an IoT SoS, e.g., coupling several sensor systems, or to Cloud services, as for accessing public databases when checking reference or availability data,

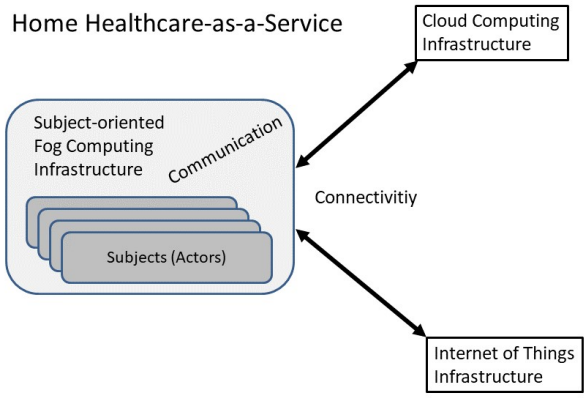


Figure 5.25: Fog Computing Architecture

depending on the state of affairs in the home healthcare setting.
Fog nodes are subject carriers representing resources including hardware (computing, networking and storage) capabilities. They provide âĂŸlocalâĂŹ real-time data processing capabilities, and, despite multi-tenancy, can execute applications in isolation to prevent unwanted interference from other processes. Policies to control service orchestration, filtering, and for adding security can be implemented dedicating a specific control subject, since the primary scheme of control is choreography.
The approach scales, due to the decentralized management mechanisms allowing to setup, and configure a large number of devices in the Fog. In this context, subjects correspond to foglets (cf. [**?**]), i.e. software agents for each fog node, monitoring the state of the node and services. A subject can use abstraction tier APIs to monitor the state associated with (physical) devices and services deployed on this device. It analyses the entire information (encoded in an SBD), and delivers it to receivers linked through messages for further processing. These subjects can also perform lifecycle activities. As demanded by Vaquero et al. [**?**], SFC comprising a fog abstraction layer provides uniform programmable interfaces for resource control and management.
According to the S-BPM concepts, normalization can be used to abstract essential behavior patterns. For instance, in case Blood Pressure Management requires a machine-dependent procedure, its action behavior (performing functions) as a subject can in principle contain many internal functions which are performed in sequence, in order to accomplish an assigned task. In these sequences of internal functions, no sending and receiving nodes are included. Accordingly, extensive and therefore confusing behavior diagrams can be avoided. Since these sequences of internal functions are not important for communication, model representations can be simplified, and normalized behavior can lead to larger functions by hiding functional details. Actually, for the sake of understanding the home healthcare setting, the subjects shown in Figure 5.24 have been normalized.
In case the communication patterns are generalized, the process-network feature of S-BPM facilitates representation. For instance, when the Shopping Collector needs to collect sensor data from various storage devices, such as a refrigerator or a food isle, its communication requests and the respective replies can be denoted in a summative way. In SFC this feature helps representing mutually

dependent processes, i.e., when subjects of a near-edge process communicate with subjects of other (near-edge) processes. As shown in Figure 5 the Home care near-edge process interacts with the Goods delivery process through the Personal Scheduler. In this case, the interaction is not further detailed, rather indicated through directed links. The same holds for the interaction between the Shopping Collector and the Medication Handler, which helps ensuring the quality of drug support in the Medicare process.

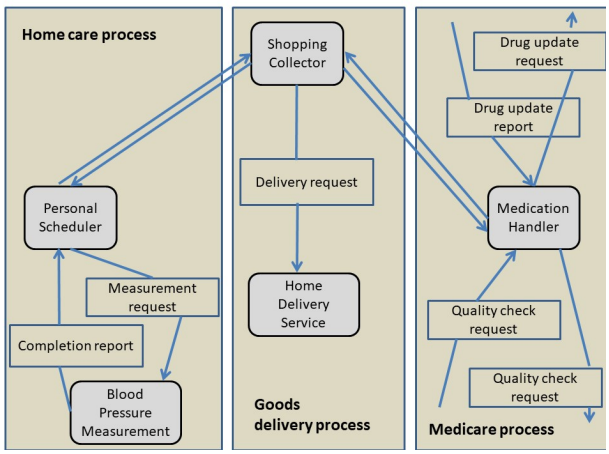


Figure 5.26: Extended subject interaction diagram for the process 'home care'

For SFC implementation the open source engine UeberFlow [?] can be used. Hereby, SFC actions or tasks are ordered in the sequence as defined through SBDs and SIDs. The Workflow Specification of UeberFlow represents an entirely executable model of an application, given the subject actions and communication with others. It acts as container for so-called WorkflowUnits that are created for each subject, and captures all activities (WorkflowSteps). In addition, WorkflowUnit manages the data processed by the WorkflowSteps and its WorkflowFunctions. Consequently, Fog applications are executed through WorkflowSteps.
Thereby, the WorkflowFunctions are the most fine-grained units of execution in the UeberFlow Language meta-model, and define the actual execution logic of a WorkflowStep, its prerequisites and results. Once a step is triggered, a specific sequence of WorkflowFunctions is executed. The WorkflowFunctions can be one of 6 different types. For each of them an Actor has been implemented utilizing the Akka framework (http://akka.io/). Hence, an instance in UeberFlow is equivalent to all actor instances created in the context of this particular workflow instance. All of those actor instances are aggregated using the actor structuring and supervision mechanisms by defining a root actor representing the entire instance.

5.5.2 Conclusion

Fog Computing (FC) as a near-the-edge-computing paradigm has the potential to improve user support. When defined as a collection of various small distributed clouds deployed closer to the systems or devices at the edge of a communication network subject-oriented applications support spacing

- wide and dense geo-distribution due to their behavior abstraction, as e.g.,

required for home healthcare support systems, linking not only (medical) devices at home, but also medical infrastructure (physician, pharmacy, nursing services etc.) from the region

- low or predictable latency due to the runtime concept of parallel processing
- cloud interplay of Fog nodes, due to separating specification from technical implementation which allows for processing data at different time scales, e.g., when monitoring body condition and supporting a patient treatment planning procedure
- multi-agencies choreography, loosening the need for orchestration, due to the inherent concept of choreography in subject-oriented architecting. Hence, Fog actors or subjects only need to be synchronized as tight as required, e.g., when a running monitor subject requires coordination with healthcare policy implementation at the same time
- consistency, due to mapping all respective requirements to corresponding interaction patterns. Hence, demands and capabilities can be adjusted specifying message exchange patterns, in order to ensure overall consistent system states, either through subjects working in parallel, or through information distribution triggering further subject behavior.

Our future standardization effort will focus on including for networking information into the subject-oriented behavior abstractions, to enable modeling stakholder-specific settings according to their case-specific needs and available Fog actors. Once stakeholders are able to edit and validate the subject behavior models, they also can deal with organizational and technical implementation details, allowing them to adapt an entire application as System-of-System dynamically. Adaptation to new policies can be implemented in this way (cf. [**?**]), leading to more situation-sensitive Fog applications (cf. [**?**]).

5.6 Activity Based Costing

CS: table refs are incorrect

5.6.1 Basic Concepts

Process Controlling

Process controlling has both a strategic and an operational dimension [cf. e.g. [**?**], p. 229 ff.]. We concentrate on methods and techniques for planning, designing and coordinating the supply of information necessary to allow continuous operational process controlling with key figures as indicated in the closed-loop approach to performance management (see lower part of figure 5.13 in 5.3). As operational process controlling aims for post-execution analysis of business process instances it can be complemented by Business Activity Monitoring (BAM) which, based on event processing concepts, observes instances during execution and sets alerts or triggers actions in real-time or near real-time according to the particular situations identified (cf. [**?**], [**?**]).
The major question is how to measure process performance. A typical parameter for the evaluation of process effectivity is customer satisfaction while process time, quality and cost and adherence to schedules are suitable to assess

efficiency (cf. e.g. [?] p. 229 ff.). As these parameters have high significance for the competitive position they are crucial for process controlling. While the assessment of customer satisfaction and maturity levels of processes usually are matters of periodic monitoring activities there might be other, more technical parameters needed to be watched permanently, like the response time of application systems. Those aspects are especially relevant if processes are extensively supported by IT.
Figure 5.27 gives a conceptual overview of a key figure-based operational process controlling, split into continuous and periodic or occasional controlling activities, like it could be set up successively for the S-BPM approach. The integrated collection and analysis of common managerial data allows for a cohesive evaluation and control in terms of process controlling [cf. [?] p. 248 ff., 1 p. 385 ff., 7 p. 158 ff.]. It feeds back results to take decisions and actions, fostering a steadily growth of experience regarding the interdependencies between cost, quality and time (organizational learning).

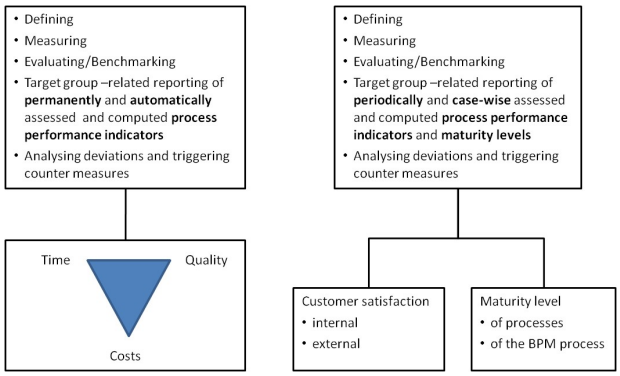


Figure 5.27: Operational Process Controlling

In this section we focus on cost figures. Calculating them is more difficult than assessing those for time and quality. S-BPM allows determining cost figures for processes and process steps as well as for the occupation of cost centers and organizational units with little additional effort though. The reason is that S-BPM specifies subjects as actors in a process, their interaction and their assignment to elements of the organizational structure (organizational units, positions, roles).
The basic methodology to integrate such cost information into process controlling is Activity-Based Costing (ABC).

Methodology of Activity-Based Costing

The concept of Activity-Based Costing origins in the work of Miller and Vollmann [?] and Cooper and Kaplan [?] and was established in the German-speaking community by Horvath und Mayer [?].
ABC roots in a simple fact: producing and delivering a product or service involves many activities within cost centers and across boundaries of cost centers or functional areas, all causing costs. Major factors influencing these costs (cost drivers) usually are measures of the activity quantity, e.g. the number of purchasing orders being processed in procurement.

Step 1: Analysing activities

Starting point for ABC is an analysis of activities performed in the cost centers, using common methods like interviews, questionnaires, self-monitoring, third-party observation, document analysis or multi-moment recording. This analysis is essential for bordering cost center internal process steps and main processes running across cost center boundaries. Self-monitoring and multi-moment recording can bring up time standards for the execution of processes and their steps, but needs high effort. In order to ease the investigation of times, controllers instead often conduct interviews to find out what share of work force capacity the process steps occupy in a cost center. The analysis results in a transparent, hierarchical process structure showing the assignment of activities to process steps, the assignment of process steps to cost centers and the aggregation of process steps to main processes (cf. figure 5.28)

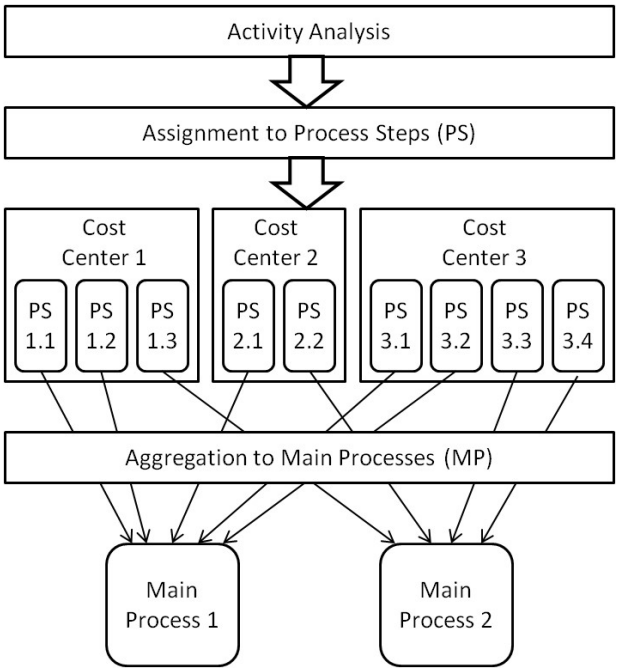


Figure 5.28: Process Structure

This first step of Activity-Based Costing can be based on the results of the activity bundles analysis and modeling in S-BPM [6]. ABC-relevant information on subjects, their activities and the business objects being worked on are contained in the S-BPM process model (see section 2.3).

Step 2: Determining cost drivers
Horvath und Mayer differentiate between activity quantity induced (aqi) and activity quantity neutral (aqi) processes [9]. The latter (e.g. leading a department) cause costs independent from the activity quantity (e.g. the salary of the department leader). In activity quantity induced processes (e.g. purchasing goods) resource consumption and related costs vary with the activity quantity. Hereby process costs incurring depend on the number of cost drivers and there is need to determine measures for those as a second step in ABC. Cost drivers serve as an allocation base for resource utilization and thus also for causing costs.
Cost drivers need to meet some requirements in order to make cost dependence

transparent: spacing

- Process costs should, at least in the long term, vary with the activity quantity
- Cost driver values should be easy to assess and to understand.
- Input of resources should be approximately the same for all process instances, otherwise processes and cost drivers need to be further differentiated.

Usually ideas of what the major cost driver is already come up during the activity analysis and the activity quantity can be determined simultaneously then.

Step 3: Determining process costs
In practice it is often difficult to only assign the major cost driver to the main processes, because a main process can consist of process steps with different cost drivers not being proportionally related.
In a given organizational and cost accounting context resources and costs are planned and actual costs are recorded on cost center level. This means planning and assessing process costs initially is also related to cost centers.
Although theory suggests to plan process costs analytically and by cost type like in direct costing, practitioners prefer more simple concepts. One alternative is to only plan labor costs analytically and to allocate all other cost types proportionally. Another option would be to assign to the analyzed processes the capacities they consume and the related costs. In any case the accountants usually assume the labor cost to be the major cost element. Process costs then can be computed by multiplying a qualified estimate of the number of employees involved in the process by their average wage. If need be activity quantity neutral costs also can be passed on proportionally. In case there are more activity quantity induced costs with a significant extent they need to be considered in addition to labor costs. Even then the described procedure is still easy to handle.

Step 4: Determining process cost rate
In a last step, for the purpose of job order costing or product calculation, a simple division results in a process cost rate similar to the computation of a machine hour rate. As shown Activity-Based Costing can be implemented in various ways. The identified problem of different cost drivers that cannot be aggregated can be solved by using time-related allocation bases [**?**] p. 23.
The concrete process times can be computed if a workflow engine writes time stamps for begin and end events of the process steps. In order to have the engine processing a workflow at runtime, process activities must be assigned to concrete actors. Both kinds of information are needed for establishing ABC as elaborated in section 5.6.2.

5.6.2 BPM as Data Supplier

Subject-oriented Business Process Management (S-BPM) focuses on the acting elements (actors) and their interactions as they drive a process. Its modeling notation includes all building blocks of a complete sentence in natural language as there are subject, predicate and object. The clear formal semantic of the underlying process algebra makes it possible to automatically generate code and makes

subject-oriented process descriptions executable at a finger tip [**?**], [**?**].
Major parts of the model are subject interaction diagrams, describing the subjects involved in the process and the messages they exchange, and subject behavior diagrams, specifying subject activities as there are sending and receiving messages and other functions (e.g. manipulating business objects). The ladder means that at a time subjects can either be in a send, receive or functional state. Transforming the model into a workflow and integrating IT solutions (e.g. ERP functionality) to support particular activities is subject to the embedding of the process into IT. Assigning subjects to elements of the existing organizational structure (organizational units, positions, roles) being responsible for carrying out the activities as defined in the model, is called embedding the process (model) into the organization. Existing directory services based on Lightweight Directory Access Protocol (e.g. Active Directory) can ease the assignment of subjects to roles, groups and people as implemented in Metasonic Suite. A process engine like Metasonic Flow interprets the model at runtime, instantiates process instances and controls their execution. According to the defined behavior the engine involves users and IT services or applications as subject representatives. It also controls the handling of business objects included the in subject behavior (creation, modification, deletion, exchange through messages). During execution the engine can capture many single pieces of data relevant for process controlling, especially by setting time stamps for state transitions and by counting instances. Examples are

spacing

- begin time and end time of every single instance,
- begin time and end time of the single steps within an instance or
- number of instances of a certain process per time unit.

Using such raw data suitable software can compute key figures like spacing

- waiting time of an instance from the moment it appeared in the in-box of an actor until he or she takes it out for processing (per case, on average) and
- processing time from taking the instance out of the in-box until putting the result into the out-box (per case, on average).

This means the workflow system generates a valuable data basis for a meaningful Activity-Based Costing. This data needs to be categorized though and the key figures need to be defined precisely and unique in order to derive useful management information (see example in section 5.6.2).

Example for Estimating Process Costs in S-BPM

Effective process controlling, allowing to turn such decisions into the right actions additionally requires information about cost dimensions in processes and cost-related consequences of processes for cost centers. Cost information enables monetary valuation of the enterprise performance as well as identifying weak points in operations and valuating their economic impact.
We exemplify the determination of process costs using an order process. Figure 5.29 depicts the behavior diagram for the subject 'purchaser' enriched with

some time information. These are times recorded as time stamps for state transitions by the process engine and stored in its event log [**?**]. For clarity reasons in the figure 5.29 we only added time stamps for one state and just added the duration for the others.

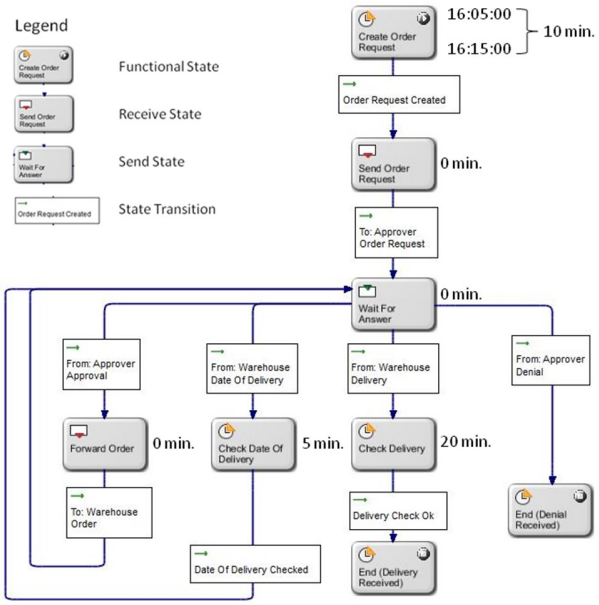


Figure 5.29: Calculating Processing Time of the Subject 'Purchaser'

In the example the processing time of the subject 'purchaser' in case of the sunshine path (goods are on stock) can be computed by adding up the differences between start and end time of the functional states 'create order request' and 'check delivery'. The sunshine path sequence is as follows: After creating the request the purchaser sends it to the approver and then waits for an answer. If the request is denied the instance ends (right column in figure 5.29). Otherwise the purchaser forwards the request to the warehouse (left column), waits for them to announce the delivery date and checks it (second to left). Finally he waits for the delivery and checks it after reception, before the instance comes to an end (third to left). As a simplification we assume the subject representatives to work permanently when performing functions (being in a functional state). We did not insert times for send and receive states because message exchange is considered to be accomplished electronically with no latency for sending, transmission and receiving. Applying this procedure to all subjects could lead to the result in table 5.4.

Subject	Processing Time
Purchaser	35 min
Approver	10 min
Warehouse	30 min
Invoice Verification	5 min
Accounting/book keeping	5 min
Total	85 min

Table 5.4: Processing times

For estimating the costs of the process we need an hourly or minute-related rate of wage for the people being assigned to the subjects. It is also possible to provide those values aggregated on group or role level. In Figure 5 we visualized how the subjects in our example are mapped to persons, while table **??** gives an overview of the rates for the employees involved as subject representatives.

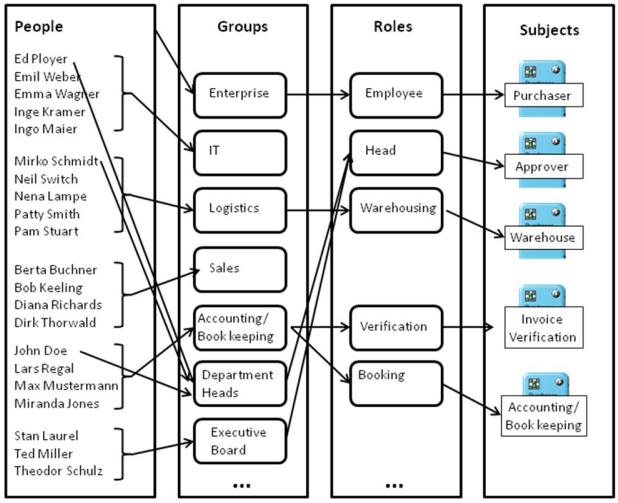


Figure 5.30: Embedding the Subjects into the Organization

Employee	**Wage rate**
Miller, Laurel, Schulz	200 Euro/hour
Ployer, Schmidt, Doe, Keweling	100 Euro/hour
Weber, Wagner, Kramer, Meier, Switch, Lampe, Smith, Stuart, Buchner, Richards, Thorwald, Regal, Mustermann, Jones	50 Euro/hour

Table 5.5: Hourly wages rate

Having these time and wage values available it is a simple multiplication to determine the personnel-related process costs for every single instance (cf. table **??**). Considering a sufficient number of instances over a representative period of time allows computing a valid average cost value.

Subject	**Employee**	**Costs**
Purchaser	Kramer	35 Min. x 50 Euro/60 min. = 29,17 Euro
Approver	Ployer	10 Min. x 100 âĆň/60 min. = 16,67 Euro
Warehouse	Lampe	30 Min. x 50 âĆň/60 min. = 25,00 Euro
Invoice verification	Regal	5 Min. x 50 âĆň/60 min. = 4,17 Euro
Accounting/Book keeping	Regal	5 Min. x 50 âĆň/60 min. = 4,17 Euro
Total		79,18 Euro

Table 5.6: Personal Related Process Costs

Assigning employees as subject representatives embeds the subjects into the organizational structure, because people belong to organizational units. As cost centers usually also are assigned to organizational units it is now possible to

determine the costs of a process incurring in a certain cost center. From the cost center perspective it is also possible to see how its total costs are distributed over the processes and process steps it is involved in. Table **??** shows examples for cost figures which can be computed.

Key figure (Euro)	**Computation (e.g. for 10 work days)**
Process costs per process step/process	Multiply all processing times by the appropriate wage rate and aggregate the products over all instances occurring during the observation period
Process costs per cost center	Multiply all processing times incurring in the cost center by the appropriate wage rate and aggregate the products over all instances occurring during the observation period

Table 5.7: Cost Figures

As mentioned before key figures need to be defined carefully and precisely. A useful instrument helping to assure this are structured fact sheets being filled in with all necessary information [**?**]. Table **??** depicts such a fact sheet created for our purposes [**?**]. A more formal structure can be found in [**?**].

5.6.3 Conclusion

With the example in chapter 3 we could show that it is relatively easy to integrate cost information into S-BPM. Focusing on personnel costs as suggested avoids the problematic proportioning of costs and therefore is particularly suited for people-intensive areas with a high degree of indirect costs as it is characteristic for services.

5.6.4 Future Work

A more detailed investigation of how the implementation of Activity-Based Costing can benefit from a preceding S-BPM implementation seems to be promising. Exploiting the conceptual particularities coming with and the data collected by S-BPM seems to bear considerable potential of savings when introducing ABC. Processes are defined and modeled and as-is process quantities per period are available as well as the distribution of the overall capacity of the cost centers over the process steps. These parameters allow determining a standard time for processes which lays the ground for planning process costs. Next steps could be to extend the example by determining and specifying more key figures and testing them with representative numbers of instances of different processes. Learning from this could help to further elaborate the ABC concept for S-BPM. The OWL specification has to be extended with features which support Activity based costing. This includes the data which should be collected and the structure in which these data are stored.

Attribute	Content
	Characteristics
Description	Average costs of a process activity for a certain period
To-be value/unit	tbd specifically (Euro)
Tolerance range/unit	tbd specifically (%)
Escalation rule	In case of violation alert the process owner and start escalation process (tbd specifically)
Responsibility	Process Owner (tbd specifically)
	Measuring and Computing
Measurement	Read time stamps written by Metasonic Flow, compute processing time as difference between time stamps for beginning and end, multiply processing time by hourly wage rate, divide product by number of completed instance
Algorithms	âĹŚ Processing time * hourly wage rate âĹŚ completed instances
Data sources(general)	Tables in the database of Metasonic Suite: RT_PROCDESC, RT_PROCINST, REC_PARADESC, REC_-RECTRANS, UM_USER
Data sources(specific)	Processing time: **SELECT** TIMESTAMP1 (**SELECT** STARTTIME **FROM** RT_PROCINST **WHERE** RT_PROCDESC = *process* **AND** ID = *instance* **FROM** REC_RECTRANS **WHERE** RT_STDESC = *state* **AND** RT_PROCINST = *instance* Hourly wage rate: UM_USER (manually enriched by hourly wage rates) Completed instances: see separate fact sheet
Frequency	weekly
	Presentation
Addressees	Process Owner, Middle Management, Accountants (tbd specifically)
Presentation	As-is value and to-be value in combination with a sparkline showing the historical development, deviation from to-be value in %
Archiving	Stored in additional database table, linked with RT_-PROCDESC

Table 5.8: Fact Sheet for the Key Figure 'Average costs of a process activity'

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 BÃűrger'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.Ã§Ã§ 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 ObjectProperties 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 âĂŞ the state the subject is currently in as defined by a **State** in the model	*SID_state*	Execution concept âĂŞ no model representation, Not to be confused by a model âĂIJstateâĂİ in an SBD Diagram. State in the SBD diagram define possible SID_States.
SubjectBehavior âĂŞ 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*	âĂIJPassive ElementâĂİ of an edge in an SBD-graph
TransitionConditionn	*ExitCondition*	Static Concept that represents a Data condition
Execution Concept âĂŞ 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 BÃűrger	*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 âĂIJvariablesâĂİ known from programming languages, they can be read and written. Static functions are initialized before the execution, they can only be read. Derived functions "evaluateâĂİ other functions, they can only be read. âĂIJThey may be thought of as a global method with read-only variablesâĂİ

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 âĂŞ 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** âĂŞ 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** âĂŞ 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 (âĂIJmessage typeâĂİ) 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 âĂŞ can be restricted by **InputPoolConstraint** âĂŞ 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 âĂIJrememberâĂİ 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 âĂIJrememberâĂİ 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 âĂIJrememberâĂİ 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 textitCompulsory(altExit(D))	

Table B.6: Other Functions

B.7 Elements Not Covered not by BÃűrger (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 BÃűrger Interpreter does not assume the existence of such data elements as part of the model. However, the refinement concept of ASMs could easily been used to integrate according interpretation aspects. (Includes Elements such as **PayloadDescription** for Messages or **DataMappingFunction**

Table B.7: Other Functions

APPENDIX C

CoreASM PASS Reference Implementation

C.1 Conceptual Differences to the OWL Description

This reference implementation has some conceptual differences to the OWL description.

CS: is there an overall rationale for that?

For example it provides less flexible InputPool constraints and does not support synchronous communication, however it supports advanced features that are required for distributed inter-company business processes and provides concrete implementations for the usage of Subject Data.

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

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

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

This reference implementation adds a scope for Data Object Definitions, allowing a Data Object to be accessible either "globally" for all states in all behaviors of a Subject or only within a single Macro Behavior instance, allowing macros to execute concurrently without influencing

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

Table C.1: Key Conceptual Differences

each other. Further, this enables copying Data Ojects as arguments from the scope of a Macro State into the scope of the called Macro instance.

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

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

C.2 Architecture

The overall architecture is shown in figure C.1 and consists on the top level of a console application as user interface and the CoreASM Engine as execution environment for the PASS-interpreter specification. Both high level components are running in parallel in separate Java Virtual Machine processes. The PASS-interpreter specification file includes an ASM specification of asynchronous multi-agent ASMs, executing each Subject instance by one ASM-agent each.

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

The scheduler is used to support asynchronous multi-agent ASMs; with the *default* scheduling policy it selects a random subset of the running ASM agents and calls the interpreter for the

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

Table C.2: Further Conceptual Differences

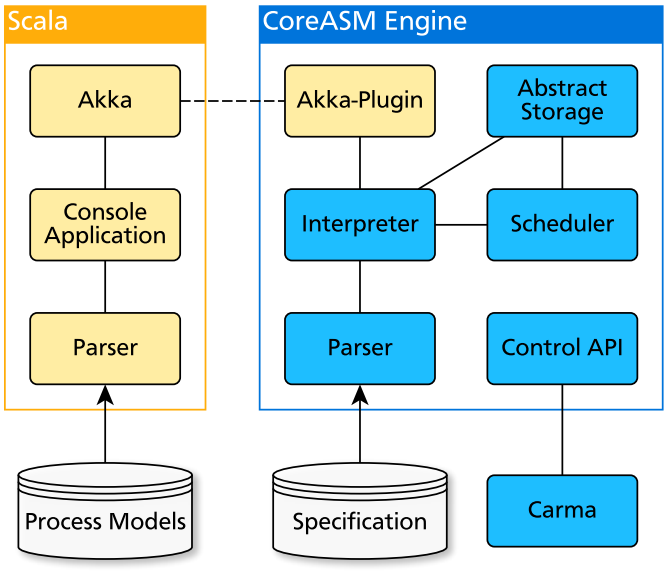


Figure C.1: Architecture of the reference implementation

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

Table C.3: Subject Data Conceptual Differences

parallel evaluation of their main rules. If the resultant update sets are consistent they are applied in the abstract storage, otherwise the scheduler selects a different subset of the running ASM agents for a re-evaluation. Supplementary the *onebyone* scheduling policy can be used to evaluate only a single agent per step and the *allfirst* policy can be used to attempt to evaluate all agents at once before falling back to the default scheduling policy in case of inconsistencies. The reference implementation uses the *allfirst* scheduling policy for performance reasons, but the *default* policy could be used as well.

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

The console application is developed in Scala. A user can load, start and execute locally stored PASS process model files with it. The interaction between the console application and the CoreASM engine is realized by Akka actors. The Akka actor of the engine has full read and write access to all CoreASM functions by using a Plug-In interface.

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

Table C.4: Observer Concept Differences

C.3 Editorial Note

To improve the typography the code blocks in this appendix have been compacted. Instead of writing the `rule Behavior` spread-out like in Listing 31 the code blocks are transformed to a more compact notation as shown in Listing 32.

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

Listing 31: Behavior, spread-out snippet

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

Listing 32: Behavior, compact snippet

Some elements are intentionally left out in order to preserve space. For example, the `rule InputPool_Pop` and the `derived availableMessages` function contain lengthy list traversals, and their definitions contain undocumented parameters used in the test environment for debugging. Other functions like `derived searchMacro` or `derived hasTimeoutTransition` should be self-explanatory.

```
function channelFor : Agents -> LIST

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

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

Listing 33: channelFor

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

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

Listing 34: activeStates

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

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

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

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

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

Listing 35: nextPI

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

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

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

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

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

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

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

Listing 36: initializedState

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

Listing 37: shouldTimeout

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

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

Listing 38: variable

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

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

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

Listing 39: macroTerminationResult

C.5 Interaction Definitions

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

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

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

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

Listing 40: inputPool

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

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

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

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

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

Listing 41: receivedMessages

C.6 Subject Rules

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

Listing 42: GenerateUniqueProcessInstanceID

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

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

Listing 43: StartProcess

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

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

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

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

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

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

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

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

Listing 44: StartASMAgent

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

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

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

      nextMacroInstanceNumber(ch) := MI  + 1

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

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

Listing 45: StartMainMacro

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

Listing 46: StartMacro

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

    ClearAllVarInMI(ch, MIAbort)
  }
```

Listing 47: AbortMacroInstance

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

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

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

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

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

Listing 48: SetExecutionState

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

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

Listing 49: AddState

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

Listing 50: SubjectBehavior

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

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

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

Listing 51: KillBehavior

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

        Behavior(MI, stateNumber)

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

        UpdateActiveStates(MI, stateNumber)
      endseqblock
```

Listing 52: MacroBehavior

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

    macroExecutionState(ch, MI) := DONE

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

    macroExecutionState(ch, MI) := DONE

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

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

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

Listing 53: UpdateRemainingStates

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

    addStates(ch, MI, stateNumber) := undef

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

        ResetState(xMI, xN)
      }

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

Listing 54: UpdateActiveStates

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

Listing 55: abortState

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

Listing 56: Behavior

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

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

Listing 57: AbortState

C.7 State Rules

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

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

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

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

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

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

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

Listing 58: StartState

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

    initializedState(ch, MI, s) := undef

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

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

    cancelDecision(ch, MI, s) := undef

    selectedTransition(ch, MI, s) := undef

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

    initializedSelectedTransition(ch, MI, s) := undef

    wantInput(ch, MI, s) := undef

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

Listing 59: ResetState

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

Listing 60: Perform

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

Listing 61: SelectTransition

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

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

Listing 62: StartSelectedTransition

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

Listing 63: PerformTransition

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

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

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

Listing 64: StartFunction

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

Listing 65: PerformFunction

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

Listing 66: AbortFunction

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

Listing 67: PerformTransitionFunction

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

    SetCompleted(MI, s)
  }
```

Listing 68: SetCompletedFunction

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

Listing 69: Proceed

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

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

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

Listing 70: StartTimeout

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

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

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

Listing 71: InitializeCompletion

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

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

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

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

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

Listing 72: EnableTransition

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

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

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

Listing 73: InitializeCompletionTransition

C.8 Internal Action

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

  EnableAllTransitions(MI, currentStateNumber)

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

Listing 74: StartInternalAction

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

Listing 75: PerformInternalAction

C.9 Send Function

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

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

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

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

Listing 76: StartSend

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

Listing 77: PerformSend

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

    SetCompleted(MI, s)
  }
```

Listing 78: TryCompletePerformSend

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

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

Listing 79: SelectReceivers

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

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

Listing 80: SelectReceivers_Selection

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

    SetAbortionCompleted(MI, s)
  }
```

Listing 81: AbortSend

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

      EnsureRunning(r)
    }

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

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

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

Listing 82: PerformTransitionSend

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

Listing 83: SetMessageContent

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

Listing 84: DoReservations

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

Listing 85: DoReservation

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

Listing 86: CancelReservation

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

Listing 87: ReplaceReservation

C.10 Receive Function

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

Listing 88: PerformReceive

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

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

Listing 89: PerformTransitionReceive

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

Listing 90: CheckIP

C.11 End Function

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

      FinalizeInteraction()

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

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

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

Listing 91: PerformEnd

C.12 Tau Function

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

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

Listing 92: Tau

C.13 VarMan Function

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

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

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

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

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

Listing 93: VarMan

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

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

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

  SetCompletedFunction(MI, currentStateNumber, undef)
}

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

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

Listing 94: VarMan_Assign

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

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

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

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

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

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

Listing 95: VarMan_Concatenation

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

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

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

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

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

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

Listing 96: VarMan_ExtractContent

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

function selectionResult   : LIST * NUMBER * NUMBER -> SET

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

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

    SetCompletedFunction(MI, currentStateNumber, undef)
  }

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

Listing 97: VarMan_Selection

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

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

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

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

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

Listing 98: Selection

C.14 Modal Split & Modal Join Functions

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

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

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

Listing 99: ModalSplit

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

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

Listing 100: ModalJoin

C.15 CallMacro Function

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

Listing 101: AbortCallMacro

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

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

Listing 102: InitializeMacroArguments

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

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

      SetExecutionState(MI, s, DONE)

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

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

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

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

Listing 103: CallMacro

C.16 Cancel Function

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

Listing 104: CheckCancel

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

Listing 105: Cancel

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

Listing 106: Abort

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

Listing 107: PerformTransitionCancel

C.17 IP FUNCTIONS

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

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

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

Listing 108: CloseIP

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

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

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

Listing 109: OpenIP

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

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

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

Listing 110: CloseAllIPs

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

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

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

Listing 111: OpenAllIPs

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

Listing 112: IsIPEmpty

C.18 SELECTAGENTS FUNCTION

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

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

function selectAgentsResult : LIST * NUMBER * NUMBER -> SET

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

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

Listing 113: SelectAgentsAction

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

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

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

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

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

Listing 114: SelectAgents

Bibliography

[Kee76] E. L. Keenan. 1976.

[LRS11] Alexander Lawall, Dominik Reichelt, and Thomas Schaller. Intelligente Verzeichnisdienste. In Thomas Barton, Burkard Erdlenbruch, Frank Herrmann, and Christian Müller, editors, *Herausforderungen an die Wirtschaftsinformatik: Betriebliche Anwendungssysteme*, AKWI 2011, pages 87–100, Berlin, 2011. News & Media.

[LRS14] Alexander Lawall, Dominik Reichelt, and Thomas Schaller. Propagation of agents to trusted organizations. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on*, volume 3, pages 74–77, August 2014.

[LSR13] Alexander Lawall, Thomas Schaller, and Dominik Reichelt. Integration of dynamic role resolution within the s-bpm approach. In *S-BPM ONE 2013*, pages 21–33, Heidelberg, 2013. Springer.

[LSR14a] A. Lawall, T. Schaller, and D. Reichelt. Local-global agent failover based on organizational models. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on*, volume 3, pages 74–77, Nov 2014.

[LSR14b] Alexander Lawall, Thomas Schaller, and Dominik Reichelt. Cross-organizational and context-sensitive modeling of organizational dependencies in c-org. In *S-BPM ONE (Scientific Research)*, pages 89–109, Heidelberg, 2014. Springer-Verlag.

[LSR14c] Alexander Lawall, Thomas Schaller, and Dominik Reichelt. Enterprise architecture: A formalism for modeling organizational structures in information systems. In Joseph Barjis and Robert Pergl, editors, *Enterprise and Organizational Modeling and Simulation*, volume 191 of *Lecture Notes in Business Information Processing*, pages 77–95. Springer Berlin Heidelberg, 2014.

[LSR14d] Alexander Lawall, Thomas Schaller, and Dominik Reichelt. Restricted relations between organizations for cross-organizational processes. In *Business Informatics (CBI), 2014 IEEE 16th Conference on*, volume 2, pages 74–80, July 2014.

[Sch98] Thomas W. Schaller. *Organisationsverwaltung in CSCW-Systemen*. Dissertation Universität Bamberg, 1998.